Clayton Escouper das Chagas
Cássia Blondet Baruque
Lúcia Blondet Baruque

Volume único

Java Básico e Orientação a Objeto

Fundação CECIERJ
Consórcio cederj

Fundação CECIERJ Extensão

Java Básico e Orientação a Objeto

Volume único

Clayton Escouper das Chagas
Cássia Blondet Baruque
Lúcia Blondet Baruque

GOVERNO DO Rio de Janeiro

SECRETARIA DE CIÊNCIA E TECNOLOGIA

TI Gestão Auditoria Governança

Apoio:

FAPERJ
Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro

FUNDAÇÃO SANTA CABRINI
Provedora de acesso à Cidadania

Fundação Cecierj / Extensão

Rua Visconde de Niterói, 1364 – Mangueira – Rio de Janeiro, RJ – CEP 20943-001
Tel.: (21) 2334-1569 Fax: (21) 2568-0725

Material Didático

ORGANIZAÇÃO
Cássia Blondet Baruque

ELABORAÇÃO DE CONTEÚDO
Clayton Escouper das Chagas
Cássia Blondet Baruque
Lúcia Blondet Baruque

Departamento de Produção

EDITORA
Tereza Queiroz

COPIDESQUE
Cristina Freixinho

REVISÃO TIPOGRÁFICA
Elaine Bayma
Daniela Souza
Janaína Santana
Thelenayce Ribeiro

COORDENAÇÃO DE PRODUÇÃO
Katy Araujo

PROGRAMAÇÃO VISUAL
Márcia Valéria de Almeida
Ronaldo d´Aguiar Silva

ILUSTRAÇÃO
Jefferson Caçador

CAPA
Sami Souza

PRODUÇÃO GRÁFICA
Oséias Ferraz
Verônica Paranhos

C433j

Chagas, Clayton Escouper das.
Java Básico e Orientação a Objeto: volume único / Clayton Escouper das Chagas, Cássia Blondet Baruque, Lúcia Blondet Baruque. – Rio de Janeiro: Fundação CECIERJ, 2010.
238p.; 19 x 26,5 cm.

ISBN: 978-85-7648-648-0

1. Linguagem de programação. 2. Java. 3. Programação gráfica. I. Baruque, Cássia Blondet. II. Baruque, Lúcia Blondet. III.Título.

CDD: 005.133

2010/1

Referências Bibliográficas e catalogação na fonte, de acordo com as normas da ABNT e AACR2.

Governo do Estado do Rio de Janeiro

Governador
Sérgio Cabral Filho

Secretário de Estado de Ciência e Tecnologia
Alexandre Cardoso

Universidades Consorciadas

UENF - UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO
Reitor: Almy Junior Cordeiro de Carvalho

UERJ - UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO
Reitor: Ricardo Vieiralves

UFF - UNIVERSIDADE FEDERAL FLUMINENSE
Reitor: Roberto de Souza Salles

UFRJ - UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Reitor: Aloísio Teixeira

UFRRJ - UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
Reitor: Ricardo Motta Miranda

UNIRIO - UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Reitora: Malvina Tania Tuttman

DEDICATÓRIA

Dedico este trabalho à memória da professora Cássia Blondet Baruque, principal responsável por minha participação neste importante projeto de ensino a distância junto ao CECIERJ/CEDERJ.

Seu dinamismo e competência deixaram ensinamentos a todos. Onde estiver, tenho certeza de que está feliz com o resultado alcançado, fruto das sementes por ela lançadas.

Java Básico e Orientação a Objeto

Volume único

SUMÁRIO

Prefácio

Para mostrarmos a importância do aprendizado da linguagem de programação Java, precisamos antes fazer uma retrospectiva sobre programação e suas linguagens através dos tempos, ate chegarmos em Java.

Sempre que converso sobre a história da computação em minhas aulas, digo que no seu nascimento, na década de 40 e depois da 2ª guerra, o computador era apenas uma máquina pouco amigável. A única forma de se comunicar com ele era através de linguagem de máquina – um tipo de programação de baixo nível – exclusiva para a máquina que estava sendo programada. Essa linguagem consistia de comandos binários ou hexadecimais, e isso era extremamente trabalhoso e improdutivo do ponto de vista da programação.

No ITA, em 1968, meu trabalho de fim de curso de graduação era um programa extremamente complexo feito em linguagem de máquina para o IBM 1620. Era uma verdadeira "tripa" de comandos em linguagem de máquina, para calcular a "Entropia da Língua Portuguesa", que determinava as frequências de grupamentos de letras da língua para otimizar a banda de um canal de comunicações. A implementação de qualquer tarefa ou suas modificações em linguagem de máquina era extremamente penosa.

Com o tempo, percebia-se que o programador gastava muito do seu trabalho se digladiando com comandos e instruções de máquina e pouco com a lógica do seu sistema. Um aluno de hoje pode achar uma loucura o que vou falar, mas, na evolução seguinte, para os Assemblers, a vida do programador melhorou muito. Foi uma mudança do purgatório para o paraíso, eram as chamadas Linguagens de Segunda Geração onde o programador podia usar nomes simbólicos para seus comandos e dados.

Ainda no ITA, minha experiência seguinte foi com o IBM 1130, no final da década de 1960 e início da de 1970. Os computadores ainda tinham pouca memória e poucos recursos, mas as linguagens de "alto nível" já eram mais acessíveis e a programação em linguagem de máquina ou Assembler eram somente complementares. No IBM 1130 a linguagem principal era o Fortran.

As linguagens de "alto nível" como FORTRAN e COBOL, eram chamadas de Linguagens de Terceira Geração, pois traziam um novo paradigma ao se libertarem da dependência dos códigos de máquina e favoreciam a descrição lógica dos procedimentos da programação. Durante esse tempo, na academia, as linguagens de programação estavam sendo formalizadas, em termos de suas gramáticas, sintaxes etc. Uma primeira dessas linguagens mais bem-nascidas foi a Algol 60 (Algorithmic Language). Outras linguagens dessa mesma época tiveram algum destaque em nichos específicos como, por exemplo, Simula, Lisp, Pascal, C, entre outras. Também na década de 1970 veio a onda da programação estruturada, dando uma nova perspectiva para a forma de programar com mais qualidade.

Entretanto, na prática, nesse tempo se programava muito em Cobol, Fortran e depois em PL/I (uma linguagem que era um "mix" de Fortran, Cobol e Algol). Mais tarde apareceu a linguagem BASIC que ficou famosa nos primeiros microcomputadores. BASIC é o acrônimo de *Beginners All-purpose Symbolic Instruction Code*, e nasceu de Fortran (acrônimo de *Formula Translation*). BASIC deveria ser a linguagem didática para iniciantes, mas não trazia nada de bom para programação organizada, ou seja, estruturada. Costumo dizer que quem aprendeu BASIC como sua primeira linguagem tem "problema de infância". Contudo, seu sucesso foi muito grande. Todos programavam em BASIC, tanto que até hoje temos linguagens originadas a partir dela, mesmo sendo as versões atuais muito diferentes de sua antepassada, como é o caso de Visual BASIC da Microsoft.

O importante nicho da área comercial, ou seja, as áreas bancária, atuária, seguros, comércio, entre outras que tinham a necessidade de utilizar recursos computacionais face à grande quantidade de informações e manipulação de dados que concentravam, ficou com COBOL. Hoje, muitos dos sistemas legados dessas áreas estão escritos nessa linguagem.

Com o passar do tempo, as Linguagens de Terceira Geração foram incluindo mais facilidades para a construção de programas bem modularizados e estruturados. A programação com essas linguagens foi difundida nos diversos computadores, que deixaram de ser luxo das grandes indústrias, universidades e órgãos militares e se tornaram muito mais capilares, tendo

seu ápice com a criação e disseminação dos computadores pessoais (PCs). Com o avanço tecnológico das linguagens, foi necessário facilitar o desenvolvimento de sistemas, já que a demanda crescera desesperadamente. Dessa forma, os paradigmas evoluíram e as técnicas de análise e projeto de sistemas se tornaram cada vez mais refinadas.

Portanto, as linguagens que começaram com instruções de máquina e Assemblers deram lugar às estruturadas. As linguagens estruturadas evoluíram sempre focadas na legibilidade e elegância de programação. Também trouxeram as facilidades de funções e sub-rotinas ou procedimentos e de utilização de bibliotecas para a modularização de sistemas. Em paralelo, a área de programação evoluía com novas técnicas e métodos para melhorar a qualidade dos sistemas produzidos. Isso tudo resultou em uma nova engenharia (Engenharia de *Software*).

Esta necessidade de melhoria trouxe mais um conceito, que apesar de ter nascido na década de 1970, só tomou forma e cresceu na década de 1980, e até hoje domina o projeto de qualquer linguagem que seja criada com o intuito de concorrer com as outras: a Orientação a Objetos. Costumo dizer que, com esse paradigma de programação, estamos chamando a atenção da importância da modelagem dos dados junto com seu comportamento. Segundo esse paradigma, o programa implementa as classes de objetos que melhor representam o enunciado do problema. Assim, os sistemas seriam projetados de forma mais intuitiva, retratando o mundo tal qual ele é.

Duas das primeiras linguagens desse novo paradigma, que se popularizaram foram C++ e Smalltalk. Essas linguagens introduziram o conceito de classes de objetos em hierarquias. A programação ficou mais hierarquizada e com isso surgiram novos conceitos e características que ajudaram muito na programação, garantindo-lhe mais elegância e simplicidade. A partir deste momento, o programador poderia se dedicar realmente a expressar sua lógica de negócios. Conceitos como herança, polimorfismo, encapsulamento e outros são até hoje parte principal da teoria da Orientação a Objetos em qualquer linguagem de programação desse gênero.

Paralelamente, um outro movimento tecnológico ganhou muita força na mesma época da consolidação da Orientação a Objetos como paradigma de programação: o mundo da computação nunca mais foi o mesmo depois do nascimento da *web*. Mas com ela também surgiram outros problemas, como a interoperabilidade das plataformas, já que os sistemas numa rede mundial são os mais heterogêneos possíveis. Para resolver isto, emergiram os estudos sobre máquinas virtuais. A Sun Microsystems tinha um projeto

inovador em uma linguagem denominada Oak, cujo slogan era: *write once, run anywhere*, (em português: escreva uma vez, rode em qualquer lugar). O novo contexto justificava o retorno desse projeto com força, e essa linguagem que trazia uma plataforma associada a ela mudou seu nome, devido a questões de direitos autorais, para o que seria a plataforma mais promissora do fim do século XX: Java.

Java podia ser executada em qualquer plataforma, era compilada uma única vez, numa única plataforma, gerando um código intermediário chamado *bytecode*, e este poderia ser interpretado por uma Máquina Virtual Java (JVM) em qualquer tipo de dispositivo, seja um computador, um celular, uma televisão ou até uma geladeira. Depois desse fenômeno os conceitos de programação nunca mais foram os mesmos, desde a década de 90, quando surgiu o Java, uma nova linguagem simples, elegante, orientada a objeto e portátil. Orquestrada por uma empresa poderosa chamada Sun *Microsystems* tem toda uma comunidade participativa e comprometida com os mais modernos conceitos de uma linguagem de programação associada à engenharia de *software*.

Essa força do Java se mostrava mais poderosa ainda quando íamos para o mundo "www", onde as limitadas e sem graça páginas html ganharam efeitos muito mais interessantes através de objetos dinâmicos Java chamados Applets. E com o tempo, a própria Sun viu a necessidade de abrir o Java, delegando-o completamente à sua comunidade, sempre organizada e rigorosa nos seus processos evolutivos. Através de seus vários comitês, a padronização virou um conceito forte dentro do Java, o que permitiu que este sobrevivesse e continuasse crescendo até hoje, mesmo sem ter um dono.

Dessa forma, evoluímos das sub-rotinas e simples organização de programas para bibliotecas de classes, onde enfatizamos o reuso através de organizações mais elaboradas dessas bibliotecas nas *APIs* e *Frameworks*. Não se escreve mais código a partir do zero, mas se reusam as classes especializadas em seus contextos. O desenvolvedor deixou de ser um programador braçal para ser um compositor, um projetista. E com isso ganhou-se muito mais velocidade e consistência de desenvolvimento, o que é necessário, pois a demanda é grande, e continua crescendo.

E assim continua a saga do Java, elegante, simples, modular, mas ao mesmo tempo abrangente e completo. Java é uma das linguagens mais importantes da atualidade, apesar de sua pouca idade (seu primeiro *release* 1.0 é de 1995) e não é mais uma promessa, mas uma realidade e com um futuro ainda muito longo e brilhante.

Nesse contexto, com uma linguagem que converge tantos aspectos fundamentais da computação, desde os conceitos de linguagem de programação e orientação a objetos, passando por engenharia de *software* e redes, vale ressaltar a importância do ensino de Java, onde este livro passa a ter papel importante pela forma didática que aborda o assunto, buscando o equilíbrio através de um conteúdo abrangente e completo, mas sem tirar a importância do cunho prático que deve nortear qualquer livro de programação.

O livro *Java Básico e Orientação a Objeto* serve como referência tanto para o programador iniciante, já que mostra com riqueza de detalhes todo o processo inicial de instalação e configuração dos ambientes necessários para desenvolver em Java, como para o programador mais experiente, já que trata com seriedade e profundidade outros assuntos mais avançados.

Enfim, tenho certeza que qualquer aluno que queira aprender a linguagem de programação Java, qualquer que seja o seu nível, vai agregar conhecimentos sólidos ao ler esse livro sobre esta tecnologia que está dominando os mercados de programação em várias áreas do desenvolvimento: *web*, celulares, TV digital, robótica, entre outras.

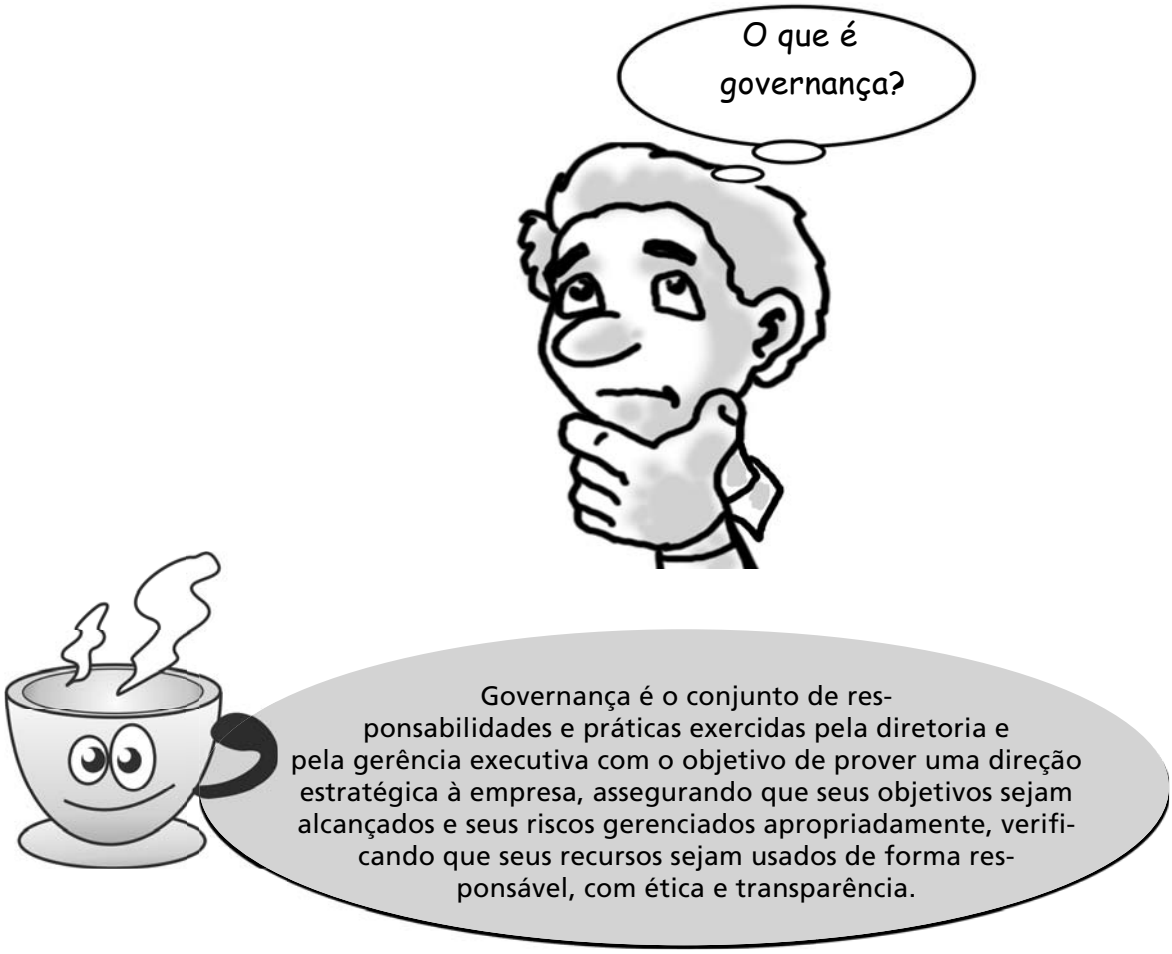
Prof. Dr. Rubens Nascimento Melo
PUC-Rio

Rubens Nascimento Melo possui graduação em Engenharia Eletrônica pelo Instituto Tecnológico de Aeronáutica, mestrado e doutorado em Ciência da Computação também pelo Instituto Tecnológico de Aeronáutica. Atualmente, é professor associado da Pontifícia Universidade Católica do Rio de Janeiro, onde leciona cursos de graduação e pós-graduação, orienta alunos de iniciação científica, mestrado e doutorado. Tem mais de quarenta anos de experiência na área de Ciência da Computação em geral e é um dos pioneiros nas áreas de Banco de Dados, Computação Gráfica e Interface de Usuário no Brasil. Seus temas de pesquisa principais são: integração semântica de informação, *data warehousing*, *Business Intelligence*, *e-learning* e sistemas de informação.

Introdução

"Escrever código uma vez e reutilizá-lo sempre que possível."

Você sabe o que é governança?



Repare que, em linhas gerais, o processo de governança nas empresas visa a responder a quatro questões básicas:

1. Se a empresa está fazendo as coisas certas.
2. Se a empresa está atuando de forma correta.
3. Se o uso de recursos é eficaz e eficiente.
4. Se os objetivos estabelecidos são alcançados.

Observe que o conceito de governança é relativamente novo, mas já se reconhece que boas práticas de governança aplicam-se a qualquer tipo de empreendimento. Pense e responda: quais são as três principais áreas do conhecimento que podem contribuir diretamente para uma boa governança?

Cada uma dessas áreas tem um objetivo definido dentro da governança:

- Gestão – estabelece um sistema de **controle** gerencial, bem como um ambiente que promova o alcance dos objetivos do negócio.
- Auditoria – avalia de forma independente a adequação e a eficácia dos **controles** estabelecidos pela gerência/diretoria.
- Tecnologia da Informação – apoia e capacita a execução dos **controles** do nível estratégico ao operacional.

A **governança corporativa** tornou-se um tema dominante nos negócios por ocasião dos vários escândalos financeiros ocorridos nos EUA em meados de 2002 – Enron, Worldcom e Tyco, para citar apenas alguns. A gravidade de tais escândalos abalou a confiança de investidores, realçando a necessidade das empresas de proteger o interesse de seus ***STAKEHOLDERS***. A governança corporativa tem a gerência de risco como um de seus principais componentes, que são: planejamento estratégico, liderança, definição de processos, acompanhamento e gerência de riscos.

STAKEHOLDERS

São aquelas pessoas ou instituições que possuem algum tipo de envolvimento profissional ou pessoal com uma empresa: investidores, clientes, funcionários, fornecedores, credores, acionistas, usuários, parceiros etc.

O PAPEL DE JAVA NA GOVERNANÇA

Em linhas gerais, vimos que a governança é o estabelecimento e administração de controle sobre um ambiente de modo a influenciar e garantir resultados, ações e comportamento desejados.

Observe na figura anterior que a TI é um dos três pilares que sustentam o conceito de governança. A TI é um pilar fundamental já que as informações disponibilizadas por ela sustentarão a sua empresa, pois todos os controles, processos, procedimentos e métricas serão apoiados pela TI. Note que não é possível haver uma gestão eficaz do negócio sem informações produzidas em tempo real e que sejam corretas, precisas e autorizadas.

Um dos desafios para TI é transformar os processos da empresa em "engrenagens", que funcionem de forma sincronizada e integrada. Com esse objetivo é necessário que as aplicações que suportam tais processos possam "se comunicar" e sejam desenvolvidas em uma linguagem de programação flexível, portátil e de fácil manutenção. Java vem de encontro a essa necessidade.

Java é uma linguagem de alto nível, simples, orientada a objeto, independente de arquitetura, robusta, segura, extensível, bem estruturada, ***MULTITHREADED*** e com coletor de lixo.

O termo ***MULTITHREADED*** refere-se a sistemas que suportam múltiplas linhas de execução.
Fonte: Wikipedia – A enciclopédia livre - http://pt.wikipedia.org/wiki/Thread_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o)

Algumas características importantes de Java:

- **SIMPLICIDADE:** Java, que se parece com a Linguagem C++, é simples de aprender, permitindo a você produtividade desde o início e possui um pequeno número de construtores.
- **ORIENTAÇÃO A OBJETO:** Em Java, os elementos de um programa são objetos, exceto os tipos primitivos. O código é organizado em classes, que podem estabelecer relacionamentos de herança simples entre si.
- **PROCESSAMENTO DISTRIBUÍDO:** Java permite o processamento distribuído através de classes que suportam a distribuição, dando suporte a aplicações em rede. Ademais, Java suporta vários níveis de conectividade através das classes contidas nos pacotes Java.net
- **PORTABILIDADE:** Java pode rodar em qualquer plataforma o que permite sua alta portabilidade. Isto se deve ao fato de que seu compilador não gera instruções específicas a uma plataforma, mas sim um programa em um código

intermediário. Tal programa, denominado *bytecode*, pode ser caracterizado como uma linguagem de máquina destinada a um processador virtual que não existe fisicamente.

- ***MULTITHREADING:*** Java fornece suporte a múltiplos threads de execução, que podem manipular diferentes tarefas.
- **COLETOR DE LIXO:** Sua função é a de varrer a memória de tempos em tempos, liberando automaticamente os blocos que não estão sendo utilizados, evitando, assim, erros na alocação de memória.

PORQUE JAVA É IMPORTANTE

Você sabia que a *Internet* vem se tornando cada vez mais popular e cresce a variedade de máquinas e sistemas operacionais disponíveis no mercado? Assim sendo, portabilidade, confiança e segurança, características que Java apresenta, tornam-se requisitos essenciais para uma linguagem de programação. Java oferece várias camadas de controle de segurança para a proteção contra código malicioso, além de possuir um compilador altamente rigoroso que evita erros básicos de programação, ajudando a manter a integridade do *software*, um dos conceitos essenciais em Governança. Além dessas características, há um diferencial que deve ser considerado: Java é uma solução gratuita.

Java é notoriamente um padrão na indústria de *software* e permite o desenvolvimento de aplicações na *Internet* para servidores, *desktop* e dispositivos móveis. A expectativa é de que as empresas vão continuar necessitando de profissionais, como você, com conhecimento em Java para atuarem no desenvolvimento de *software* e aplicativos, sendo que o conhecimento requerido deverá se expandir proporcionalmente à evolução de Java.

Saiba que a Fundação Cecierj trabalha para apoiar você nessa meta!

Representada pela professora Masako Masuda e pela professora Mirian Crapez, a Fundação ampliou a sua missão e incluiu no escopo do seu público-alvo os profissionais de mercado. No início de 2008, foi estabelecida uma nova área do conhecimento na Diretoria de Extensão voltada para oferta de cursos não só aos professores da Educação Básica, mas também aos demais profissionais, intitulada **Governança: Gestão, Auditoria e Tecnologia da Informação (TI)**.

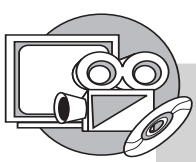
Um dos objetivos dessa área, projetada e implantada pelas professoras Lúcia Baruque e Cássia Baruque, é de que os profissionais do conhecimento, que são parte essencial da sociedade da informação, possam se capacitar e se atualizar em temas de ponta, incluindo aqueles que moram no interior do estado.

Sociedade da informação é aquela cujo modelo de desenvolvimento social e econômico baseia-se na informação como meio de criação do conhecimento. Ela desempenha papel fundamental na produção de riqueza e na contribuição para o bem-estar e a qualidade de vida dos cidadãos. Tal sociedade encontra-se em processo de formação e expansão.

Profissional do conhecimento é um termo adaptado de *knowledge worker*, referindo-se a profissionais que trabalham diretamente com a informação, produzindo ou fazendo uso do conhecimento, em oposição àqueles envolvidos na produção de bens ou serviços. São analistas de sistemas, programadores, desenvolvedores de produtos ou pessoal ligado à tarefa de planejamento e análise, bem como pesquisadores envolvidos principalmente com a aquisição, análise e manipulação da informação.
Esse termo foi popularizado pelo *guru* Peter Drucker.

Agora que você está familiarizado com o conceito de governança e com a importância de adquirir conhecimentos em Java, mãos à obra. Vamos aprender muito a seguir!

Bons estudos!



Note que este material foi projetado para ser autoinstrucional. Entretanto, você pode interagir com outros profissionais da área ou mesmo realizar atividades adicionais com *feedback* do tutor ao participar do nosso curso no ambiente virtual de aprendizagem da Fundação. Para mais informações, acesse: http://www.cederj.edu.br/extensao/governanca/index.htm

INFORMAÇÕES SOBRE O CURSO

Público-alvo
Estudantes, professores e profissionais das áreas de engenharia, computação, exatas e afins.

Pré-requisito
Noções de lógica de programação (ter algum conhecimento em qualquer linguagem de programação, exceto HTML) e sistema operacional Windows XP ou Windows Vista instalado.

Objetivos
Capacitar o participante no entendimento da arquitetura e programação da linguagem Java, com a finalidade de prepará-lo para suprir a crescente necessidade do mercado pelo profissional especialista nessa tecnologia, desenvolvendo projetos flexíveis, reutilizáveis e com padrões de qualidade reconhecidos.

Ementa

- Introdução ao Java Standard Edition
- Tipos de dados do Java
- Estrutura de Programação em Java
- Estruturas de Controle
- Arrays
- Java Orientado a Objetos: Classes, Atributos, Métodos, Herança, Polimorfismo, Classes Abstratas e Interfaces
- Controle de Erros
- Tratamento de Exceções
- Bibliotecas do Java Standard Edition
- Programação Gráfica em Java: Swing e AWT

Carga horária: 45 horas aula

Masako Oya Masuda

Presidente da Fundação Centro de Ciências e Educação Superior a Distância do Estado do Rio de Janeiro (Cecierj), vinculada à Secretaria de Estado de Ciência e Tecnologia. Possui graduação em Ciências Biológicas pela Universidade de São Paulo, mestrado em Ciências Biológicas (Biofísica) pela Universidade Federal do Rio de Janeiro, doutorado em Ciências Biológicas (Biofísica) pela Universidade Federal do Rio de Janeiro e pós-doutorado na University of California. Tem experiência na área de Fisiologia, com ênfase em Fisiologia Geral.

Mirian Araujo Carlos Crapez

Vice-presidente de Educação Superior a Distância da Fundação Cecierj. Graduada em Ciências Biológicas pela Universidade Federal de Minas Gerais, com doutorado em Metabolismo de Aromáticos e Poliaromáticos realizado na Université D'Aix-Marseille II e pós-doutorado na Université Paris VI (Pierre et Marie Curie). Atualmente, é professora associada da Universidade Federal Fluminense.

Lúcia Blondet Baruque

Mestre e doutora em Informática pela PUC-Rio; bacharel em Ciências Econômicas pela UFRJ; possui *Certificate in Management* pela John Cabot University (Roma). Foi professora na PUC-Rio e atuou no seu Centro de Educação a Distância. Atualmente, é professora associada da Fundação Cecierj, coordenadora da área de Governança: Gestão, Auditoria e TI e pesquisadora associada do Laboratório de Tecnologia em Banco de Dados da PUC-Rio. Trabalhou na auditoria da Exxon Company International e na ONU, em Roma, bem como na diretoria do Instituto dos Auditores Internos do Brasil. Membro do Institute of Internal Auditors. CIA, CISA Exam.

Cássia Blondet Baruque

Mestre e doutora em Informática pela PUC-Rio. Foi docente e atuou como cocoordenadora da área de Governança: Gestão, Auditoria e TI da Fundação Cecierj e como pesquisadora associada do Laboratório de Tecnologia em Banco de Dados da PUC-Rio. Trabalhou como pesquisadora e professora na PUC-Rio, IMPA, FGV-online e UEZO em temas como e-learning, bibliotecas digitais, data warehousing/OLAP e mineração de dados, além de ter exercido cargos importantes, o que lhe conferiu grande experiência em desenvolvimento de sistemas, nas empresas Fininvest, Cyanamid, Banco Nacional, Capemi, Shell e RFFSA.

Clayton Escouper das Chagas

Engenheiro de Telemática pelo Instituto Militar de Engenharia (IME/RJ), pós-graduado em Sistemas Modernos de Telecomunicações pela Universidade Federal Fluminense (UFF) e mestrando em Sistemas de Telecomunicações com estudos na área de TV Digital Interativa pela Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Trabalha como engenheiro em projetos de Pesquisa e Desenvolvimento (P&D) no Centro Tecnológico do Exército e é instrutor de Java em empresas privadas e universidades do Rio de Janeiro desde 2005.

AULA 1

Introdução ao Java

Meta da aula

Apresentar as principais características da plataforma Java.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. descrever o histórico da tecnologia Java;
2. descrever as características do Java;
3. identificar os componentes da plataforma Java Standard Edition;
4. criar e identificar as principais estruturas de uma classe em Java.

Pré-requisitos

Para o correto e completo aprendizado desta aula, é preciso que você: tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK), conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans, conforme as instruções do Apêndice B; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

INTRODUÇÃO

Você já ouviu falar na ilha de Java? Além de uma das tecnologias mais importantes da computação, Java também é uma das mais famosas ilhas do oceano Índico, e foi justamente essa ilha que deu origem ao nome da linguagem de programação Java.

Pesquise mais em http://pt.wikipedia.org/wiki/Java e descubra mais sobre a ilha de Java antes de entrar no mundo fantástico da tecnologia Java.

O Java como linguagem de programação nasceu em 1991 pelas mãos de um pesquisador da Sun Microsystems, James Gosling.

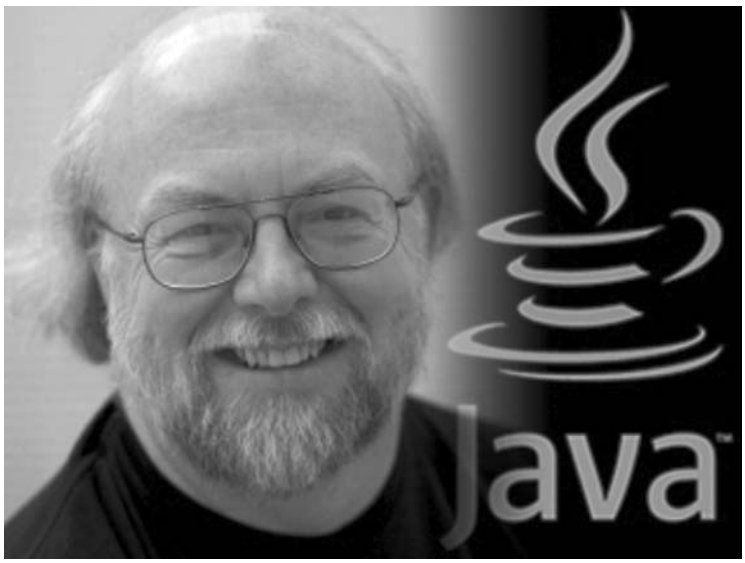


Figura 1.1: James Goling, o criador do Java.
Fonte: http://cnettv.cnet.com/2001-1_53-27469.html

A Sun Microsystems é uma das maiores empresas de tecnologia do mundo. Atua nos setores de fabricação de computadores, semicondutores e *software*. Sua sede fica em Santa Clara, Califórnia, no Silicon Valley (Vale do Silício). As fábricas da Sun localizam-se em Hillsboro, no Oregon, EUA, e em Linlithgow, na Escócia.
O nome Sun vem de Stanford University Network (Rede da Universidade de Stanford).
Os produtos da Sun incluem servidores e estações de trabalho (*workstations*) baseados no seu próprio processador SPARC, no processador Opteron, da AMD, nos sistemas operacionais Solaris e Linux, no sistema de arquivos de rede NFS e na plataforma Java.
Saiba mais sobre a Sun Microsystems em http://pt.wikipedia.org/wiki/sun_microsystems. É interessante conhecer a empresa que criou a tecnologia que você irá estudar.

O nome inicial da linguagem era *Oak* (carvalho, em português), pois existiam muitos carvalhos na área da empresa e eles eram visíveis pela janela dos programadores.

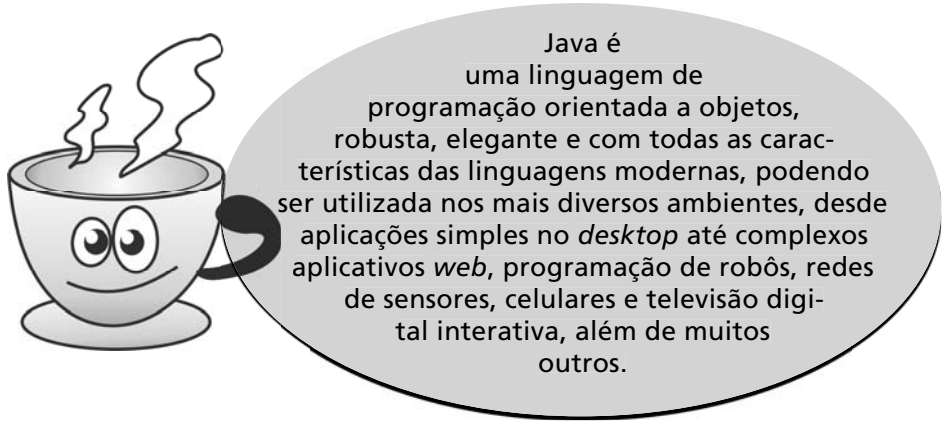
Devido a problemas de direitos autorais, o nome acabou sendo alterado para Java, pois já existia uma linguagem chamada Oak.

A ideia original do projeto era desenvolver uma linguagem de programação que fosse independente de plataforma e que pudesse ser executada em diferentes dispositivos eletrônicos, como geladeiras, televisões ou fogões.

Inicialmente, o projeto não foi prioridade para a Sun, já que alguns executivos acharam absurda a ideia de ter uma linguagem que pudesse ser processada numa televisão. Com a chegada da TV digital interativa, podemos ver que não estavam tão errados assim... Mas com o sucesso da internet e a necessidade de linguagens que suportassem objetos dinâmicos e independentes de plataforma, a empresa viu uma oportunidade para o emprego da tecnologia que o projeto *Oak* havia desenvolvido.

A TECNOLOGIA JAVA

Linguagem de programação



Ambiente de desenvolvimento

Como desenvolvedor Java, você terá à sua disposição um conjunto de ferramentas poderosas capazes de abranger várias tarefas envolvidas no processo de desenvolvimento de *software* dentro do seu ciclo de vida.

Dentre as principais ferramentas disponíveis, podemos citar:

- o compilador (javac);
- o interpretador (java);
- o gerador de documentação (javadoc);
- a ferramenta de compactação de arquivos (jar);
- diversas outras ferramentas instaladas no diretório bin da distribuição.

JAVA VIRTUAL MACHINE (JVM)

Máquina virtual que faz com que o seu programa em Java se comunique com o sistema operacional e com o *hardware* do computador.

Ambiente de aplicação

A forma de execução da linguagem Java é baseada na interpretação por meio de uma máquina virtual, a **JAVA VIRTUAL MACHINE (JVM)**. Ela proporciona um ambiente de aplicação completo para execução de aplicativos Java.

Atividade prática 1

Vamos pesquisar um pouco?
Acesse, no *site* da Sun, as áreas sobre Java e descubra por que ele é uma das tecnologias que mais intrigam o mundo moderno de tecnologia: http://java.sun.com.

RAIO X DO JAVA

O desenvolvimento em Java é dividido em três grandes plataformas, de acordo com as características e os objetivos do sistema implementado:

- A **primeira plataforma**: é o *core* de todo sistema Java; é utilizada como base da linguagem, além de possuir algumas bibliotecas para desenvolvimento de aplicações gráficas, programação em redes e processamento paralelo. A esse conjunto *core* damos o nome de **JAVA STANDARD EDITION (JSE)**.

JAVA STANDARD EDITION (JSE)

Pacote-base para o desenvolvimento de programas em Java.

JAVA ENTERPRISE EDITION (JEE)

Pacote para desenvolvimento de programas para o segmento corporativo, programação distribuída e programação *web*.

A **segunda plataforma**: engloba desenvolvimento *web*, programação distribuída corporativa, serviços corporativos especiais como Web Services e Enterprise Java Beans, além de bibliotecas de gerenciamento. Essa plataforma se chama **JAVA ENTERPRISE EDITION (JEE)** e tem como requisito o próprio JSE.

A **plataforma de desenvolvimento para dispositivos móveis e embarcados**: dá suporte a toda programação para celulares, televisão digital e outros dispositivos com menor poder de processamento; é a plataforma **JAVA MICRO EDITION (JME)**.

JAVA MICRO EDITION (JME)

Pacote para desenvolvimento de programas para dispositivos móveis como celulares, *palm tops* etc.

Nosso curso será focado no JSE (primeira plataforma), que é a base para que o aluno prossiga no estudo de plataformas mais complexas.

Veja a seguir os módulos que compõem o JSE:

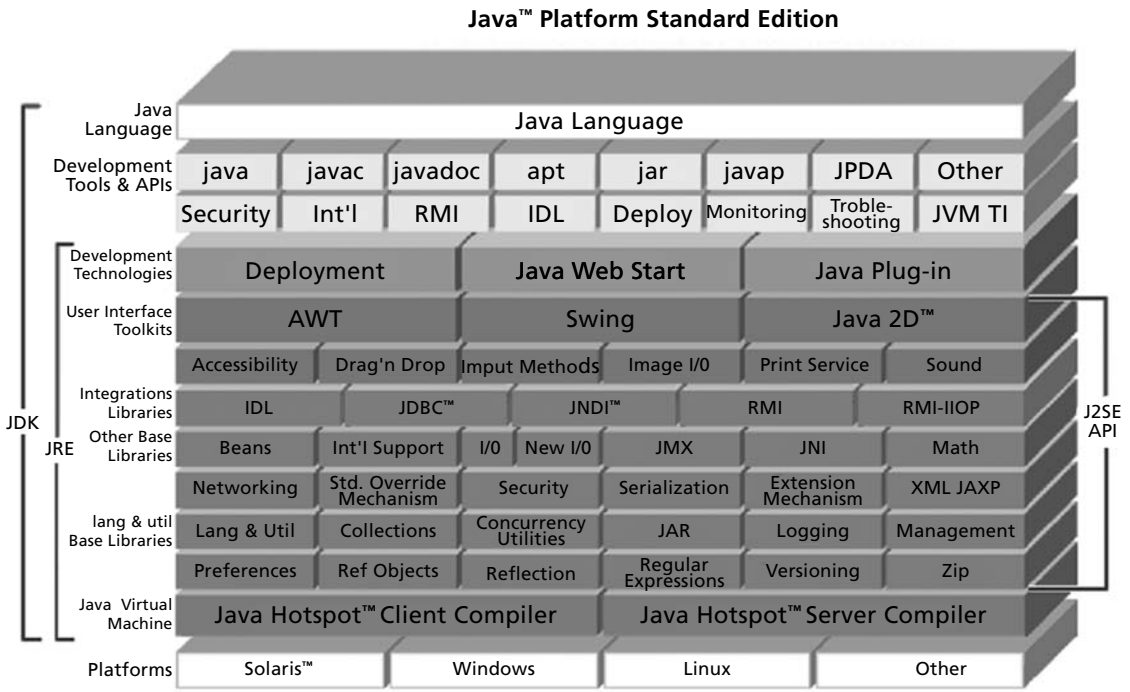


Figura 1.2: Estrutura do Java Standard Edition.
Fonte: javadocs - jdk 1.6.03.

CARACTERÍSTICAS DO JAVA

Java Virtual Machine (JVM)

Imagine uma máquina criada por meio de um *software* emulador dentro de uma máquina real, como se fosse um minicomputador dentro do seu próprio computador, e que possui gerenciamento de memória, CACHE, THREADS e outros processos de um computador com seu sistema operacional. Exatamente! Essa máquina existe e se chama Máquina Virtual Java, ou JVM.

A Máquina Virtual Java provê especificações de plataforma de *hardware* na qual se compila todo código de tecnologia Java. Essas especificações permitem que o *software* Java seja uma plataforma independente, pois a compilação é feita por uma máquina genérica, a JVM.

O resultado de uma compilação de um código fonte Java é o *bytecode*, que é uma linguagem de máquina inteligível para a JVM. O *bytecode* é independente de *hardware*; assim, basta o computador ou o dispositivo eletrônico (como um celular ou televisão) ter o interpretador adequado (a JVM) que poderá executar um programa Java compilado, não importando em que tipo de computador ele foi compilado.

CACHE

Subsistema especial da memória de alta velocidade, utilizado para armazenar dados temporariamente até que eles sejam solicitados. Os dados utilizados com mais frequência são copiados para ela, permitindo acesso mais rápido. É uma forma de acelerar a memória, o processador e as transferências do disco. Os caches de memória armazenam o conteúdo das posições mais atualizadas da RAM e os endereços onde esses dados estão armazenados.
Fonte: http://www.aoli.com.br/dicionarios.aspx?palavra=Cache.

Garbage Collection

THREADS

Processo de divisão de tarefas em programação de computadores.
Fonte: http://pt.wikipedia.org/wiki/Thread.

Visualize a memória do seu computador como uma grande rua onde uma equipe de gandulas de tempos em tempos vai limpando tudo que não está sendo utilizado pelo Java. Isso mesmo. Essa equipe presta ao sistema operacional um importante serviço de coleta de lixo, que chamamos de *Garbage Collection*.

Muitas linguagens de programação permitem ao programador alocar memória durante o tempo de execução. Entretanto, após utilizar a memória alocada, deve existir uma maneira para desalocar o bloco de memória, de forma que os demais programas a utilizem novamente. Em C, C++ e outras linguagens, o programador é o responsável por isso, o que às vezes pode ser difícil, já que os programadores podem esquecer de desalocar instâncias da memória e resultar no que chamamos escapes da memória.

No Java, você, programador, não possui a obrigação de retirar das áreas de memória uma variável criada. Isso é feito por uma parte específica da JVM que chamamos de *Garbage Collection*. A *Garbage*

Collection é a grande responsável pela liberação automática do espaço em memória. Isso acontece automaticamente durante o tempo de vida do programa Java.

Java Runtime Environment

A implementação de uma máquina Java juntamente com a JVM para o usuário final dos sistemas é feita por meio de um pacote chamado Java Runtime Environment (JRE). O JRE roda códigos compilados para a JVM e executa o carregamento de classes (por meio do Class Loader), a verificação de código (por meio do verificador de *bytecode*) e finalmente o código executável.

O Class Loader é responsável por carregar todas as classes necessárias ao programa Java. Isso adiciona segurança, por meio da separação do *namespace* entre as classes do sistema de arquivos local e aquelas que são importadas pela rede. Isso limita qualquer ação de programas que podem causar danos, pois as classes locais são carregadas primeiro. Depois de carregar todas as classes, a quantidade de memória que o executável irá ocupar é determinada. Isso acrescenta, novamente, proteção ao acesso não autorizado de áreas restritas ao código, pois a quantidade de memória ocupada é determinada em tempo de execução.

Após carregar as classes e definir a quantidade de memória, o verificador de *bytecode* verifica o formato dos fragmentos de código e pesquisa, nesses fragmentos, códigos ilegais que possam violar o direito de acesso aos objetos.

Depois que tudo isso tiver sido feito, o código é finalmente executado.

CICLO DE EXECUÇÃO DE UM PROGRAMA EM JAVA

Você vai entender agora como é o fluxo de execução de um programa em Java, desde sua codificação até sua interpretação pela máquina virtual.

O primeiro passo para a criação de um programa Java é escrever os programas em um editor de texto. Qualquer tipo de editor pode ser utilizado, como o bloco de notas, vi, emacs etc. A única obrigatoriedade é que esses arquivos devem ser armazenados no disco rígido com a extensão **.java**.

Após o programa Java ter sido criado e salvo, você deve compilá-lo utilizando o compilador Java (javac). A saída desse processo é um arquivo de *bytecode* com extensão **.class**.

O arquivo **.class** é então lido pelo interpretador Java (java – JVM), que converte os *bytecodes* em linguagem de máquina do computador ou dispositivo que está sendo usado.

A figura a seguir ilustra todo o processo de execução de um programa em Java.

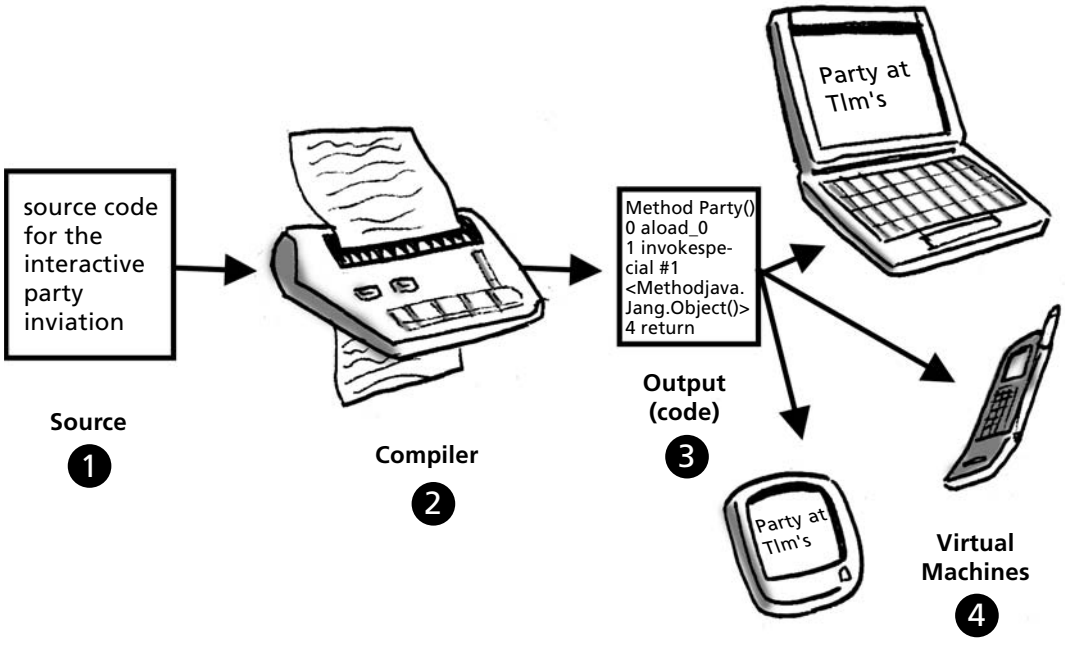


Figura 1.3: Processo de execução de um programa em Java.
Fonte: Livro *Use a cabeça! Java*.

Atividade prática **2**

Vamos começar a programar!

Para compilar e executar um programa Java corretamente, você deve ter o JDK (Java Development Kit) instalado. Além disso, as variáveis de ambiente do Windows devem estar corretamente configuradas. Execute as tarefas de download, instalação e configuração do JDK seguindo as orientações do Apêndice A. A partir de então, você poderá testar seus códigos Java.

ESCREVENDO SEU PRIMEIRO PROGRAMA EM JAVA

Utilizando o prompt do Windows e o Bloco de Notas

Você vai agora construir o primeiro exemplo de programa em Java. A compilação e a execução do programa serão feitas pelo *prompt* de comando do Windows.

Passo 1: Inicie o Bloco de Notas do Windows seguindo os comandos *Iniciar* => *Programas* => *Acessórios* => *Bloco de Notas*.

Passo 2: Abra o *prompt* de comando do Windows seguindo os comandos *Iniciar* => *Programas* => *Acessórios* => *Prompt de Comando*.

Passo 3: No Bloco de Notas, digite as seguintes instruções de um simples programa em Java:

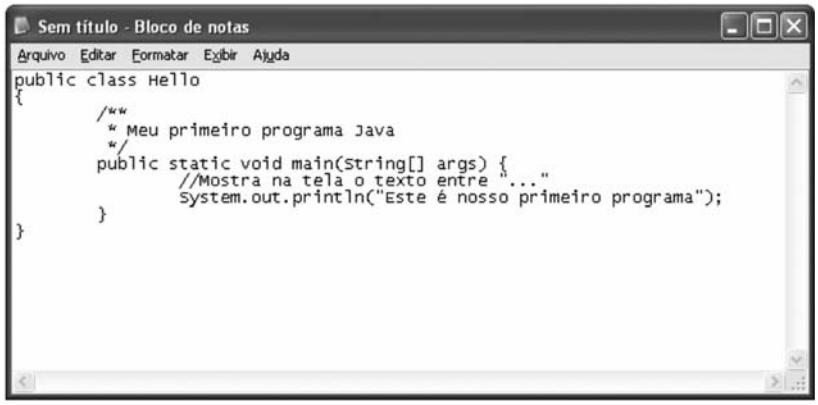


Figura 1.4: Primeira classe em Java.
Fonte: Bloco de Notas.

Seja rigoroso em todos os símbolos utilizados, principalmente no que se refere a letras maiúsculas e minúsculas, pois fazem diferença no Java.

Passo 4: Salve o programa como "Hello.java" e coloque numa pasta chamada "MeusProgramas".

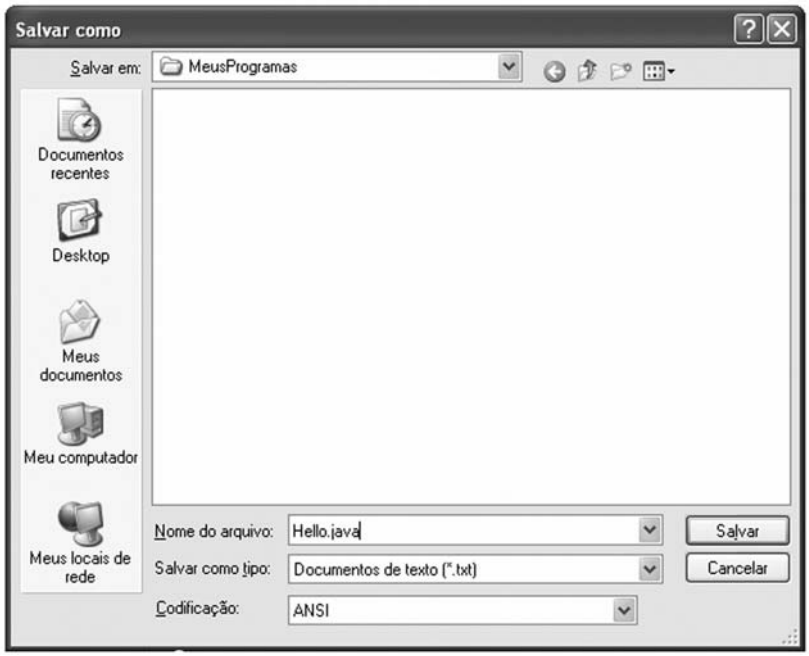


Figura 1.5: Salvando sua primeira classe em Java.
Fonte: Bloco de Notas.

Passo 5: Vá para o *prompt* de comando e navegue até a pasta "MeusProgramas".

Passo 6: Nessa pasta, digite o comando *dir* para verificar se o seu arquivo *Hello.java* está dentro do diretório.

Se estiver tudo certo, já podemos compilar nosso primeiro programa em Java.

Passo 7: Para compilar um programa Java, devemos executar o compilador Java (javac) passando na linha de comando o nome do programa a ser compilado; no nosso caso, o *Hello.java*.

Os comandos a serem executados são os mostrados na figura do *prompt* a seguir. Se não tiver nenhum erro, o cursor do *prompt* aparecerá novamente na linha de baixo.

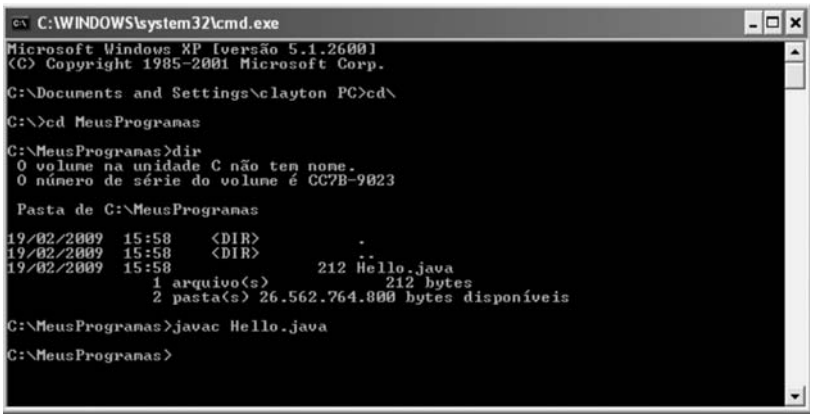


Figura 1.6: Compilando sua primeira classe em Java.
Fonte: Prompt de Comando.

Passo 8: Digite novamente o comando *dir* para verificar se a compilação foi realizada normalmente. Como você já viu, a compilação de um arquivo *.java* tem como resultado o *bytecode*, que é um arquivo *.class* com o mesmo nome, como mostrado na figura a seguir:

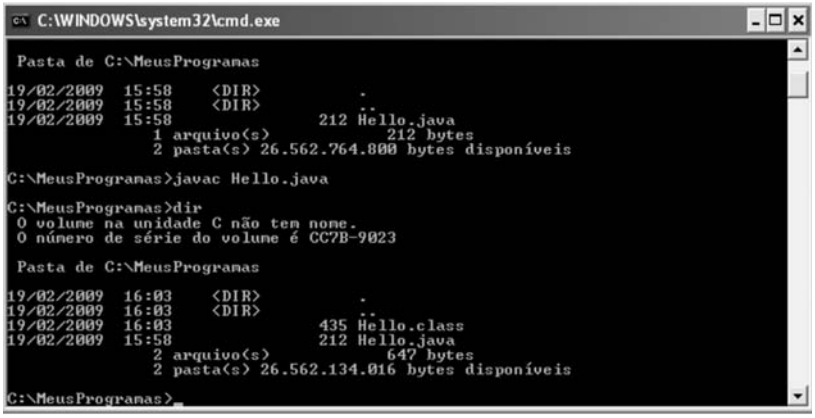


Figura 1.7: Verificando a classe compilada.
Fonte: Prompt de Comando.

Passo 9: Por último, chame a Máquina Virtual Java (JVM) por meio do comando *java*, passando na linha de comando o arquivo do *bytecode*, através do nome do arquivo gerado na compilação, dessa vez sem a extensão (sem o **.class**) como mostrado a seguir. O *bytecode* é interpretado pela JVM e a saída do texto digitado é realizada pelo *prompt*, conforme foi solicitado no código do programa.

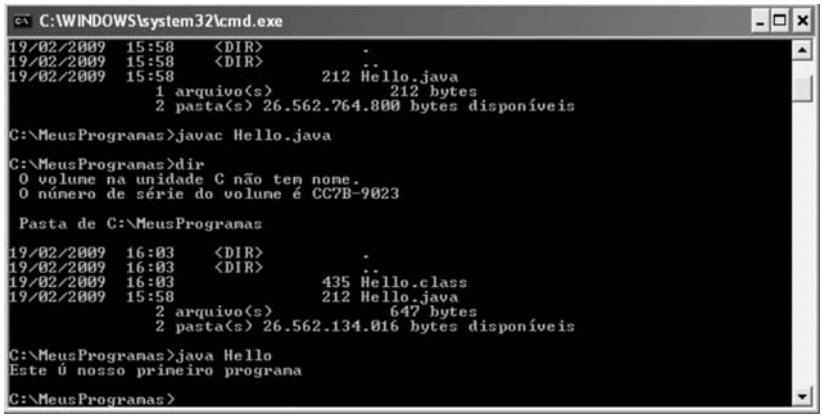


Figura 1.8: Executando sua primeira classe em Java.
Fonte: Prompt de Comando.

Atividade prática 3

Vamos começar a programar!

Siga os passos anteriores e veja você mesmo seu primeiro programa em Java funcionando!

Utilizando um Ambiente de Desenvolvimento (IDE)

Para continuar seu aprendizado de forma produtiva, iremos empregar um ambiente profissional, que é o mesmo utilizado nas empresas de desenvolvimento de *software*: usaremos o ambiente de desenvolvimento NetBeans, da Sun Microsystems. Para isso, você precisará executar as tarefas de *download*, instalação e configuração do NetBeans que estão explicadas de forma detalhada no Apêndice B.

Apesar de muitos dos nossos programas não exigirem ferramentas sofisticadas para programação, como você viu no último exemplo, profissionalmente as empresas necessitam de um ambiente integrado com algumas características, como maior velocidade de programação, automatização de processos e testes, além da necessidade de melhorar a produtividade e o gerenciamento das equipes. Nesse contexto, pode ser interessante a utilização de ferramentas especializadas que controlem todos esses fatores, trazendo vantagem considerável no ciclo produtivo das empresas de *software*. Essas ferramentas se chamam ambiente de desenvolvimento integrado (*integrated development environment*), ou simplesmente IDE.

Atividade de reflexão 1

Pesquise um pouco!

Pesquise em sites de busca (como www.google.com.br) e em informações colaborativas (como www.pt.wikipedia.org) sobre a linguagem de programação Java e também sobre IDE.

Depois da pesquisa, escreva com suas próprias palavras nas linhas a seguir o que entendeu sobre a linguagem Java e também sobre IDE.

Java:

_____

_____

_____

_____

_____

IDE:

_____

_____

_____

_____

_____

Vamos utilizar nesta disciplina uma IDE produzida pela própria Sun (que criou o Java) e uma das mais utilizadas pelas empresas de desenvolvimento no mundo: o NetBeans, na sua versão 5.5.1. Entre os vários recursos que a IDE possui, podemos citar como principais a interface construtora, o editor de texto, o editor de código, o compilador, o interpretador e o depurador. Além disso, a IDE possui muitas outras funcionalidades e *wizards* que auxiliam você a dar mais produtividade a seu trabalho e a reduzir a quantidade de erros. Para utilizar o NetBeans, execute os seguintes passos:

Passo 1: Inicie o NetBeans clicando no ícone de atalho *NetBeans 5.5.1* que se encontra na área de trabalho do computador e espere a abertura de sua interface gráfica.

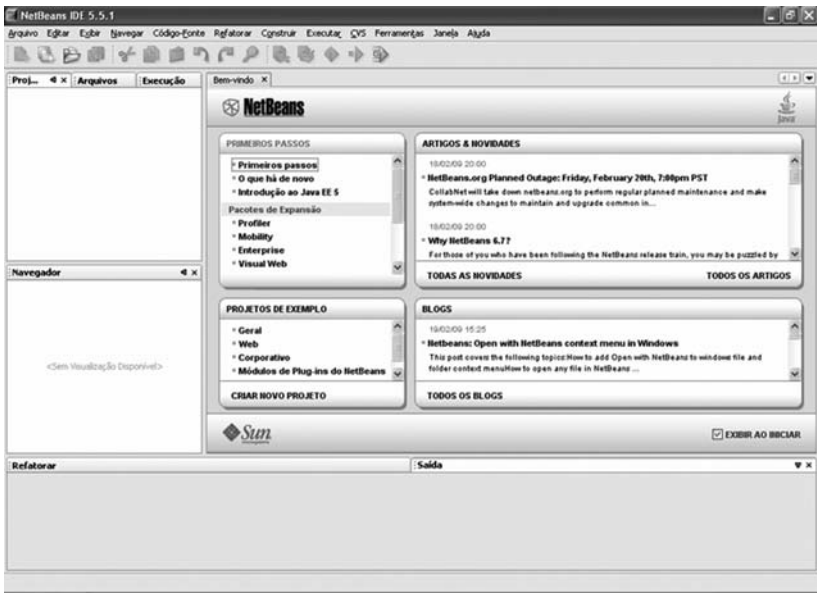


Figura 1.9: Tela inicial do NetBeans.
Fonte: NetBeans.

Passo 2: Feito isso, vamos construir nosso primeiro projeto, já que todo programa Java numa IDE deve estar associado a um projeto, através da sequência *Arquivo* => *Novo Projeto*, e na caixa de diálogo que se abrir escolha na janela *Categorias* a opção *Geral* e na janela *Projetos* a opção *Aplicação Java*.

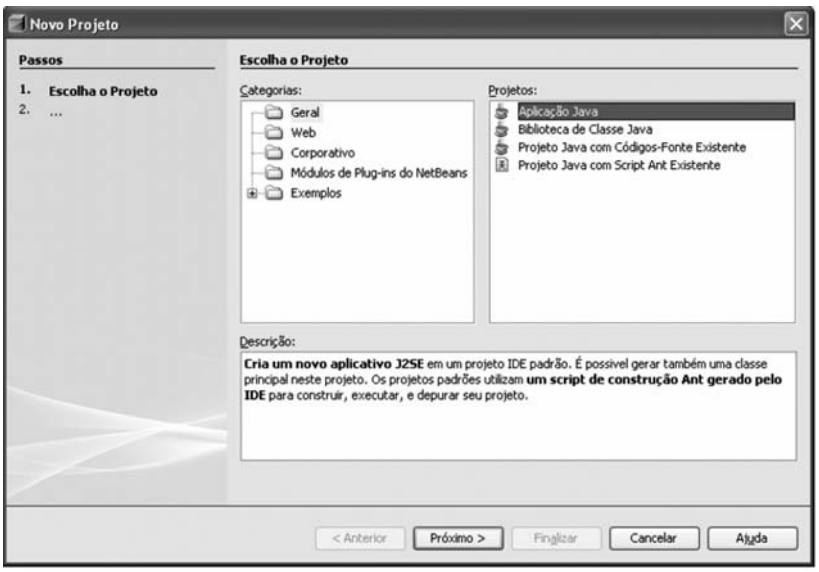
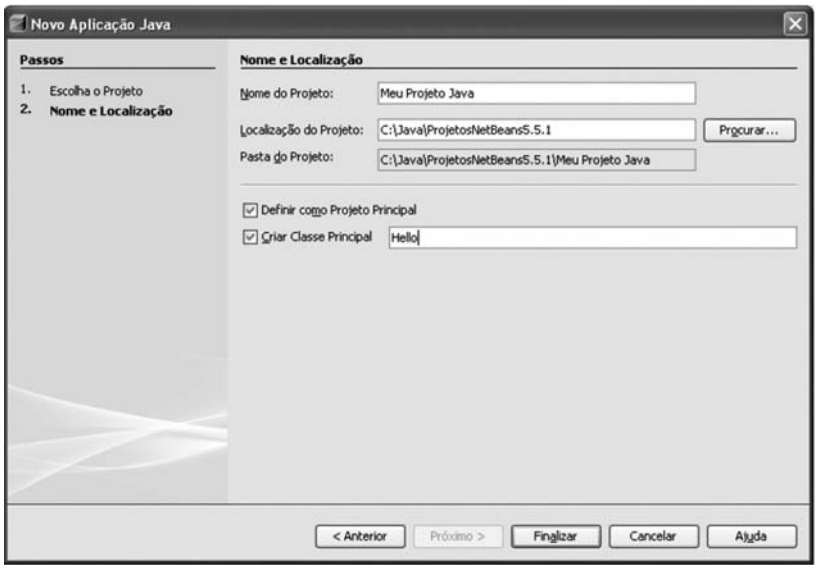


Figura 1.10: Criando um projeto no NetBeans.
Fonte: NetBeans.



Passo 3: Clique no botão *Próximo* e na caixa de diálogo seguinte preencha o *Nome do Projeto* com *Meu Projeto Java* e ao lado da *checkbox Criar Classe Principal* defina seu nome como *Hello* e clique no botão *Finalizar*.

Figura 1.11: Criando um projeto no NetBeans.
Fonte: NetBeans.

Passo 4: Na janela que se abre, a IDE insere um código básico de todo programa Java completado com algumas informações que foram inseridas durante a criação do projeto. Complete esse código com o texto do programa feito no Bloco de Notas.

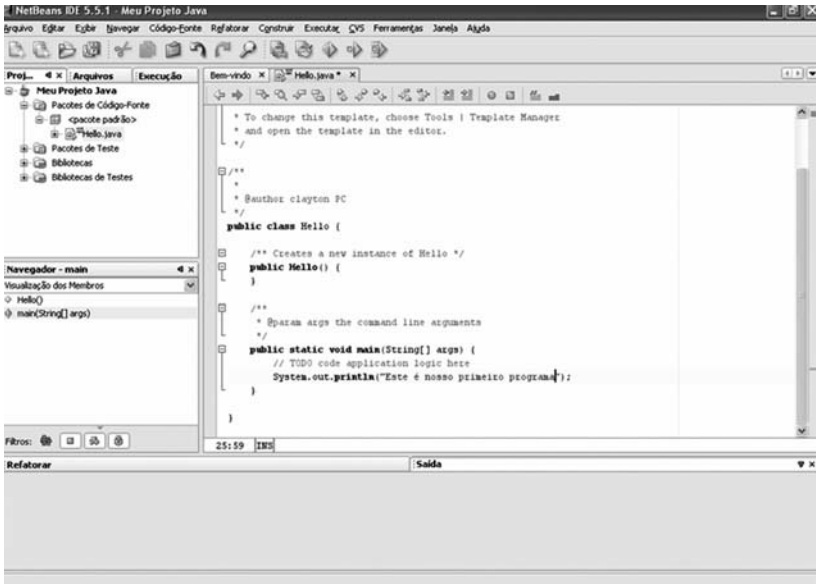


Figura 1.12: Criando uma classe no NetBeans.
Fonte: NetBeans.

Ao mandar executar um programa, a IDE automaticamente compila e interpreta o código, mostrando o resultado dos comandos, caso seja uma aplicação via console (que será o nosso caso, por enquanto), na janela abaixo do código.

Passo 5: Para executar o código, devemos selecionar o programa que desejamos executar na janela do lado esquerdo superior, janela *Projetos*, e clicar com o botão direito do mouse selecionando a opção *Executar Arquivo*.

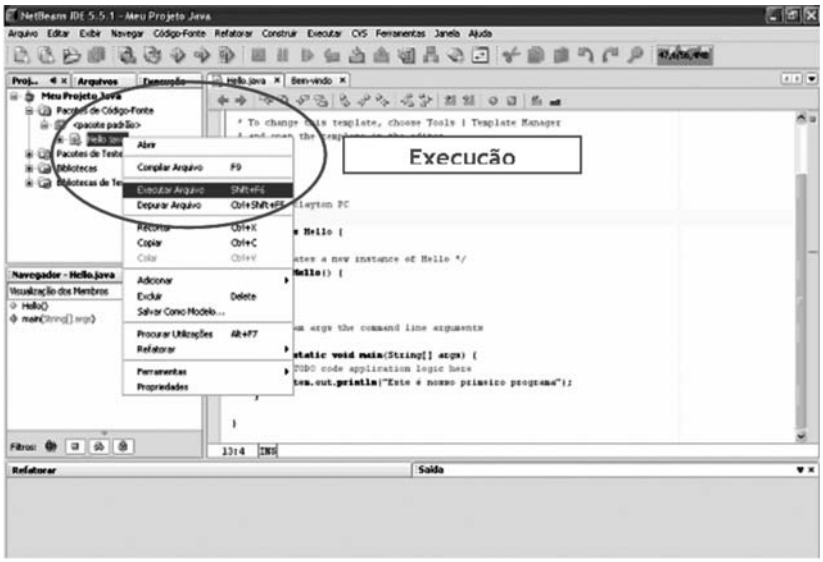


Figura 1.13: Executando uma classe no NetBeans.
Fonte: NetBeans.

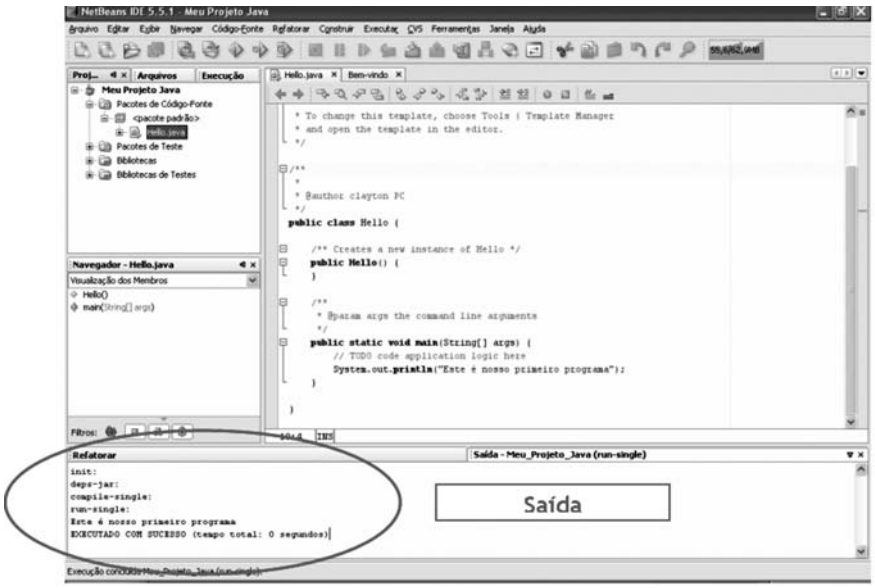
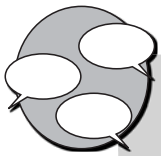


Figura 1.14: Executando uma classe no NetBeans.
Fonte: NetBeans.

Atividade prática 4

Agora vai ficar mais legal!
Siga os passos mencionados e veja seu primeiro programa em Java agora executado no NetBeans!



INFORMAÇÕES SOBRE FÓRUM

Nesta primeira semana de aula, vamos entrar no fórum para nos conhecermos; vamos falar sobre a experiência de cada um e o que esperamos do curso.

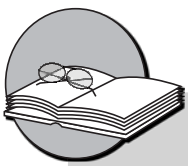
RESUMO

Vamos rever os principais conceitos vistos nesta aula.

- Histórico da tecnologia Java: o Java nasceu na empresa americana Sun Microsystems na década de 1990 com o intuito de ser utilizado em dispositivos eletrodomésticos. O projeto inicialmente não foi um sucesso, mas, com o advento da internet, a Sun viu uma oportunidade de utilizar essa tecnologia principalmente pela característica de poder ser utilizada em plataformas heterogêneas.
- Além de ser uma das mais completas, modernas e utilizadas linguagens de programação, o Java também dá suporte a um completo ambiente de desenvolvimento, por meio de suas ferramentas integradas, além de ser um ambiente de aplicação completo, com todas suas funcionalidades implementadas na Máquina Virtual Java (JVM).
- O Java possui três plataformas básicas de desenvolvimento, de acordo com o projeto do sistema a ser implementado. A plataforma básica para todos os tipos de sistemas é o JSE (Java Standard Edition), que possui as bibliotecas básicas para o desenvolvimento em Java. Para desenvolvimento corporativo, distribuído e/ou *web* é preciso utilizar a plataforma JEE (Java Enterprise Edition), que dá suporte aos sistemas desenvolvidos

para esses tipos de ambiente mais complexos. Por último, temos uma plataforma especial chamada JME (Java Micro Edition), que serve para o desenvolvimento de sistemas para dispositivos móveis e embarcados, como celulares, televisões e outros dispositivos com menor poder de processamento e memória.

- Todo programa em Java deve ser compilado pelo compilador javac, gerando assim o *bytecode*, para depois ser interpretado pela máquina virtual; esta é implementada de acordo com cada sistema operacional, garantindo a portabilidade de sistemas do *bytecode*.
- Para melhorar a produtividade do desenvolvimento e ganhar em integração e controle da programação, principalmente para sistemas complexos e grandes equipes, podemos utilizar ferramentas que auxiliam nessas tarefas; essas ferramentas são as chamadas IDEs (*integrated development environment*).



Informação sobre a próxima aula

Na próxima aula, você irá estudar um pouco mais sobre a estrutura dos programas em Java e os tipos de dados suportados pela linguagem. Até lá.

Tipos de dados e estrutura de programação em Java

AULA 2

Meta da aula

Apresentar a estrutura de uma classe Java e os tipos de dados da linguagem.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. descrever a estrutura básica de uma classe em Java;
2. criar e testar as classes que você escreveu em Java;
3. identificar os principais tipos de dados da linguagem Java, suas características e utilizações;
4. identificar os operadores utilizados na programação em Java.

Pré-requisitos

Para o correto e completo aprendizado desta aula, é preciso que você tenha um computador com Windows XP ou Windows Vista instalado, com pelo menos 512Mb de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; esteja animado e motivado para iniciar o aprendizado desta fabulosa linguagem de programação, que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

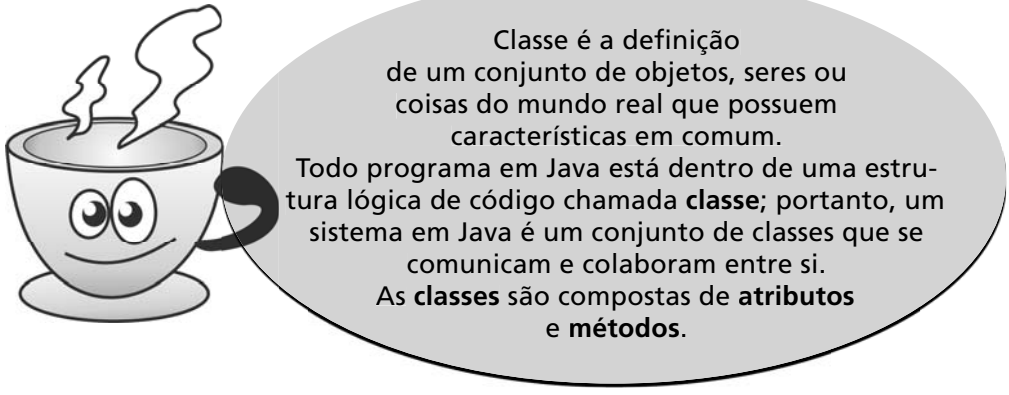
ESTRUTURA BÁSICA DE UM PROGRAMA EM JAVA

Veja como se comporta e como se compõe um programa em Java. Estudaremos isso por meio da análise de um código-fonte em Java. É como se um médico estivesse entendendo como funciona o corpo humano por dentro.

Imagine que uma montadora lhe pedisse para criar um carro. Que características você considera importantes para definir um carro?

( ) Velocidade
( ) Aceleração
( ) Cor
( ) Número de portas

Pois bem, todas essas características são comuns a todos os carros. Ao pensar num carro de forma geral, você está aprendendo um conceito bastante importante em Java: o conceito de classe.



No nível de análise e requisitos, os **atributos** são as características que os objetos do mundo real possuem, e estes são sua representação no mundo computacional.

Fazendo a mesma comparação, os **métodos** são os comportamentos ou as ações executadas por esses objetos.

Assim, com um exemplo bem simples, podemos dizer que, se estamos definindo e programando o objeto do mundo real **carro**, a classe relativa a esse objeto terá atributos como cor, modelo, ano de fabricação

etc. Além disso, a classe também terá métodos como acelerar, frear, fazer conversão etc.

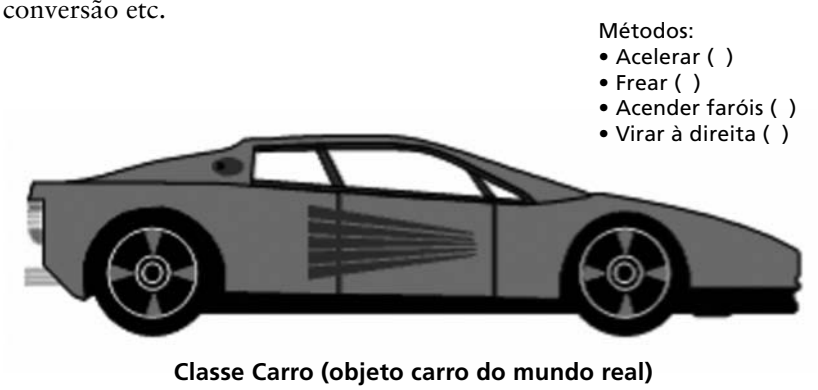


Figura 2.1: Classe Carro e seus métodos.

Por enquanto, vamos nos focar mais na estrutura de um programa em Java, pois adiante teremos aulas voltadas especificamente para o aprendizado de orientação a objeto.

Para visualizar melhor como funciona uma classe, vamos analisar a estrutura do nosso primeiro exemplo, que foi implementado e testado na aula passada:

```
1     public class Hello {
2           /**
3           *      Nosso primeiro programa em Java
4           */
5
6         public static void main(String[ ] args) {
7           // A próxima linha imprime a frase entre (“…”)
          no console da IDE
8                     System.out.println("Esse é nosso
          primeiro programa");
9         }
10
11    }
```

Vamos começar a decifrar esse programa por meio do entendimento de cada linha.

A **linha 1** define que estamos iniciando a estrutura de uma classe pela palavra reservada **class** e é seguida pelo nome da classe.

Após as instruções relativas a classe, temos o símbolo { (**abre chaves**) que define a abertura de um bloco de comando. Repare que o { de abertura tem o seu correspondente } (fecha chaves) na linha 11 indicando que aquele bloco foi fechado. Os blocos de comando são hierarquizados; assim, podemos ter blocos de comando dentro de um primeiro bloco.

Toda classe é um arquivo **Java**, e este possui a extensão **.java**; obrigatoriamente o arquivo **deve possuir o mesmo nome da classe, respeitando letras maiúsculas ou minúsculas**. Assim, nossa **classe Hello** está armazenada fisicamente num arquivo **Hello.java**.

Os comandos das **linhas 2** a **4** definem um **comentário em Java**. Um comentário serve para documentação do código que está sendo implementado, seja como histórico para os programadores ou equipe que estão fazendo o projeto, seja como subsídio para futuras manutenções e atualizações do sistema.

Existem três tipos de comentários em Java; o que foi utilizado nas linhas anteriores é o comentário de bloco que é inserido na documentação padrão de código Java automaticamente, o javadoc, e é definido pelos intervalos /** ... */, suportando quebra de linhas. Caso você não queira que o comentário de bloco seja inserido no javadoc gerado, é só suprimir o segundo símbolo * que abre o comentário. O trecho documentado fica entre os intervalos /* ... */.
O último tipo de comentário é o comentário de linha; é definido pelos caracteres //; tudo que vier após esses símbolos é ignorado pelo compilador. Só serve para uma única linha. Esse tipo de comentário foi utilizado na **linha 7** do nosso exemplo.

A **linha 5** está em branco. Isso não tem problema nenhum em Java, aliás não teremos nenhum problema com linhas ou espaços em branco, pois eles são ignorados pelo Java; utilize-os para ajudar na organização do código, mas sem exagero.

Como já falamos, um **sistema em Java** é composto de **classes que se comunicam**. As **classes são compostas de métodos e atributos**; na **linha 6**, temos nosso primeiro método, o método **main**, que é um método muito especial. Um método é geralmente uma palavra iniciada com letra minúscula. Seu nome vem seguido de ( ) (**abre e fecha parênteses**). Aliás, você pode se perguntar por que o nome de uma classe se inicia com letras maiúsculas e métodos iniciam com letras minúsculas. Isso quer dizer que haverá um erro se fizer ao contrário?

Na verdade não, isso é o que chamamos **convenções de codificação**; servem para auxiliar o programador a identificar partes de uma estrutura de código. Assim, caso um programador receba uma classe para estudar ou alterar, ele saberá que aquilo que começa com letras maiúsculas provavelmente são classes, e que letras minúsculas seguidas de ( ) geralmente caracterizam métodos.

Caso o desenvolvedor não siga essa convenção, não receberá nenhum erro de sintaxe, pois não são palavras reservadas, mas sim identificadores que o próprio programador inseriu no código para dar nome às estruturas criadas por ele. Entretanto, estará cometendo um erro grave de boa prática, deixando o código confuso para alguém que irá lê-lo futuramente. Todas as convenções de codificação do Java estão definidas num documento curto, mas muito importante, que foi preparado pela própria Sun e que serve como base para construir o manual de programação que as próprias empresas têm definido na sua metodologia (isso quando não indicam o próprio manual da Sun como padrão a ser seguido). É o *Code Conventions*; tem somente 24 páginas e vale a pena dar uma olhada nele, principalmente o programador iniciante. O *download* está disponível em:
http://java.sun.com/docs/codeconv

Atividade prática 1

Leia um pouco...

Faça o *download* e leia o *Code Conventions*. Iremos fazer alguns comentários e retirar as dúvidas sobre esse documento no fórum desta semana. Só para adiantar, o *Code Conventions* define a forma e o padrão de como serão codificados seus programas, ou seja, como são os comentários quando eu escrevo com letras maiúsculas, quando eu utilizo minúsculas, quando eu escrevo junto palavras compostas, quando eu utilizo hífen para separá-las etc.

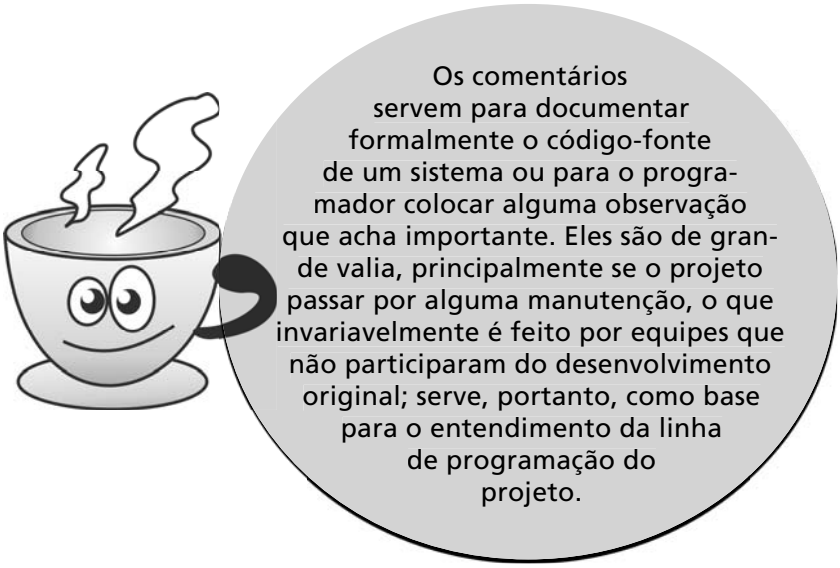
Continuando a explicar as estruturas das classes: um método deve possuir ( ) após seu nome, mas no nosso método **main** vimos alguma coisa dentro deles; esses são os argumentos ou parâmetros que são passados para o método e não são obrigatórios.

Voltando a explicar o método **main**: por que ele é um método especial? Um sistema possui muitas classes, mas uma delas deve ser o ponto inicial pelo qual o sistema é inicializado; essa classe é aquela que possui o método **main**. A JVM começa a executar o sistema por esse método. Como, no nosso caso, nosso sistema possui uma única classe, para ser uma aplicação completa e para que possa ser executada por si só, essa classe deve possuir o método **main**, e toda execução do programa deve ocorrer a partir dele.

Na **linha 8**, temos um novo método que faz parte das bibliotecas do Java, o método ***System.out.println***("..."), que escreve na saída-padrão do computador o conteúdo entre ("...").

Por fim, não podemos nos esquecer de que bloco de comando que é aberto deve ser fechado, e os blocos abertos nas **linhas 1** (abertura da classe) e **6** (abertura do método ***main***) são fechados nas **linhas 9** e **11**.

COMENTÁRIOS EM JAVA



Todo comentário é ignorado pelo compilador, não interferindo na execução do programa.

O Java dá suporte a três tipos de comentários:

1. Comentário de linha: nele, toda sentença escrita após o // é considerada comentário, mas somente essa linha. Exemplo:

```
// Comentário de uma única linha
```

2. Comentário de bloco: neste, todo texto definido entre /* e */ é considerado comentário pelo compilador; suporta quantas quebras de linhas existirem. Por questões visuais, os programadores costumam colocar * também no início de cada linha que não está nos símbolos que definem o comentário de bloco. Exemplo:

```
/* Comentário
de bloco
aceita mais de
uma linha */
```

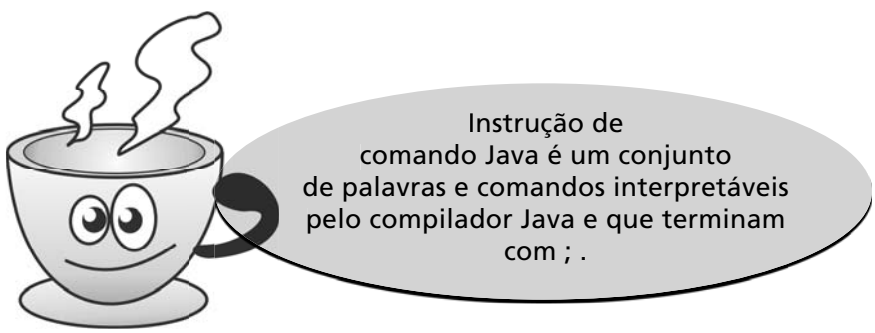
ou

```
/* Comentário
* de bloco
* aceita mais de
* uma linha
*/
```

3. Comentário de Javadocs: o último tipo de comentário é, na verdade, uma modificação sutil no comentário de bloco; nele, o primeiro símbolo passa a ser o /** ao invés do /*. Isso faz com que o comentário seja inserido na documentação padrão de código gerada pelo aplicativo javadocs, dentro de seu HTML. Além disso, o comentário de javadocs contém algumas tags (etiquetas indicativas) no formato de *annotations* (iniciando com @) que são predefinidas, dizendo o significado daquela linha de comentário. Exemplo:

```
/** Comentário inserido
nos javadocs
@author Clayton Chagas
@version 1.0
* /
```

INSTRUÇÕES E BLOCOS DE COMANDO



Um bloco de comando é formado por uma ou mais instruções entre chaves; pode ter espaços em branco e quebras de linhas.

Por uma questão de organização de código, é aconselhável a correta identação dos níveis de blocos de comando e suas instruções internas, como você vê no exemplo a seguir:

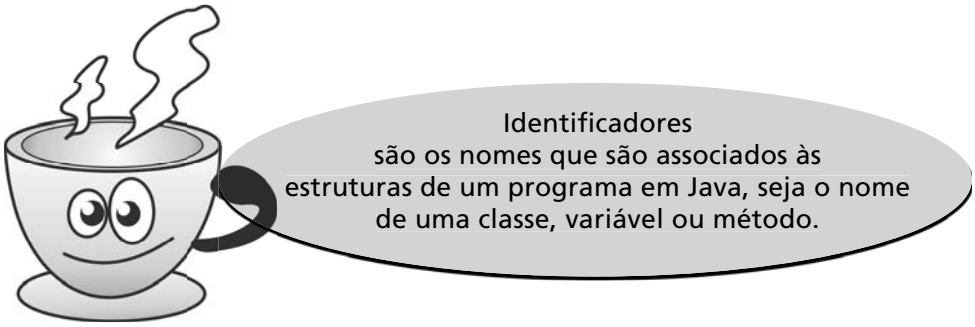
```
public static void main (String[ ] args) {
        System.out.println("Teste1");
        System.out.println("Teste2");
}
```

Início do bloco

Instruções de Comando

Fim do bloco

IDENTIFICADORES



Vale lembrar que o Java é **case-sensitive**, ou seja, diferencia caracteres maiúsculos de minúsculos. Assim, um recurso do código cujo identificador é **Teste** é diferente de outro recurso cujo identificador é **teste**. Existem inclusive regras de boas práticas a serem seguidas nas definições de letras maiúsculas e minúsculas dos identificadores de acordo com o *Code Conventions* da Sun (http://java.sun.com/docs/codeconv).

Basicamente, classes se iniciam com letras maiúsculas e o restante das letras minúsculas; se o identificador de classe for composto de mais de uma palavra, elas virão aglutinadas, cada uma delas também se inicia com maiúsculas. Como em:

```
ExemploDeNomeDeClasse.
```

Para método e variável (ou atributo) a regra é parecida, porém a primeira palavra se inicia com letra minúscula. Exemplo:

```
exemploDeNomeDeMetodo( )
exemploDeNomeDeVariavel
```

Existem algumas outras regras, por isso recomendamos a leitura do *Code Conventions*.

Quanto ao universo que pode ser utilizado, os identificadores podem começar com qualquer letra, *underscore* (_) ou cifrão ($). Os caracteres subsequentes podem utilizar também números de 0 a 9.

PALAVRAS-CHAVE

Algumas palavras não podem ser utilizadas como identificadores em Java; normalmente representam algum comando a ser utilizado numa instrução. Ao longo do aprendizado, você irá conhecer a utilização da maioria dessas palavras. Por enquanto vamos apenas enumerá-las; tente memorizá-las, mas saiba que a maioria das IDEs irão avisar caso você esqueça e tente utilizar uma dessas palavras-chave:

ABSTRACT	CONTINUE	FOR	NEW	SWITCH
ASSERT	DEFAULT	GOTO	PACKAGE	SYNCRONIZED
BOOLEAN	DO	IF	PRIVATE	THIS
BREAK	DOUBLE	IMPLEMENTS	PROTECTED	THROW
BYTE	ELSE	IMPORT	PUBLIC	THROWS
CASE	ENUM	INSTANCEOF	RETURN	TRANSIENT
CATCH	EXTENDS	INT	SHORT	TRY
CHAR	FINAL	INTERFACE	STATIC	VOID
CLASS	FINALLY	LONG	STRICTFP	VOLATILE
CONST	FLOAT	NATIVE	SUPER	WHILE

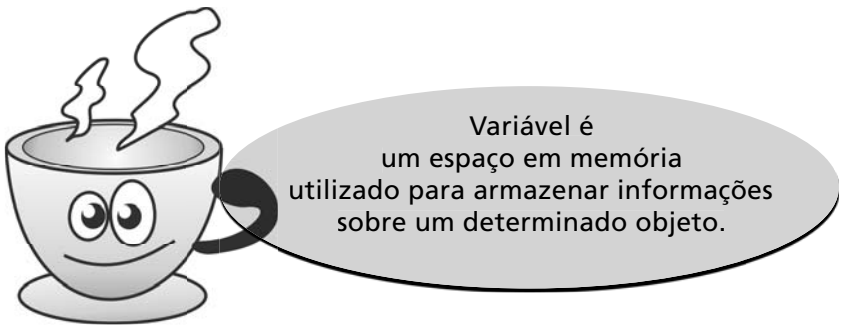
Além dessas palavras-chave, **true**, **false** e **null** são palavras reservadas; portanto, não podem ser utilizadas como identificadores.

TIPOS DE DADOS

O Java possui quatro famílias básicas de tipos de dados:

- **Boolean**: que serve para trabalharmos com valores lógicos *true* ou *false* e seu tipo primitivo é o **boolean**.
- **Character**: são caracteres representativos da tabela **Unicode**, que é uma tabela complementar à tabela **ASCII**. Esta armazena somente 8 bits enquanto a Unicode armazena um símbolo de 16 bits, o que possibilita armazenar muito mais caracteres especiais, além de uma quantidade muito maior de idiomas. Para representar caracteres Unicode, sempre utilizamos aspas simples ('...') ao invés de aspas normais, que são duplas ("...") e utilizadas para cadeias de caracteres (*string* em Java). Seu tipo primitivo é o **char**.
- **Integer**: esse tipo de dados serve para representações numéricas de bases, como base decimal, hexadecimal ou octal. O tipo primitivo padrão da família **Integer** é o **int** e define um número de base decimal de 32 bits com sinal. Existe o tipo **long**, que estende esse decimal para 64 bits com sinal; para tanto, deve-se colocar um **L** após o número, além de definir o tipo como **long**. Caso você deseje definir um hexadecimal ao invés de um decimal, é só colocar **0X** antes do símbolo; para octal, deve-se colocar **0**. Existem também os tipos primitivos **byte** e **short**.
- **Float-Point:** esses dados representam os tipos numéricos com ponto flutuante. Existem dois tipos com ponto flutuante em Java: o tipo **double**, que é o tipo padrão e tem precisão de 64 bits, e o tipo **float**, que tem precisão de 32 bits e necessita da colocação do indicativo **f** após o número.

VARIÁVEIS



Uma variável contém um identificador e um tipo, os dois de acordo com as definições que você já estudou. Isso é o que chamamos de *declaração de uma variável*.

Além da declaração, uma variável também pode ser inicializada por meio do operador de atribuição "=".

As declarações de variáveis do mesmo tipo podem ser feitas todas na mesma linha, mas essa é considerada uma má prática, pois deixa o código ilegível, confuso e desorganizado.

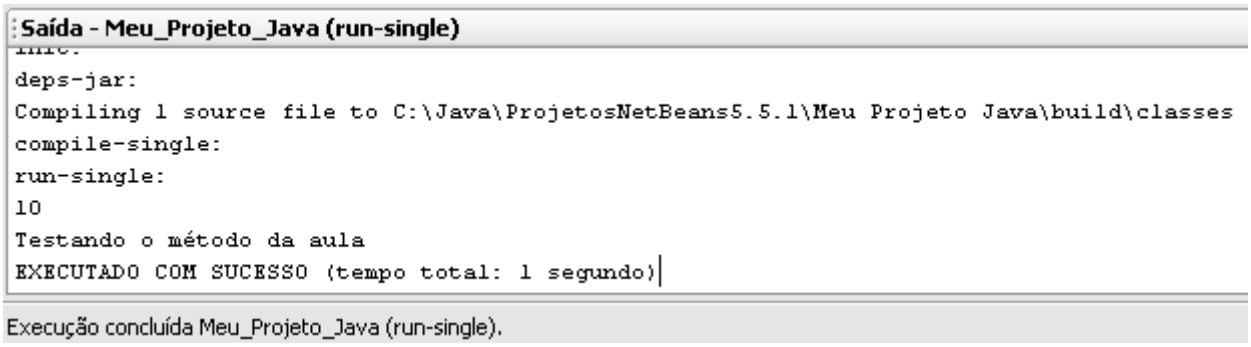
Veja algumas operações com variáveis no exemplo a seguir:

```
public class TesteVariaveis {
    public static void main(String[] args) {
        //declaração de duas variáveis int na mesma linha: má prática
        int var1, var2;
        //declaração de uma variável int na mesma linha
        int var3;
        //inicialização de var3 com o valor 5, utilizando o operador
        // "=" de atribuição
        var3 = 5;
        //declaração e inicialização de uma variável numa única linha
        int var4 = 10;
    }
}
```

Para exibir o valor de uma variável, utilizamos o método *println( ) de System.out* passando a variável como argumento ou, caso queira, a própria String entre aspas ("...").

```
public class TesteVariaveis {
    public static void main(String[] args) {
        int var2 = 10;
        System.out.println(var2);
        System.out.println("Testando o método da aula");
    }
}
```

A saída da classe anterior no console do NetBeans, quando executada, será o valor da variável (no caso 10) e o conteúdo da String.



Os conteúdos de uma variável podem ser acessados por valor ou por referência, dependendo da sua origem.

Caso uma **variável** seja de **tipo primitivo**, seu **conteúdo** ou **estado** (já que variável é um espaço em memória) será **acessado por valor**, ou seja, seu dado está armazenado no espaço de memória destinado a variável.

Pode parecer estranho, mas **quando a variável é um objeto** (um tipo originário de uma classe), ela possui um comportamento diferente (existe um motivo muito importante para que se comporte assim; você estudará futuramente), o dado ou **os dados são acessados por referência**, ou seja, é armazenado o endereço de memória onde o dado é armazenado, uma referência para ele.

Suponha que, no nosso último exemplo, ao invés de termos passado diretamente a String como argumento para o método *println*,

tivéssemos criado uma variável do tipo String e a tivéssemos passado ao método:

```
public class TesteVariaveis {
    public static void main(String[] args) {
        int var2 = 10;
        System.out.println(var2);
        String str = "Testando o método da aula";
        //System.out.println("Testando o método da aula");
        System.out.println(str);
    }
}
```

O resultado seria o mesmo, mas conseguiríamos visualizar melhor o acesso ao dado por valor e por referência na memória, de acordo com o que foi explicado e como está ilustrado na figura a seguir:

ENDEREÇO DE MEMÓRIA	NOME DA VARIÁVEL	DADO ACESSADO NA MEMÓRIA
1001	var2	10
...		...
1401	str	Endereço (1701)
...		...
...		...
1701		"Testando o método da aula"

OPERADORES

Todos os tipos comuns de operadores existentes nas linguagens de programação estruturadas estão presentes em Java. Veja quais são eles, suas principais características e formas de sua utilização.

Operadores aritméticos

Operadores aritméticos são aqueles utilizados para efetuar operações matemáticas

OPERADOR	SINTAXE	FUNÇÃO
*	a * b	Multiplica a por b
/	a / b	Divide a por b
%	a % b	Resto da divisão de a por b
-	a - b	Subtrai a de b
+	a + b	Soma a e b

Esses operadores são aplicados a tipos numéricos, que possuem tamanhos e características diferentes, conforme seu tipo. Assim, caso você opere com duas variáveis numéricas de tipos diferentes, por exemplo, um **int** multiplicado por um **double**, o resultado será sempre do maior tipo; no exemplo anterior, um **double**.

Atividade prática 2

Crie um programa e declare nele 3 variáveis numéricas **int**, inicializando-as com algum valor e obtenha a soma dessas variáveis e a média delas, imprimindo esses resultados na tela.

Operadores de Incremento e Decremento

Operadores de Incremento e Decremento têm a função de aumentar ou diminuir em uma unidade o valor de uma variável.

Deve ser dada atenção especial à posição do operador, pois, dependendo dela, pode-se efetuar a operação desejada antes ou depois de uma avaliação numa determinada expressão ou condição.

OPERADOR	SINTAXE	FUNÇÃO
++	i ++	Incrementa a variável em 1; avalia a expressão antes do incremento
++	++ i	Incrementa a variável em 1; o incremento está antes de avaliar a expressão
--	i --	Decrementa a variável em 1; avalia a expressão antes do decremento
--	-- i	Decrementa a variável em 1, antes de avaliar a expressão

Operadores Relacionais

Operadores Relacionais comparam duas variáveis e determinam seu relacionamento com um **resultado lógico**, ou seja, **true** (para verdadeiro) ou **false** (para falso).

OPERADOR	SINTAXE	FUNÇÃO
>	a > b	Avalia se a variável a é maior que a variável b (e retorna **true** ou **false**, dependendo dos valores de a e b)
>=	a >= b	Avalia se a variável a é maior ou igual à variável b (e retorna **true** ou **false**, dependendo dos valores de a e b)
<	a < b	Avalia se a variável a é menor que a variável b (e retorna **true** ou **false**, dependendo dos valores de a e b)
<=	a <= b	Avalia se a variável a é menor ou igual à variável b (e retorna **true** ou **false**, dependendo dos valores de a e b)
= =	a = = b	Avalia se a variável a é igual à variável b (e retorna **true** ou **false**, dependendo dos valores de a e b)
! =	a ! = b	Avalia se a variável a é diferente da variável b (e retorna **true** ou **false**, dependendo dos valores de a e b)

Atividade prática 3

Crie um programa que calcule a área de um retângulo, declare e inicialize duas variáveis para testá-lo, imprimindo esses resultados na tela.

Operadores Lógicos

Operadores Lógicos avaliam um ou mais operandos lógicos que geram um único valor lógico (**true** ou **false**) como resultado final da expressão avaliada.

Existem seis Operadores Lógicos:

- &&: and (operador e) lógico;
- &: and (operador e) binário;
- ||: or (operador ou) lógico;
- |: or (operador ou) binário;
- ^: ou exclusivo binário;
- !: operador de negação.

Os Operadores Lógicos podem se aplicar a expressões, variáveis ou constantes.

Vamos ver as tabelas-verdade dos Operadores Lógicos:

• **&& (and lógico)** e **& (and binário):**

OPERANDO A	OPERANDO B	RESULTADO
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	falso
falso	verdadeiro	falso
falso	falso	falso

• **|| (or lógico)** e **| (or binário):**

OPERANDO A	OPERANDO B	RESULTADO
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
falso	verdadeiro	verdadeiro
falso	falso	falso

• **^ (ou exclusivo binário):**

OPERANDO A	OPERANDO B	RESULTADO
verdadeiro	verdadeiro	falso
verdadeiro	falso	verdadeiro
falso	verdadeiro	verdadeiro
falso	falso	falso

• **! (negação):**

OPERANDO A	RESULTADO
verdadeiro	falso
falso	verdadeiro

Precedência de operadores

A precedência de operadores serve para indicar a ordem na qual o compilador interpretará os diferentes tipos de operadores, para que ele sempre tenha como saída um resultado consistente com a regra que quer ser aplicada pelo programador, evitando ambiguidades e inconsistências de operações.

Na dúvida, evite fazer operações complexas de uma só vez. Caso seja necessário, prefira utilizar parênteses a confiar que irá decorar e aplicar corretamente essas precedências.

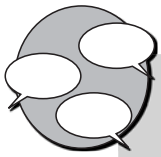
ORDEM	OPERADOR
1	**( )** parênteses
2	**++** pós-incremento e **--** pós-decremento
3	**++** pré-incremento e **--** pré-decremento
4	**!** negação lógica
5	***** multiplicação e **/** divisão
6	**%** resto da divisão (mod)
7	**+** soma e **-** subtração
8	**<** menor que, **<=** menor ou igual, **>** maior que, **>=** maior ou igual
9	**==** igual e **!=** diferente
10	**&** and binário
11	**|** or binário
12	**^** ou exclusivo binário
13	**&&** and lógico
14	**||** or lógico
15	**=** operador de atribuição

Atividade prática **4**

Crie um programa que elabore o orçamento de uma construtora para o cálculo do valor total de construção de uma piscina. O valor total será dado pela cubagem da piscina (metros cúbicos) multiplicada por R$100,00, que é o preço do metro cúbico construído de piscina. Faça a simulação dos cálculos no programa para ver se está tudo funcionando corretamente.

Observação final:

• As atividades devem ser feitas diretamente na IDE NetBeans, mas um rascunho pode auxiliá-lo na construção da lógica a ser implementada.



INFORMAÇÕES SOBRE FÓRUM

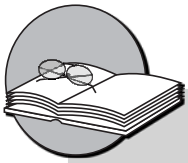
É importante e útil compartilhar as experiências e dificuldades de cada um, pois a dúvida de um pode ser a dúvida de todos. Entre no fórum da semana e participe da discussão das dúvidas e das soluções das atividades desta segunda aula.
Além disso, vamos ter também duas sessões de *chat*. Entre no fórum para ver a divisão dos grupos e o horário das sessões.

RESUMO

Reveja os principais conceitos vistos nesta aula:

- Todo programa em Java é composto por uma estrutura de código chamada **classe**.
- Uma **classe** é composta por **atributos** e **métodos**.
- Assim, um sistema em Java é composto por diversas classes que se comunicam e colaboram entre si por meio da troca de mensagens, que na prática são as chamadas aos métodos.
- Toda vez que você for digitar suas classes, siga as orientações e convenções de codificações que estão no documento oficial da Sun chamado *Code Conventions*.
- Em Java, existem três tipos de comentários: **comentário de linha (//...), comentário de bloco (/*...*/)** e **comentário para inclusão de observações nos javadocs (/**...*/).**
- Instrução de comando Java é um conjunto de palavras e comandos interpretáveis pelo compilador Java e que terminam com ; .
- Um bloco de comando é formado por uma ou mais instruções entre chaves; aceita espaços em branco e quebras de linhas.
- Identificadores são os nomes que são associados às estruturas de um programa em Java, seja o nome de uma classe, variável ou método.
- Palavras-chave são palavras que não podem ser utilizadas como identificadores ao escrever uma classe, pois o compilador já utiliza essa palavra para alguma função.
- O Java possui quatro famílias base de tipos de dados: Boolean, Character, Integer e Float-Point.

- Variável é um espaço em memória utilizado para armazenar informações sobre um determinado objeto.
- O Java, como toda linguagem de programação, possui alguns tipos de operadores para executar operações sobre os dados e atributos dos objetos. Dentre eles podemos citar: Operadores de Incremento e Decremento; Operadores Relacionais e Operadores Lógicos.



Informações sobre a próxima aula

Na próxima aula, iremos estudar um pouco mais as estruturas de controle no Java, como realizar *loopings* e como criar programas mais interativos e com melhores recursos gráficos para fazer inserção de dados. Até lá...

AULA 3

Estruturas de controle e entrada de dados em Java

Meta da aula

Apresentar as estruturas de controle do Java e sua utilização.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. citar os tipos de estruturas de controle utilizadas no Java;
2. criar estruturas de controle nos seus programas em Java;
3. diferenciar a utilização das diversas estruturas de controle;
4. criar programas que recebam entradas de dados em tempo de execução de forma gráfica e interativa.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você: tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

INTRODUÇÃO

Até agora, executamos programas de forma unicamente sequencial, ou seja, tem um ponto de entrada e a partir dele ocorre a execução dos comandos de acordo com os parâmetros passados, linha a linha.

Nesta aula, você irá aprender como executar desvios ou ações iteradas (que ocorrem mais de uma vez) por meio de conjuntos de comandos dentro de estruturas chamadas **estruturas de controle**.

Essas estruturas estão divididas em:

- **estruturas de decisão**: que permitem a seleção de partes do código para execução, os chamados desvios (**if** e **switch**);
- **estruturas de repetição**: que permitem a repetição da execução de partes específicas do código, os chamados *loopings* (**while**, **do-while** e **for**);
- **interrupções**: estruturas que param e redirecionam o fluxo do programa (**break**, **continue** e **return**).

Vamos então estudar cada uma dessas estruturas para ver como elas podem ajudar na programação de classes Java.

ESTRUTURAS DE DECISÃO

Estruturas de decisão são comandos do Java que permitem que blocos sejam executados ou não, de acordo com algum critério.

Existem quatro formas de estruturas de decisão em Java:

- if
- if-else
- if-else-if
- switch

Declaração if

A declaração **if** dentro de um bloco de comando determina que as instruções daquele bloco só serão executadas se a expressão lógica associada ao **if** for verdadeira.

A declaração **if** possui a seguinte sintaxe:

```
if (expressão_lógica)
instrução;
```

ou

```
if (expressão_lógica)  {
       instrução1;
       instrução2;
       instrução3;
       ...
}
```

Lembre-se de que expressão lógica é alguma estrutura em Java (expressão ou variável) que retorna um **valor booleano** (**true** ou **false**).

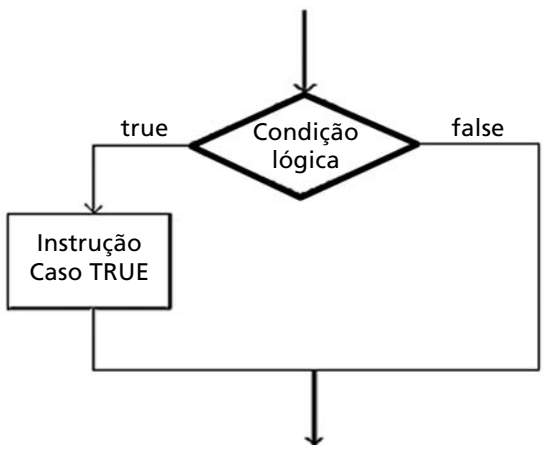


Figura 3.1: Fluxograma da declaração **if**.

O trecho de código a seguir exemplifica uma implementação da declaração **if**:

```
double nota = 6.0;
if (nota >= 5.0) {
    System.out.println("Parabéns, você passou na disciplina");
}
```

No exemplo anterior, o atributo **nota** tem o valor **6.0**. Após essa definição, temos um IF que testa logicamente o valor de nota, perguntado no teste **se** (**if**) **nota é maior** ou **igual a 5.0**. Como no nosso caso essa pergunta tem uma resposta positiva, pois realmente nota tem um valor maior ou igual a 5.0, já que vale 6.0, o trecho entre parênteses retorna um resultado verdadeiro (**true**), e quando isso ocorre, o bloco de comando dentro do teste é executado, então o programa executa a linha **System.out.println**(**"Parabéns, você passou na disciplina"**);

Caso a pergunta do teste tenha uma resposta negativa, ele retorna um resultado falso (**false**) e o bloco de comando dentro do teste não é executado, passando a execução do programa para o próximo comando após o bloco de comando do teste.

IF

A palavra **if** utilizada na estrutura de decisão é uma palavra em inglês e significa **se**, tendo inclusive o mesmo significado dentro da estrutura do programa. Se você for um programador iniciante, a leitura como se fosse português pode ajudar no entendimento da lógica representada. Por exemplo, no trecho de código acima podemos fazer a seguinte leitura: "se a nota for maior ou igual a 5.0..." e então a estrutura lógica é interpretada de acordo com o valor da variável utilizada para avaliar a expressão, no caso *nota*, que tem o valor 6.0 (definido na linha anterior); assim, a expressão lógica retorna *true* e então entra no bloco interior da estrutura executando-o – no caso, imprimindo a mensagem passada como parâmetro do método *println*.

Declaração if-else

A declaração **if-else** em dois blocos de comando associados determina que as instruções dentro do bloco ao qual o **if** pertence serão executadas se a expressão lógica associada ao **if** for verdadeira; as instruções dentro do bloco ao qual o **else** pertence serão executadas se a expressão for falsa.

A declaração **if-else** possui a seguinte sintaxe:

```
if (expressão_lógica)
instrução_caso_expressão_true;
else
       instrução_caso_expressão_false;
```

ou

```
if (expressão_lógica)  {
instrução1_caso_expressão_true;
instrução2_caso_expressão_true;
...
}  else  {
instrução1_caso_expressão_false;
instrução2_caso_expressão_ false;
       ...
}
```

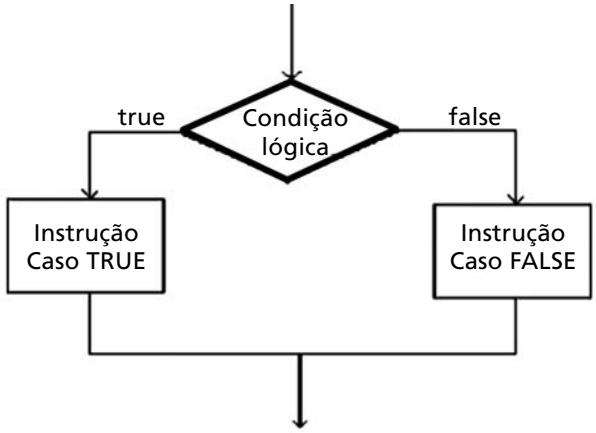


Figura 3.2: Fluxograma da declaração **if-else**.

O trecho de código a seguir exemplifica uma implementação da declaração **if-else**:

```
double nota = 6.0;
if (nota >= 5.0) {
    System.out.println("Parabéns, você passou na disciplina");
} else {
    System.out.println("Você foi reprovado na discipina");
}
```

No exemplo anterior, o atributo **nota** tem o valor **6.0**. Após essa definição, temos um **if** que testa logicamente o valor de **nota**, perguntado no teste **se (if) nota é maior** ou **igual a 5.0**. Como no nosso caso essa pergunta tem uma resposta positiva, pois realmente nota tem um valor maior ou igual a 5.0, já que vale 6.0, o trecho entre parênteses retorna um resultado verdadeiro (**true**) e quando isso ocorre, o bloco de comando dentro do teste é executado, então o programa executa a linha **System. out.println("Parabéns, você passou na disciplina")**.

Ainda nesse exemplo, suponha que o valor de **nota** não tivesse sido inicializado com **6.0**, mas com **4.0**; então, o teste do **if** retornaria um valor ***false*** já que sua resposta ao teste seria negativa. Assim, o bloco de comando do **IF** não seria executado e seria executado o bloco de comando do **ELSE**, que significa *senão*. A leitura pode ser feita da seguinte forma: **"Se (if) tal assertiva for verdade execute tal função, senão (else, do contrário) execute esta outra..."** A JVM irá imprimir a frase **"Você foi reprovado na disciplina"**, por meio do método **System.out.println**.

IF-ELSE

As palavras **if-else** utilizadas na estrutura de decisão são palavras em inglês e significam **se** e **senão** em português, tendo inclusive o mesmo significado dentro da estrutura do programa. Se você for um programador iniciante, a leitura como se fosse português pode ajudar no entendimento da lógica representada. Por exemplo, no trecho de código anterior podemos fazer a seguinte leitura: **SE** A NOTA FOR MAIOR OU IGUAL A 5.0... **SENÃO**... Nesse caso, significa que **se** a nota for maior ou igual a 5.0, a expressão avaliada retornará *true* e executará o bloco de comando interior da estrutura do **if**; caso contrário, executará o bloco de comando interior da estrutura do **else**.

Declaração if-else-if

Com as estruturas **if** e **if-else** conseguiríamos apenas tratar um fluxo binário, no máximo; para fluxos mais complexos, utilizamos a estrutura **if-else-if**, na qual conseguimos tratar mais de dois fluxos possíveis, como no exemplo a seguir.

A declaração **if-else-if** possui a seguinte sintaxe:

```
if (expressão_lógica1)
      instrução_caso_expressão1_true;
else if (expressão_lógica2)
      instrução_caso_expressão2_true;
else
      instrução_caso_expressão2_false;
```

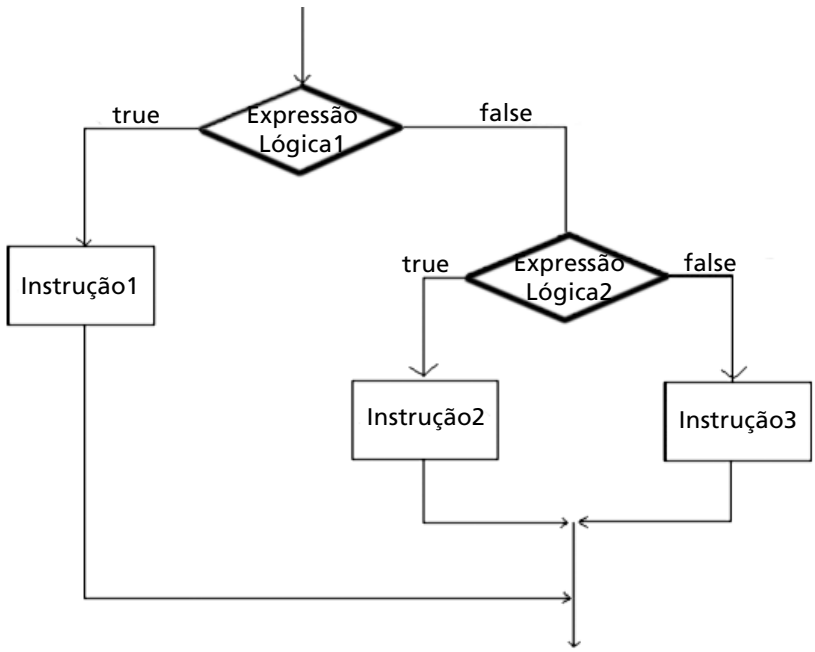


Figura 3.3: Fluxograma da declaração **if-else-if.**

O trecho de código a seguir exemplifica uma implementação da declaração **if-else-if**:

```
double nota = 6.0;
if (nota >= 9.0) {
    System.out.println("Excelente, você passou com louvor!");
} else if (nota >= 7.0 & nota < 9.0){
    System.out.println("Parabéns, você foi aprovado com boas notas
} else if (nota >= 5.0 & nota < 7.0){
    System.out.println("Você foi aprovado, não deixe de estudar");
} else {
    System.out.println("Sinto muito, você foi reprovado");
}
```

No exemplo anterior, o atributo **nota** tem o valor **6.0**. Após essa definição, temos um **if** que testa logicamente o valor de **nota**, perguntado no teste **se** (**if**) **nota é maior ou igual a 9.0**. Neste caso, não é, então a execução sai do bloco de comando do **if** e vai para o próximo **else if**, cujo teste também é falso, já que o valor de nota, que é **6.0**, não está **entre 7.0** e **9.0**. Então, no próximo **else if**, o teste pergunta se **nota** está **entre 5.0** e **7.0**, o que é verdade. Com isso, o bloco de comando dentro desse teste é executado, ou seja, imprime **"Você foi aprovado, não deixe de estudar"**, por meio do método **System.out.println**... e a linha de execução passa para o comando logo após os IF-ELSE IF aninhados.

IF-ELSE IF

As palavras **if-else if** utilizadas na estrutura de decisão são palavras em inglês e significam **se, senão se** em português, tendo inclusive o mesmo significado dentro da estrutura do programa. Se você for um programador iniciante, a leitura como se fosse português pode ajudar no entendimento da lógica representada. Nesse caso, temos o que chamamos de blocos de comando aninhados (mais de dois). Tente interpretar, assim como fizemos nos exemplos anteriores.

Declaração switch

Se você trabalhar com fluxos múltiplos e quiser escolhê-los a partir de um valor numérico (obrigatoriamente um **int**), a estrutura de decisão a ser utilizada é o **switch**, que, de acordo com o valor numérico da variável, define qual bloco será executado.

A declaração **switch** possui a seguinte sintaxe:

```
switch (variável_int) {
      case valor1:
            instrução1;
            instrução2;
            ...
            break;
      case valor2:
```

```
            instrução1;
            instrução2;
            ...
            break;
      default:
            instrução1;
            instrução2;
            ...
            break;
}
```

A declaração **switch** ainda possui um bloco definido como **default,** que é executado caso nenhum dos valores do **case** corresponda ao valor da variável inteira.

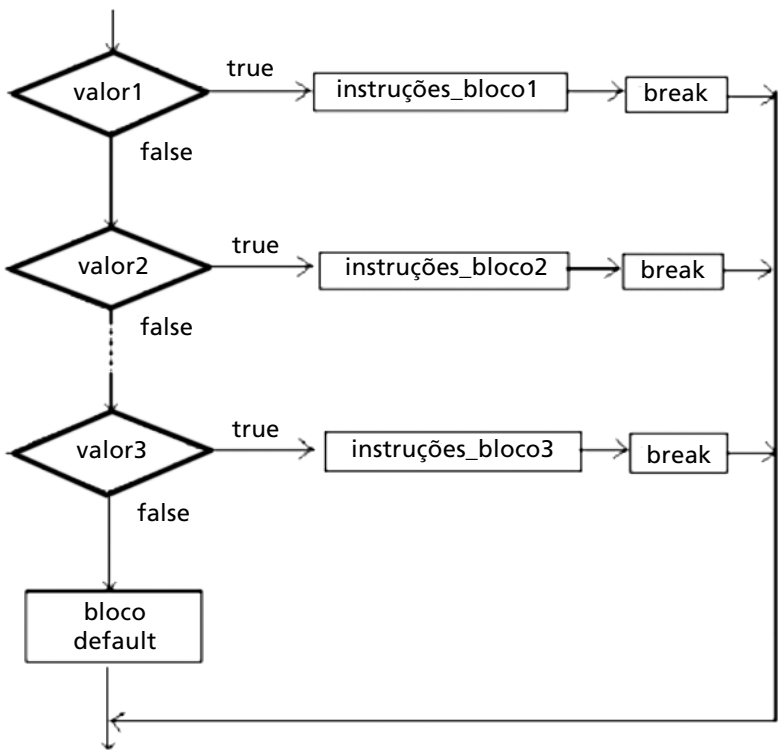


Figura 3.4: Fluxograma da declaração **switch.**

O trecho de código a seguir exemplifica uma implementação da declaração **switch**:

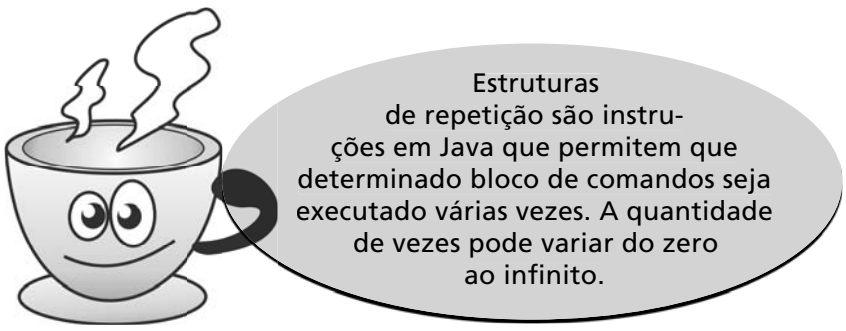
```
int andar = 3;
switch(andar){
     case 1:
          System.out.println("Você escolheu o 1° andar!");
          break;
     case 2:
          System.out.println("Você escolheu o 2° andar!");
          break;
     case 3:
          System.out.println("Você escolheu o 3° andar!");
          break;
     default:
          System.out.println("ESCOLHA UM ANDAR VÁLIDO!");
}
```

No código anterior, definimos uma variável do tipo **int** e definimos o valor dela como **3**.

A próxima linha passa o argumento para **switch**, que deve ser um valor inteiro, no caso passamos **andar**, cujo valor é **3**.

Assim, a cada linha **case** após o **switch**, é testado se há algum valor compatível com o argumento passado. No nosso exemplo, o **terceiro case** tem o valor **3**, o mesmo de **andar**, então é executado o bloco de comando dentro do **case e**, após isso, a execução sai da estrutura **switch/case**, passando a execução para o próximo comando no código.

ESTRUTURAS DE REPETIÇÃO



Declaração while

O **while** executa um bloco de comandos repetidamente, enquanto a condição lógica imposta como teste for verdadeira (expressão lógica igual a *true*).

A declaração **while** possui a seguinte sintaxe:

```
while (expressão_lógica) {
       instrução1;
       instrução2;
       ...
}
```

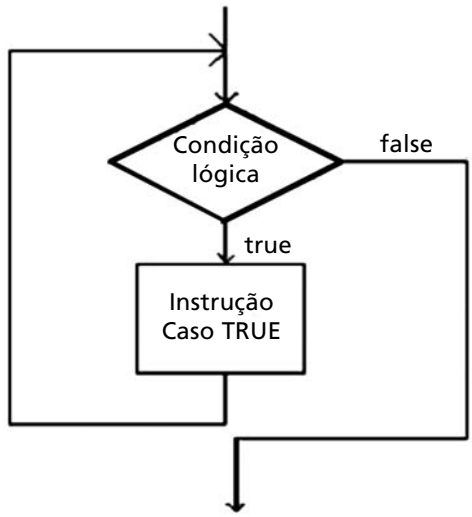


Figura 3.5: Fluxograma da declaração **while**.

O trecho de código a seguir exemplifica uma implementação da declaração **while**:

```
int i = 10;
while (i > 0){
    System.out.println(i);
    i--;
}
```

No nosso exemplo, temos uma variável **i** inteira com valor **10**. A execução e leitura do **WHILE** diz que **enquanto** o valor de **i for maior do que 0**, ele continua executando as operações do bloco de comando no interior do **while**. Nesse caso, a operação executada é o **decremento unitário do valor de i (i--)**. Assim, na primeira execução o teste lógico **pergunta se i é maior do que 0**, o que é verdadeiro, então ele entra na execução do **while** e subtrai 1 do valor atual de **i**, que passa a valer **9**, e o teste é feito novamente mas com o novo valor de **i**, e assim sucessivamente, até o teste retornar **false**, ou seja, quando **i** for menor ou igual a zero, quando sai a execução da estrutura **while**.

WHILE

A palavra **while** utilizada na estrutura de repetição é uma palavra em inglês e significa **enquanto**, em português; tem inclusive o mesmo significado dentro da estrutura do programa. Se você for um programador iniciante, a leitura como se fosse português pode ajudar no entendimento da lógica representada. Se estiver com dificuldades, tente interpretar em português assim como fizemos nos exemplos anteriores.

Declaração do-while

O **do-while** executa um bloco de comandos repetidamente enquanto a condição lógica imposta como teste for verdadeira (expressão lógica igual a *true*); a diferença entre ele e o **while** é que o teste da expressão lógica vem depois do bloco; assim, a instrução será executada pelo menos uma vez e somente depois será avaliada.

A declaração **do-while** possui a seguinte sintaxe:

```
do {
       instrução1;
       instrução2;
       ...
} while (expressão_lógica);
```

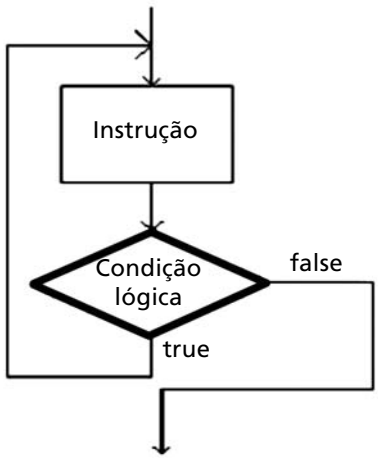


Figura 3.6: Fluxograma da declaração **do-while**.

O trecho de código a seguir exemplifica uma implementação da declaração **do-while**:

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 10);
```

Como vimos, o nosso exemplo executará a mesma operação de um bloco **while**, mas com a diferença de que ele executa o bloco de comando do laço antes para depois fazer o teste lógico; com isso, ele será executado pelo menos uma vez. Temos uma variável **i** inteira que foi inicializada com **0**, após isso o bloco **do-while**, então, o bloco de comando no seu interior é executado a primeira vez, imprimindo o valor de **i** e na linha seguinte fazendo com o que o valor inicial do **i seja incrementado** (**acrescido em 1**) por meio da operação **i++**. Assim, **i**, que valia 0, passa a valer 1 na primeira passagem. Após isso, ele faz o teste no **while**, perguntando se o valor atual de **i** é menor que 10, caso seja, ele executa o bloco mais uma vez. No caso é, então **i** é novamente incrementado até o teste lógico do **while** retornar **false**, o que ocorre quando **i** for igual a **10**, saindo do laço **DO-WHILE** e passando para o comando seguinte.

DO-WHILE

As palavras **do-while** utilizadas na estrutura de repetição são palavras em inglês e significam **faça-enquanto** em português; têm inclusive o mesmo significado dentro da estrutura do programa. Se você for um programador iniciante, a leitura como se fosse português pode ajudar no entendimento da lógica representada. Se estiver com dificuldade, tente interpretar em português, assim como fizemos nos exemplos ante-

Declaração for

O **for** executa um bloco de comandos repetidamente da mesma forma que o **while** ou o **do-while**, ou seja, o programador pode controlar a quantidade de execuções de acordo com um teste lógico.

A declaração **for** possui a seguinte sintaxe:

```
for (expressão_lógica) {
       instrução1;
       instrução2;
       ...
}
```

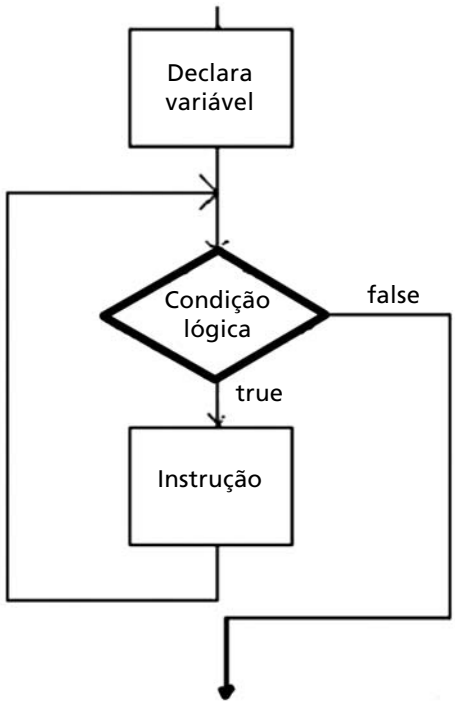


Figura 3.7: Fluxograma da declaração **for.**

O trecho de código a seguir exemplifica uma implementação da declaração **for:**

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

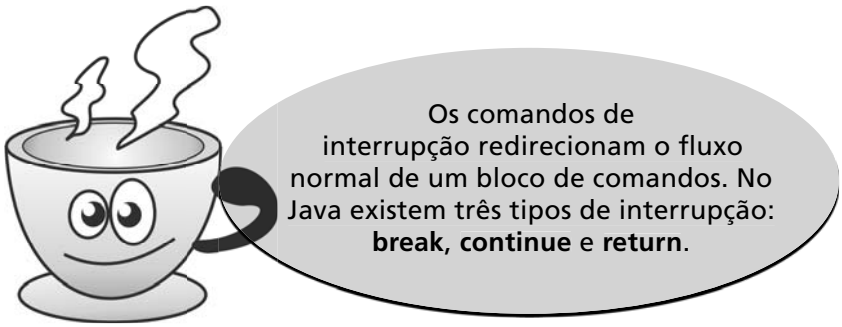
Que é um código que executa os mesmos comandos do trecho a seguir, mas dessa vez implementado com **while:**

```
int i = 0;
while (i < 10){
    System.out.println(i);
    i++;
}
```

Na leitura do laço **for** do nosso exemplo, temos o seguinte: para uma variável **i** inteira inicializada com **0** (esta é a declaração e inicialização da variável), execute o bloco de comando até ela ficar **menor do que 10** (esse é o teste lógico). Incremente-a de **1** (**i++**) após cada passagem pelo bloco até o teste dar **false**.

Então, o nosso exemplo vai imprimir o valor inicial de **i**, que é **0**, na primeira passagem pelo bloco por meio do método **System.out.println**... e após isso vai incrementar a variável **i**, então faz o teste lógico, como **i** passou de **0** para **1** e esse valor é menor do que **10**, o teste lógico retorna **true** e o bloco de comando é executado novamente. E assim sucessivamente, até o teste retornar **false**, quando a execução sai do bloco **for** para continuar a executar o restante do código.

INTERRUPÇÃO



Declaração break

A declaração **break** é utilizada juntamente com a declaração **switch** para fazer o escape desse bloco quando alguma das opções do **switch** é executada; caso contrário, o **switch** continua sua execução. Não é muito usual, mas você pode utilizar o **break** com a mesma finalidade (interromper o bloco de comandos e sair da estrutura de controle) para as instruções **for**, **while** ou **do-while**.

O trecho de código a seguir exemplifica uma implementação da declaração **break** quando utilizada junto com o **switch**:

```
int andar = 3;
switch(andar){
    case 1:
        System.out.println("Você escolheu o 1° andar!");
        break;
    case 2:
        System.out.println("Você escolheu o 2° andar!");
        break;
    case 3:
        System.out.println("Você escolheu o 3° andar!");
        break;
    default:
        System.out.println("ESCOLHA UM ANDAR VÁLIDO!");
}
```

Esse trecho de código faz a mesma operação que explicamos na parte de **switch**, mas agora a execução sai explicitamente do bloco **switch/case** ao entrar no **case** associado à variável numérica por causa do comando **break**, que "quebra" a execução da estrutura de comando.

Declaração continue

A declaração **continue** é utilizada para saltar da iteração corrente de uma estrutura de repetição para a próxima iteração. Assim, pode ser utilizada tanto com **for** quanto com **while** ou **do-while**.

O trecho de código a seguir exemplifica uma implementação da declaração **continue** sendo utilizada junto com o **for**:

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

        String alunos[] = {"Ana","Bia","Maria","Bia","José","Bia"};
        int count = 0;
        for (int i = 0; i < alunos.length; i++) {
            if(!alunos[i].equals("Bia")){
                continue;   //vai para a próxima iteração
            }
            count++;
        }
        System.err.println("Encontrado "+count+" vezes o nome de Bia no cadastro!!!");
    }
}
```

Esse programa implementa o algoritmo de contagem de uma determinada palavra dentro de uma coleção delas. Assim que o código entra no **if** (quando encontra um valor **diferente** de **Bia**, retornando **true**), ele encontra um **continue** e então vai para a próxima iteração do **for**, próxima passagem pelo bloco **for**, deixando de incrementar o contador e consequentemente de contar a palavra **Bia**. Porém, se ele encontra a palavra **Bia**, o **if** retorna **false** e ele não entra no seu bloco, executando o comando seguinte, que é justamente o incremento do contador. Assim, ao final, o contador representa quantas vezes **Bia** está presente no **array** alunos.

Declaração return

A declaração **return** é utilizada para sair de um método que está sendo executado ao final de todas as suas operações, retornando para o ponto exatamente após a chamada do método no código que o chamou.

Ao **return** pode estar associado um valor de determinado tipo (especificado na assinatura do método) ou pode não retornar nada (quando então não existe nada após o **return**, somente o símbolo de fim de comando **";"**).

Assim, podemos ter após o return o símbolo **";"** (quando o método apenas executa operações mas não retorna nada), **um valor** ou **uma variável** com valor compatível ao tipo de retorno do método.

Vamos ver os exemplos de utilização do **return**:

```
return ++i;
//retorna para o código que chamou o valor da variável i
incrementada

return 2;
//retorna para o código que chamou o valor 2

return "Ola";
//retorna para o código que chamou a String "Olá"

return;
//não retorna nada para o código que chamou, apenas passa
a execução para ele
```

ENTRADA DE DADOS E INTERATIVIDADE

Até agora, nossos programas não eram interativos, pois os dados de entrada dos programas eram inseridos diretamente no código.

Você vai estudar, a partir de agora, como deixá-los mais interessantes e interativos, fazendo com que alguns dados possam ser inseridos no momento da execução.

Nesta aula, você verá a forma mais básica de captura de dados do teclado por meio da classe BufferedReader. Nas próximas aulas, verá métodos mais sofisticados.

Entrada de dados com buffered reader

A classe **BufferedReader** do pacote java.io captura dados por intermédio do teclado.

Temos de tomar algumas "medidas administrativas" que serão explicadas mais tarde para utilizar a classe **BufferedReader**. A primeira delas é fazer a importação dessa classe para o nosso programa com o código a seguir (depois iremos importar outras classes):

```
import java.io.BufferedReader
```

A partir de então escrevemos a classe normalmente e, dentro do método *main*, instanciamos a classe passando como construtor a uma outra classe, a classe *InputStreamReader*, que recebe um *stream* de entrada (*bytes*) e no seu construtor quem será a entrada, no caso do *System.in*, que é a entrada padrão do sistema (o teclado):

```
BufferedReader dadEnt = new BufferedReader(new InputStreamReader(System.in));
```

Por enquanto, essa classe é apenas para você conseguir implementar alguma interatividade mais interessante nos seus exemplos e nas nossas atividades. Não se preocupe com os detalhes dos conceitos de instanciar uma classe, construtor e outros que serão vistos com mais detalhes nas aulas de Orientação a Objeto.

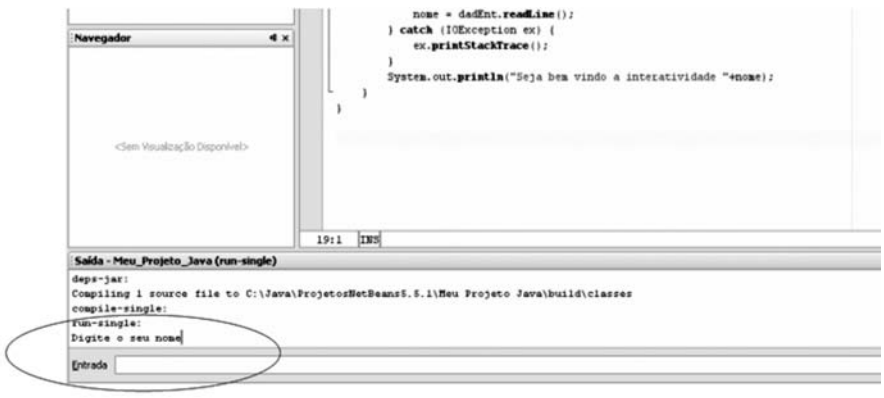
Após isso, precisaremos do código que faz a leitura da linha de entrada, que é o método *readLine( )* do seu objeto *BufferedReader*, no caso o *dadEnt*:

Nome = dadEnt.readLine( );

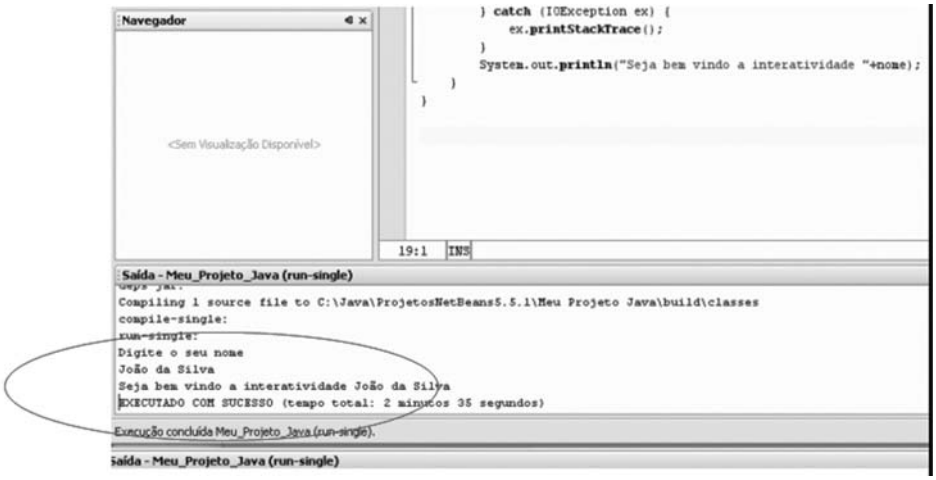
O código completo fica da seguinte forma:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class LeTeclado {

    public static void main(String[] args) {
        BufferedReader dadEnt = new BufferedReader(new InputStreamReader(System.in));
        String nome = "";
        System.out.println("Digite o seu nome");
        try {
            nome = dadEnt.readLine();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        System.out.println("Seja bem vindo a interatividade "+nome);
    }
}
```

Para executar de forma interativa o código anterior, após criar e implementar a classe no **NetBeans**, tecle **Shift+F6** para executar o programa, e uma caixa de entrada de dados irá se abrir na parte inferior da **IDE**:



Na janela **Entrada**, digite seu nome conforme foi solicitado no console; após teclar **Enter**, veja que seu nome será impresso como está descrito no código-fonte:



Agora,
vamos praticar
um pouco!
Utilize o NetBeans para executar as atividades. Vale lembrar que, quanto mais você exercitar a programação e utilizar a IDE, melhor programador se tornará. Além dos exercícios apresentados nestas aulas, procure criar seus próprios exercícios e exemplos, principalmente com situações do seu dia a dia. Isso estimulará o aprendizado e a criatividade, melhorando sua habilidade de programar.

Atividade prática **1**

Utilizando a classe **BufferedReader**, capture cinco palavras digitadas pelo usuário e mostre-as como uma única frase numa única linha.
Exemplo:
Palavra 1: Ola,
Palavra 2: bem
Palavra 3: vindo
Palavra 4: ao
Palavra 5: Java

Saída: Ola, bem vindo ao Java

Atividade prática **2**

Utilizando a classe **BufferedReader**, solicite um número ao usuário entre 0 e 10 e imprima seu valor por extenso. Caso o usuário imprima um valor inválido, escreva a mensagem "Valor inválido".

Atividade prática **3**

Utilizando a classe **BufferedReader**, faça um programa que lê três notas de um aluno inseridas via teclado e imprime sua média, dizendo se o mesmo foi aprovado na disciplina caso a nota seja maior do que cinco ou reprovado, caso contrário.

Atividade prática **4**

Faça um programa que imprima todos os números de 0 a 100.

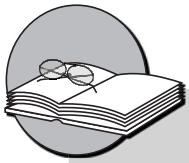
Atividade prática **5**

Faça um programa que imprima os números pares de 0 a 100.

RESUMO

Vamos rever os principais conceitos vistos nesta aula.

- Para executarmos programas sem ser de forma unicamente sequencial, utilizamos estruturas chamadas **estruturas de controle**.
- Essas estruturas podem ser divididas em: **estruturas de decisão**, **estruturas de repetição** e **interrupções**.
- As **estruturas de decisão** permitem a seleção de partes do código para execução, os chamados desvios, e podem ser do tipo **if** ou do tipo **switch**.
- As **estruturas de repetição**, como o próprio nome diz, permitem a repetição da execução de partes específicas do código, os chamados ***loopings***, e podem ser do tipo **while**, do tipo **do-while** ou do tipo **for**.
- As **interrupções** são estruturas que param e redirecionam o fluxo do programa de acordo com algum critério, e podem ser do tipo **break**, do tipo **continue** ou do tipo **return**.



Informações sobre a próxima aula

Na próxima aula, você irá estudar os **arrays** e verá como conseguimos com eles economizar linhas de código e manutenção quando precisamos lidar com uma quantidade muito grande de dados do mesmo tipo. Até lá...

Arrays

AULA 4

Meta da aula

Apresentar as principais formas de utilização de *arrays* e operações matriciais em Java.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. declarar e criar estruturas de *array* em Java;
2. acessar elementos de um *array*;
3. determinar o número de elementos de um *array*;
4. declarar e criar um *array* multidimensional.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você: tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; esteja animado e motivado para iniciar o aprendizado desta fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

INTRODUÇÃO

Nas aulas anteriores, você aprendeu a declarar diferentes variáveis usando os tipos de dados primitivos; isso é feito definindo um identificador seguido do tipo de dado que será associado àquela variável.

Para calcular a média de um aluno, você precisa inserir três notas (cujo conteúdo é um inteiro), com três variáveis do tipo **int**, com um identificador para cada variável.

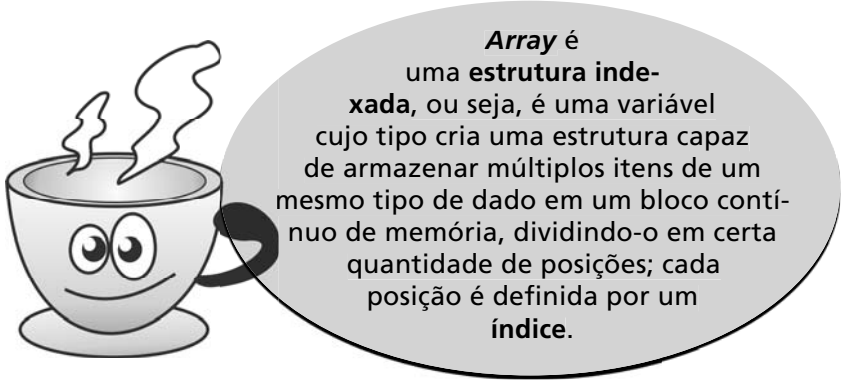
Como? Colocando três linhas para declaração e mais três para inicialização, conforme é mostrado a seguir:

```
int nota1;
int nota2;
int nota3;

nota1 = 5;
nota2 = 7;
nota3 = 8;
```

Agora imagine se você tivesse de utilizar cem variáveis! Isso seria no mínimo trabalhoso, não é mesmo?

Nesses casos, podemos utilizar um recurso que resolve esse problema, criando uma estrutura matricial para indexar (criar índices) as variáveis, como se fossem células sequenciais. Isso facilita sua criação, manipulação e controle.

Em Java essa estrutura matricial é um tipo de variável especial denominada *array*.



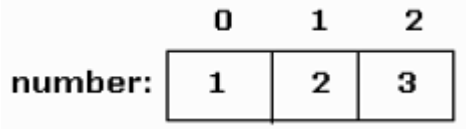


Figura 4.1: Exemplo de um *array* de inteiros.

DECLARANDO UM *ARRAY*

Como já dissemos, o *array* **é uma variável** e, como toda variável, precisa ser **declarada**. Mas ele tem uma declaração diferente dos tipos primitivos: é necessário definir qual tipo irá compor os elementos desse *array*, colocar essa definição seguida por um par de colchetes ([ ]) e pelo **identificador**. No exemplo a seguir, declaramos um *array* cujo conteúdo são números inteiros:

```
int[  ]  ages;
```

Existe também a possibilidade de colocar o par de colchetes após o identificador:

```
int  ages[  ];
```

Depois de declarado, precisamos criar o *array* e especificar seu tamanho, para alocá-lo em memória. Tecnicamente, esse processo se chama **construção**; assim, para executar tal atividade precisamos do auxílio de um **construtor**, que nesse caso será executado por meio da palavra reservada **new**:

```
// declaração do array de inteiros
int  ages[  ];
// construção do array declarado acima com espaço para cem
números inteiros
ages  =  new  int[100];
```

No exemplo, a declaração diz ao compilador Java que o identificador *ages* será usado como o nome de um *array* de inteiros cujo conteúdo terá cem elementos numéricos inteiros.

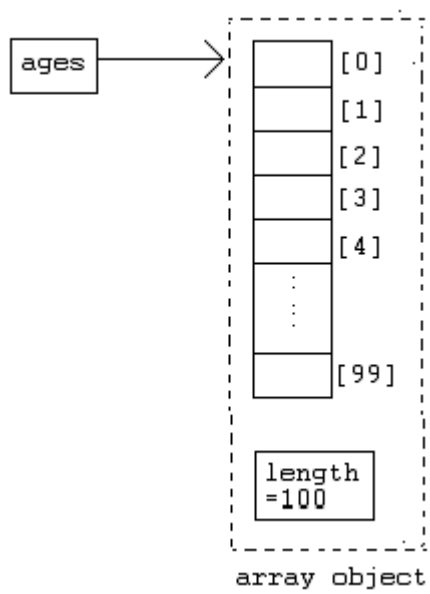


Figura 4.2: Instanciação de um *array* em memória.

Um detalhe importante é que podemos ainda **inicializar**, ou seja, colocar o conteúdo do dado no *array* ao mesmo tempo que é **construído** e **declarado**, ou seja, podemos executar a **declaração**, **construção** e **inicialização de um** *array* **numa mesma linha**, assim como fazemos com variáveis simples dos tipos primitivos. Veja os exemplos:

```
boolean  resultados[ ]  =  {true, false, false, true};
```

Nesse caso, declaramos uma variável **array** cujo identificador (cujo nome) é resultados e seu conteúdo é um valor lógico, ou seja, um **booleano** (*true* ou *false*), e ao mesmo tempo o inicializamos por meio do operador de atribuição = seguido de um par de chaves ({ }) cujo conteúdo são quatro valores **booleanos** separados por vírgulas; assim, o compilador assume implicitamente que seu *array* tem o tamanho quatro. Observe que nesse caso não foi necessário indicar o tamanho do *array*.

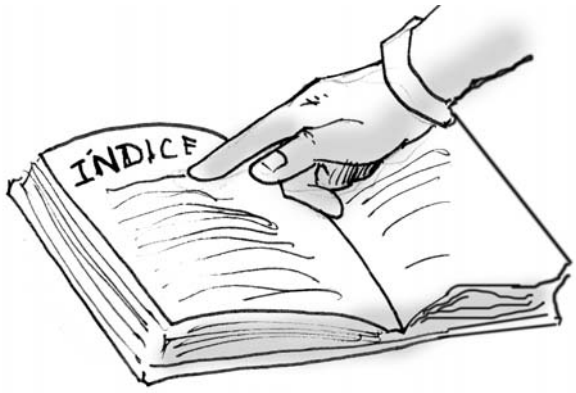
Veja outro exemplo:

```
double  [ ]notas  =  {8.0, 9.5, 6.7, 8.4, 4.0};
```

Neste caso, declaramos uma variável **array** cujo identificador (cujo nome) é **notas** e seu conteúdo é um valor numérico **double**; ao mesmo tempo o inicializamos através do operador de atribuição = seguido de um par de chaves ({ }), cujo conteúdo são cinco valores numéricos **double** separados por vírgulas; assim, o compilador assume implicitamente que seu *array* tem tamanho cinco; observe que nesse caso também não foi necessário indicar o tamanho do *array*.

Uma vez inicializado, o tamanho de um *array* não pode ser modificado, pois sua estrutura é alocada num determinado bloco contínuo de memória que não pode ser alterado.

ACESSANDO ELEMENTOS DE UM *ARRAY*



Nos conceitos iniciais, vimos que um *array* é uma estrutura que trata os seus elementos (seus dados) de forma indexada, ou seja, ele cria um índice para identificar cada elemento e, por meio desse índice, ele consegue recuperar essas informações.

O primeiro elemento do *array* recebe o índice 0 (zero), o segundo o índice 1, e assim sucessivamente.

Estruturas indexadas em computação são aquelas em que a estrutura associa cada elemento a uma forma de identificar sua posição. Essa forma de identificar é por meio de um índice, que dá origem ao nome estrutura indexada, que normalmente é numérico. Quando a estrutura inicia a indexação por 0 (zero), estamos fazendo uma indexação *zero base*; quando a indexação começa por 1 (um), estamos fazendo uma indexação *one base*. No caso de *arrays* em Java, a indexação é zero base, ou seja, o primeiro elemento do *array* é o elemento de índice 0, o segundo é o de índice 1, e assim sucessivamente.

Uma das vantagens de ter uma estrutura indexada é que conseguimos acessar individualmente e a qualquer momento cada elemento do *array*. Os índices são sempre inteiros e, **para um array de tamanho n, os índices vão de 0 até n-1**.

Lembra do *array ages*, que foi construído anteriormente? Se quisermos atribuir ou inicializar o primeiro elemento do *array* com o valor inteiro 10, já que *ages* é um *array* de inteiros, devemos escrever na nossa classe a seguinte linha após a construção do *array*:

```
ages[0] = 10;
```

Para imprimir o valor desse elemento do *array* na saída do sistema, é só utilizar o método *println*, passando como argumento o elemento do *array* que queremos imprimir:

```
System.out.println(ages[0]);
```

Uma dica importante é que você pode utilizar estruturas de repetição para manipular *arrays*, seja para preencher o valor de seus elementos, seja para imprimi-los. Veja o exemplo a seguir, que, após construir o *array*, utiliza uma estrutura **for** para inicializá-lo com valores de 0 a 100 e imprimi-lo, o que é muito mais fácil do que ter de digitar ou copiar e colar cem vezes o método System.out.println(...).

```
public class ArrayComLooping {
    public static void main(String[] args){
        int[] ages = new int[100];
        for (int i=0; i<100; i++){
            ages[i] = i;
            System.out.println(ages[i]);
        }
    }
}
```

Atividade prática 1

Abra sua ***IDE NetBeans***, implemente e execute uma classe que construa um ***array*** com capacidade de armazenar 100 inteiros. Inicialize então cada elemento do ***array*** com os números ***de 0 até 99*** e imprima-os por intermédio de uma estrutura ***for***, como foi exemplificado.

Atividade prática **2**

Implemente e execute uma nova classe que construa um ***array*** com capacidade de armazenar 100 inteiros, mas dessa vez inicialize cada elemento do ***array*** com os números ***de 1 até 100*** e imprima-os por meio de uma estrutura ***for***, conforme foi exemplificado.

DESCOBRINDO O TAMANHO DE UM *ARRAY*

Muitas vezes você pode receber um *array* para percorrer ou recuperar seus dados, mas não sabe o tamanho dele. Para isso, todo *array* possui um atributo **lenght** que retorna seu tamanho, ou seja, a quantidade de elementos que o *array* pode armazenar (é o espaço que foi alocado em memória por ocasião de sua construção, lembra-se?).

A forma de utilização do atributo **lenght** é por meio do nome do *array* seguido pelo operador "." seguido do próprio atributo **lenght**:

```
nomeDoArray.lenght
```

Assim, podemos melhorar o código do exemplo anterior fazendo com que o **for** descubra sozinho até que índice ele deve percorrer:

```
public class ArrayComLooping {
    public static void main(String[] args){
        int[] ages = new int[100];
        for (int i=0; i<ages.length; i++){
            ages[i] = i;
            System.out.println(ages[i]);
        }
    }
}
```

Se você, por descuido, exceder o valor que deve ser percorrido na utilização do *looping* dentro do *array*, será lançada uma **Exception**, pois ocorreu um erro; por exemplo, caso você faça o **for** ir até o índice 100 utilizando i < 101 (que não existe no nosso *array*) em vez de 99 com i < 100 (que é o último índice do *array* e, consequentemente, o valor do limite do for), isso vai ultrapassar os limites do *array* lançando uma exceção do tipo **ArrayIndexOutOfBoundsException**.
Por enquanto, não se preocupe com esses conceitos de **Exceptions**, apenas saiba que existem e que podem ocorrer. Futuramente, teremos uma aula somente sobre esse assunto.

Atividade prática 3

Implemente e execute no **NetBeans** uma classe que construa um ***array*** com capacidade de armazenar 100 inteiros; inicialize então cada elemento do **array** com os números de **0 até 99** e imprima-os por meio de uma estrutura ***for*** como já foi exemplificado, mas em vez de definir explicitamente o intervalo do ***looping***, utilize o atributo ***lenght*** do ***array*** para definir seu tamanho.

Atividade prática 4

Implemente e execute no **NetBeans** uma nova classe semelhante à da **Atividade 1**, mas desta vez modifique os parâmetros da estrutura **for** para forçar a saída de um erro no sistema, forçando o lançamento de uma **Exception** (vide caixa de ênfase anterior).

ARRAYS MULTIDIMENSIONAIS

Às vezes, precisamos representar um *array* multidimensional, por exemplo, uma matriz de duas dimensões. Essa abstração é feita em Java por meio de um *array* dentro de outro *array*.

Para fazer isso, você utiliza um novo par de colchetes depois do par de colchetes que já existe no *array* unidimensional.

Veja no exemplo como criar um *array* de duas dimensões para armazenar 64 por 64 números inteiros (4.096 no total). Isso criará uma estrutura com 64 linhas e 64 colunas:

```
int [ ] [ ]  array2d = new int [64] [64];
```

ENTRADA DE DADOS COM A CLASSE *SCANNER*

Veja como melhorar a interatividade dos programas por meio da entrada de dados em tempo de execução.

Da última vez utilizamos a classe *Buffered Reader* para fazer isso, lembra-se?

Desta vez vamos utilizar uma classe mais nova e mais simples: a classe Scanner do pacote *java.util*.

Vamos mostrar um exemplo utilizando a classe *Scanner* e depois explicaremos as partes mais importantes.

```
1  import java.util.Scanner;
2
3  public class EntradaDeDadosScanner {
4      public static void main(String args[]){
5          Scanner sc = new Scanner(System.in);
6          System.out.println("Digite seu nome: ");
7          String nome = sc.next();
8          System.out.println("Seja bem vindo "+nome+" !!!");
9      }
10 }
```

Na **linha 1** indicamos ao compilador que iremos utilizar uma classe externa e indicamos o caminho completo dos pacotes e subpacotes da classe:

```
import java.util.Scanner;
```

Em seguida, nas **linhas 3 e 4**, o que já sabemos: definição e declaração da classe e do método main, normal...

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

Agora sim: na **linha 5** declaramos uma variável cujo nome é **sc** (poderia ser qualquer coisa) e dissemos que ela é do tipo Scanner, ou seja, é um objeto que faz tudo que *Scanner* faz. Construímos então o objeto com *new* e passamos como argumento ao construtor, que receberá dados da entrada (***in***) padrão do sistema (***System***), que é o teclado:

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

Na **linha 6** imprimimos na tela um pedido para o usuário digitar seu nome. Na linha seguinte declaramos uma variável do tipo *String* para receber a palavra digitada pelo usuário, ela é capturada através do método ***next*** de **Scanner**, e é representada pela variável ***sc***:

```
System.out.println("Digite seu nome: ");
String nome = sc.next();
```

Por fim, imprimimos o valor da *String* armazenada (atribuída) na variável ***nome*** na linha anterior:

```
System.out.println("Seja bem vindo "+nome+" !!!");
```

Além do método **next( )**, que armazena uma entrada **String**, podemos também capturar outros tipos de dados com outros métodos da classe Scanner:

MÉTODO	FUNÇÃO
next( )	Captura uma entrada no formato ***String*** e a retorna para a classe
nextInt( )	Captura uma entrada no formato ***int*** e a retorna para a classe
nextByte( )	Captura uma entrada no formato ***byte*** e a retorna para a classe
nextLong( )	Captura uma entrada no formato ***long*** e a retorna para a classe
nextFloat( )	Captura uma entrada no formato ***float*** e a retorna para a classe
nextDouble( )	Captura uma entrada no formato ***double*** e a retorna para a classe

Atividade prática **5**

Implemente e execute no ***NetBeans*** uma classe que constrói um *array* de ***Strings*** inicializado com os nomes dos sete dias da semana. Por exemplo:

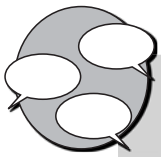
```
String dias[ ] = {“Segunda”, “Terça”, “Quarta”, “Quinta”, “Sexta”,
“Sábado”, “Domingo”};
```

Após isso, na mesma classe, utilize um laço ***while*** para imprimir o conteúdo desse *array*. Faça o mesmo com as declarações ***do-while*** e ***for***.

Atividade prática **6**

Desafio

Implemente e execute no ***NetBeans*** uma classe que, por meio do ***Scanner***, solicita 10 números inteiros ao usuário. Então utilize um *array* para armazenar o valor desses números, imprima todos eles e por último mostre o número de maior valor.



INFORMAÇÕES SOBRE FÓRUM

Entre no fórum para discutir suas dúvidas! Participe e troque informações.
Também não perca o *chat* da semana. O horário e o dia serão colocados na página do fórum.

RESUMO

Reveja os principais conceitos estudados nesta aula:

- Muitas vezes queremos criar estruturas semelhantes numa quantidade muito grande, por exemplo, cem variáveis do tipo inteiro. Isso daria muito trabalho se fizéssemos linha a linha.
- Para resolver esse tipo de problema, as linguagens de programação sabem trabalhar com estruturas matriciais, que no caso do Java recebem o nome de *arrays*.
- Os *arrays* têm a capacidade de armazenar múltiplas estruturas de dados, com a limitação de serem do mesmo tipo e de seu tamanho não poder ser modificado, ou seja, só armazenam uma quantidade predefinida de elementos.
- A declaração de um *array* em Java é feita simplesmente utilizando um par de colchetes **[ ]** após o tipo ou o após o nome da variável que se quer que seja um *array*.

```
int [ ] idades ou int idades [ ];
```

- Um *array* pode ser declarado e inicializado na mesma linha, quando o compilador assume que o tamanho do array é o da quantidade de elementos inicializados:

```
double  [ ]notas  =  {8.0, 9.5, 6.7, 8.4, 4.0};
```

- Para acessar ou inicializar individualmente um *array*, utilizamos o índice atribuído à posição daquele elemento do *array*; em Java os *arrays* iniciam sua indexação em zero, ou seja, para um *array* de tamanho N, temos os índices de 0 até N –1. Assim, para inicializar e imprimir o elemento 0 de um *array*, o primeiro elemento, escrevemos da seguinte forma:

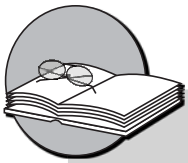
```
ages[0] = 10;                 /inicialização com o valor 10
System.out.println(ages[0]); //imprimir o elemento de índice 0
```

- Para descobrir o tamanho de um *array* desconhecido utilizamos o atributo *lenght*:

```
nomeDoArray.lenght
```

- Um *array* pode ser uma estrutura multidimensional, bastando para isso unicamente acrescentar um par de colchetes após o par inicial para cada dimensão a mais que quisermos criar. Por exemplo, para um *array* bidimensional, para armazenar uma matriz numérica com 64 linhas e 64 colunas, fazemos da seguinte forma:

```
int [ ] [ ]  array2d = new int [64] [64];
```



Informação sobre a próxima aula

Na próxima aula, você começará o mergulho no fantástico mundo do Java e orientação a objeto. Você vai começar a ver por que esse paradigma de programação dominou o mundo e serve como base para todas as novas linguagens que surgem no mercado. Até lá...

AULA 5

Orientação a Objeto I

Meta da aula

Apresentar os principais conceitos de Orientação a Objeto em Java.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. reconhecer o que é Programação Orientada a Objeto;
2. diferenciar classes e objetos;
3. explicar o que são atributos e como manipulá-los;
4. explicar o que são métodos e como manipulá-los;
5. reconhecer como fazer conversões de tipos de dados em Java;
6. reconhecer e implementar o conceito de encapsulamento;
7. criar e utilizar suas próprias classes.

Pré-requisitos

Para o correto e completo aprendizado dessa aula, precisamos que você tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; tenha entendido bem as aulas estudadas até agora e realizado com aproveitamento todas as atividades, já que entraremos numa nova fase do aprendizado; esteja animado e motivado para iniciar o aprendizado desta fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

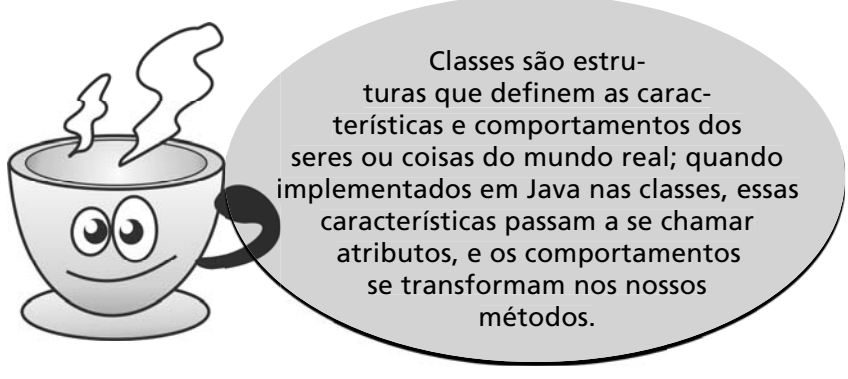
INTRODUÇÃO

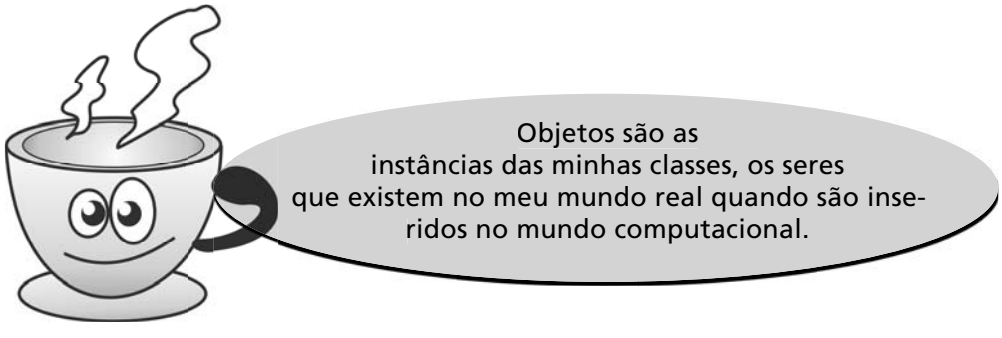
A PROGRAMAÇÃO ORIENTADA A OBJETO

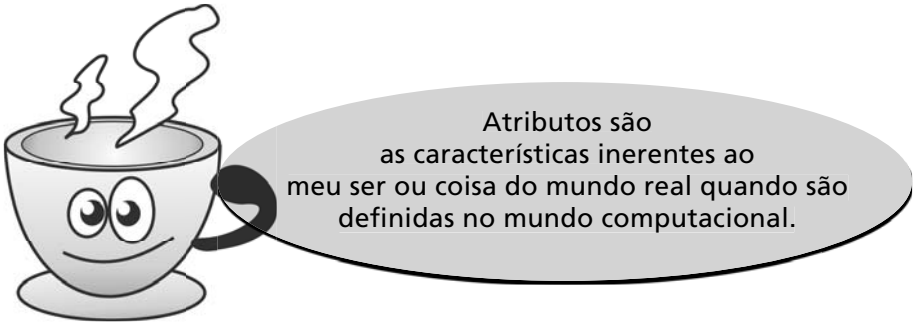
A **Programação Orientada a Objeto (POO)** tem como pilar teórico o conceito de **objeto**, ou seja, um sistema orientado a objeto é um conjunto de objetos que representam os seres e coisas do mundo real, interagindo computacionalmente com as mesmas características e comportamentos reais.

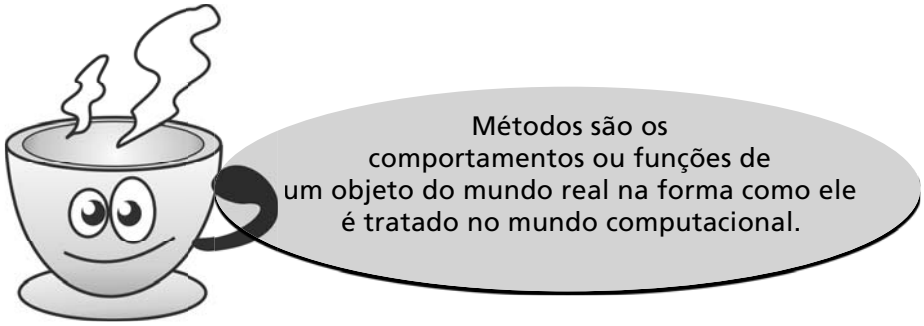
À estrutura computacional que define o modelo de um objeto do mundo real damos o nome de **classe**, e a partir dela cópias são construídas para serem utilizadas para cada objeto real que tenho no meu sistema.

As **características** dos objetos do mundo real são chamadas de **atributos** no mundo computacional, enquanto a seus **comportamentos** chamamos métodos.









Por exemplo, um objeto carro no mundo real tem as propriedades cor, ano, número de portas e tipo de combustível, entre outras. Além disso, tem os comportamentos acelerar, frear, virar à esquerda e outros. Assim, quando modelamos esse carro computacionalmente para um jogo de corridas, por exemplo, definimos uma classe carro cujas propriedades são transformadas nos atributos cor, ano, número de portas, tipo de combustível e seus comportamentos são transformados no código do programa nos métodos acelerar, frear, virar à esquerda e outros.

Como podemos ver, a filosofia da Programação Orientada a Objeto é simplificar o modelo e a implementação do ponto de vista da representação, de tal forma que você programador não precise ficar decifrando códigos e números do modelo, mas apenas escreva naturalmente tudo que está representado no seu modelo de negócio do mundo real.

Se você está modelando um *software* para um sistema escolar, provavelmente terá uma classe professor, uma classe aluno, uma classe departamento, cada uma com características e comportamentos de acordo com o objeto do mundo real. Se for um sistema para uma veterinária, você provavelmente terá uma classe animal. Se for um sistema de seguros para carro, terá uma classe carro. Se for um sistema bancário, uma classe cliente, uma classe conta-corrente. Assim, percebemos que quem define o sistema é o próprio negócio; cabe ao analista ou ao programador entender o negócio.

CLASSES E OBJETOS

Uma das primeiras confusões que quem está começando a estudar orientação a objeto faz é confundir os conceitos de classes e objetos. Vamos ver mais um pouco desses conceitos para não deixarmos dúvidas.

Classe representa o modelo do que queremos utilizar no nosso sistema; seria como um gabarito, um protótipo, a planta de uma casa ao realizarmos uma construção, a receita de um bolo.

Veja o exemplo da tabela a seguir, na qual temos as definições de uma classe que representa o carro do mundo real; ao lado, os objetos carros que iremos implementar no nosso sistema:

CLASSE CARRO		OBJETO CARRO A	OBJETO CARRO B
Atributos de objeto	**Marca**	Ford	Mitsubishi
	Modelo	Fiesta	L-200
	Cor	branco	azul royal
	Combustível	gasolina	diesel
Métodos	ligar		
	acelerar		
	frear		

Assim, durante a execução do sistema, a cada carro que cadastramos uma cópia nova da classe carro é criada, e damos a ela características que irão diferenciá-la dos demais carros, ou seja, criamos a cada vez um novo objeto do tipo de carro.

Ah! Então quer dizer que classe é um tipo? Isso mesmo.

O principal benefício da utilização da programação por meio de classes é a reutilização do código, pois a cada objeto criado você não precisa criar sua estrutura novamente. E isso é só o começo, você verá muito mais quando estudar outros conceitos como herança, polimorfismo etc.

Como você pôde perceber na tabela anterior, na coluna abaixo de carro escrevemos **Atributos de objeto**, que é o mais comum, pois o valor do conteúdo que será colocado é referente àquele objeto. Por exemplo, o **Carro a** no atributo **Marca** tem o conteúdo **Ford**, enquanto o **Carro b** tem o conteúdo **Mitsubishi**. Os atributos estão associados ao objeto; sua definição está feita na classe.

Porém, algumas vezes podemos querer um atributo que seja da classe, ou seja, é o mesmo para todos os objetos, podendo ser manipulados dentro da própria classe. Imagine um contador de carros numa loja, se cada objeto Carro tiver um atributo contador eu nunca conseguirei saber quantos carros eu tenho na minha loja. Nesse caso, em que criamos um atributo global, temos o que chamamos de **atributo da classe**, e em memória esse valor fica guardado no espaço reservado para a classe, e não para o objeto.

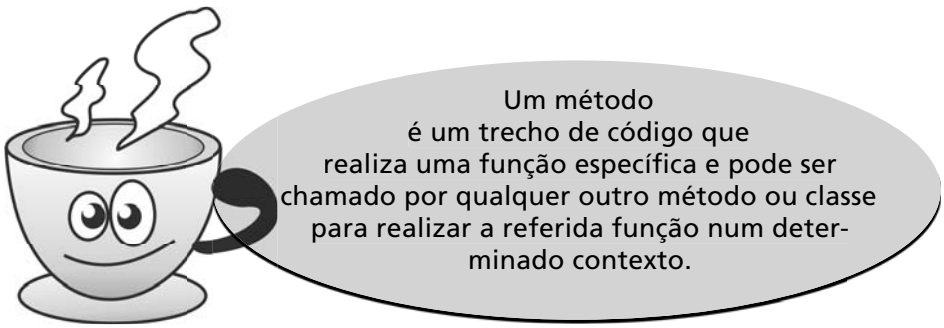
A partir de um modelo de uma classe, para criar cada objeto, utilizamos o operador **new**. Por exemplo, para criar uma instância do **Carro a** a partir da classe **Carro**, utilizamos o seguinte código:

```
Carro a = new Carro( );
```

O operador **new** reserva (aloca) a memória para o objeto conforme a estrutura de sua classe e retorna uma referência para essa área que foi alocada. O código que vem após o operador **new** é o que chamamos de **construtor** da classe e tem o mesmo nome da classe seguido de ( ) (abre e fecha parênteses); dentro deles podem vir parâmetros, dependendo da classe a ser utilizada e de sua definição.

MÉTODOS

Em todos os exemplos de programas que fizemos até agora, nossas classes possuíam apenas um método, o **main**( ) lembra? Mas em Java podemos criar quantos métodos quisermos e chamá-los dentro da classe ou a partir de outra classe.



Os métodos possuem as seguintes características:

• podem ou não retornar um valor;

• podem ou não aceitar argumentos;

• após encerrar sua execução, o método retorna o fluxo de controle do programa para quem o chamou.

Além de métodos comuns, temos também os chamados **métodos estáticos**, que são aqueles que podem ser invocados sem que um objeto tenha sido instanciado (sem utilizar o new). Métodos estáticos pertencem à classe e não ao objeto. É o equivalente dos atributos de classe. Num exemplo mais adiante, você verá a aplicação de um método estático.

CONVERSÃO DE TIPO

Casting ou **typecasting** é o processo no qual fazemos a conversão de tipos de um atributo ou dado para outro tipo de dado.

Em Java, uma vez que definimos o tipo de um atributo, não podemos associar a esse atributo um tipo diferente. Caso seja necessário fazer isso, temos de fazer uma operação de **casting**.

O tipo mais simples de *casting* é o realizado entre tipos primitivos, muito comum entre tipos numéricos. O único tipo primitivo que não aceita *casting* é o **boolean**.

Considere que seja necessário armazenar um valor do tipo **int** em um atributo do tipo **double**; nesse caso, o *casting* é feito automaticamente, é o chamado **casting implícito**, pois estamos "colocando um atributo de tamanho menor dentro de um de tamanho maior", pois o **double** é

maior que o **int**. Essa regra é geral e serve para todos os tipos primitivos (à exceção do **boolean**, como já escrevemos):

```
int numInteiro = 10;
double numDouble = numInteiro;   //casting implícito
```

Quando executamos a operação inversa, ou seja, quando convertemos um atributo cujo tipo possui um tamanho maior para um de tamanho menor, necessariamente devemos fazer o **casting explícito**, colocando o tipo a ser convertido entre parênteses na frente do valor a converter:

```
double numDouble2 = 45.32;
int numInt2 = (int)numDouble2;
```

Agora que estamos falando de tipos, classes e objetos, veja alguns conceitos complementares.

Numa das aulas anteriores, vimos operadores que faziam comparação de valores, lembra? Assim, conseguíamos saber se um atributo era igual ao outro, se era maior, se era menor ou igual etc. Um pequeno mas importante detalhe é que a maioria desses operadores só funciona com tipos primitivos. As exceções são os operadores de igualdade (==) e de diferença (!=). Mas apesar de poderem ser usados em objetos, quando utilizados com esses tipos eles não verificam se os objetos possuem o mesmo conteúdo (o que era de se supor, pois é isso que ocorre com tipos primitivos), mas sim se possuem a mesma referência em memória. Para comparar o conteúdo de dois objetos, utilizamos o método **equals**( ), conforme será exemplificado a seguir.

Vamos analisar as classes:

```
1  class TesteIgualdade {
2
3      public static void main(String args[]){
4          String frase1;
5          String frase2;
6          frase1 = new String("Testando igualdade!");
7          frase2 = new String("Testando igualdade!");
8          System.out.println("Frase 1: "+frase1);
9          System.out.println("Frase 2: "+frase2);
10         System.out.println("Frase 1 e Frase 2 são iguais? R: "+(frase1==frase2));
11     }
12 }
```

Ao executar essa classe, a classe **TesteIgualdade**, temos a seguinte saída:

Nas linhas 4 e 5 declaramos dois atributos, frase1 e frase2, do tipo String. Observe que String não é um tipo primitivo, mas é uma classe, pois começa com letra maiúscula.

Não é isso que o define como classe, mas lembre-se de que a boa prática de programação, vista no **Code Conventions**, nas aulas iniciais, prevê que classes devem começar com letra maiúscula; assim podemos supor que String é uma classe.

No construtor dos atributos do tipo String frase1 e frase2 (ou seja, após o new), passamos como argumento a mesma frase, ou seja, o atributo tem o mesmo valor. Mas, ao comparar a igualdade de frase1 e frase2, o que é feito na linha 10, o resultado retorna *false*! Isso ocorre porque, apesar de possuírem o mesmo conteúdo, frase1 e frase2 possuem referências diferentes, ou seja, diferentes espaços alocados em memória; isso ocorre porque foi construído um objeto para cada String, um new para cada atributo frase.

Atividade prática 1

Abra a sua **IDE NetBeans** e implemente a classe **TesteIgualdade**, que está apresentada anteriormente; depois, teste o exemplo e veja se você conseguiu visualizar melhor o exemplo explicado, compreendendo os conceitos associados.

Caso ainda tenha dúvidas sobre esses conceitos, interaja com seu tutor ou coloque suas dúvidas no fórum.

Veja um outro exemplo:

```
1  class TesteIgualdade2 {
2
3      public static void main(String args[]){
4          String frase3;
5          String frase4;
6          frase3 = new String("Testando igualdade!");
7          frase4 = frase3;
8          System.out.println("Frase 3: "+frase3);
9          System.out.println("Frase 4: "+frase4);
10         System.out.println("Frase 3 e Frase 4 possuem a mesma referencia? R: "+(frase3==frase4));
11     }
12 }
```

Ao executar a classe **TesteIgualdade2** temos a seguinte saída:

A classe é muito semelhante à anterior, mas em vez de utilizar um construtor na frase4 na linha 5, nós atribuímos frase3 a ela; com isso, a mesma referência de frase3 é dada à frase4. Assim, podemos perceber que, mesmo sem escrever uma frase como conteúdo de frase4, como ela aponta para a mesma referência de frase3, ao mandarmos imprimir frase4 ela tem a mesma saída de frase3 (o que fazemos nas linhas 8 e 9); quando perguntamos se elas possuem a mesma referência, o resultado é *true*, ou seja, qualquer alteração que eu faça no conteúdo de frase3 vai refletir em frase4 e vice-versa, pois as duas possuem a mesma referência.

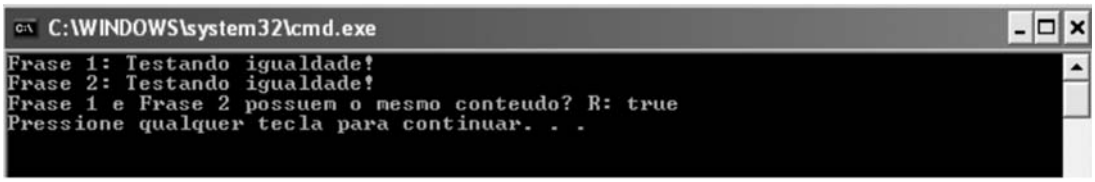
Atividade prática 2

Abra a sua **IDE NetBeans** e implemente a classe **TesteIgualdade2** já apresentada; depois, teste o exemplo e veja se você conseguiu visualizar melhor o exemplo explicado, compreendendo os conceitos associados. Caso ainda tenha dúvidas sobre esses conceitos, interaja com seu tutor ou coloque suas dúvidas no fórum.

Vamos ver um último exemplo, para consolidar o conceito de referência e de conteúdo de objetos:

```
1  class TesteIgualdade3 {
2
3      public static void main(String args[]){
4          String frase1;
5          String frase2;
6          frase1 = new String("Testando igualdade!");
7          frase2 = new String("Testando igualdade!");
8          System.out.println("Frase 1: "+frase1);
9          System.out.println("Frase 2: "+frase2);
10         System.out.println("Frase 1 e Frase 2 possuem o mesmo conteudo? R: "+(frase1.equals(frase2)));
11     }
12 }
```

Ao executar a classe **TesteIgualdade3** temos a seguinte saída:



Como explicamos, a forma de compararmos o conteúdo de um atributo do tipo objeto não é com o operador de igualdade (==), o que se faz normalmente com tipos primitivos, mas com o método **equals( )** associado aos dois objetos a serem comparados, como está mostrado no exemplo.

Atividade prática 3

Abra a sua **IDE NetBeans** e implemente a classe **TesteIgualdade3** já mostrada; depois, teste o exemplo e veja se você conseguiu visualizar melhor o exemplo explicado, compreendendo os conceitos associados.

Caso ainda tenha dúvidas sobre esses conceitos, interaja com seu tutor ou coloque suas dúvidas no fórum.

CRIANDO E MANIPULANDO NOSSAS PRÓPRIAS CLASSES

O que fizemos até agora foi utilizar classes externas já existentes (como a String) e utilizar métodos já existentes (como o **equals( )**). Agora, você estudará como criar e utilizar suas próprias classes.

Vamos criar uma classe que modela as informações de um aluno do mundo real, inserindo as operações necessárias para seu registro.

Como essa classe modela a operação de registro referente a um aluno, um bom nome para ela seria **RegistraAluno**.

Então vamos começar nossa classe **RegistraAluno** apenas com sua definição:

```
1  public class RegistraAluno {
2      // demais partes do código
3  }
```

Onde:

- **public:** é um modificador de acesso (será mais bem explicado futuramente) e define que qualquer classe pode acessar a classe **RegistraAluno**;
- **class:** é a palavra-chave usada para criação de uma classe;
- **RegistraAluno:** é o identificador único que nomeia a classe.

Vamos agora começar a declarar os atributos. Neste momento, precisamos pensar quais as informações de aluno que são importantes para o meu sistema. Vamos hipoteticamente criar um sistema no qual queremos armazenar as informações do aluno relativas a nome, endereço, idade, nota de matemática, nota de português e nota de geografia. Teremos então a seguinte definição para nossa classe:

```
1  public class RegistraAluno {
2      private String nome;
3      private String endereco;
4      private int idade;
5      private double notaMatematica;
6      private double notaPortugues;
7      private double notaGeografia;
8  }
```

Vale ressaltar que, quando queremos armazenar alguma informação textual sobre um objeto, o tipo mais ideal do atributo é a classe **String**, como fizemos com o atributo nome e endereço. Além disso, escolhemos o tipo primitivo **int** para armazenar a idade, já que é um valor inteiro, e o tipo primitivo **double** para armazenar as notas, já que estas são todas com precisão de duas casas à direita da vírgula.

Algumas observações de boas práticas:

- As declarações dos atributos são feitas sempre na primeira parte do código da classe.
- Declare um atributo por linha, mesmo que ele seja do mesmo tipo.
- Declare atributo como **private**; assim, somente a própria classe pode manipulá-lo; esse é o conceito de **encapsulamento**, muito importante na Orientação a Objeto.

Além dos atributos do objeto, que será individual para cada aluno, podemos declarar também um atributo que será compartilhado por todos os objetos daquele tipo. Na verdade, o que criamos é o que chamamos de **atributo de classe**. Isso é feito por meio da palavra **static** antes da definição do atributo. No nosso exemplo, um bom atributo de classe pode ser aquele que vai contar a quantidade de alunos, o **contadorEstudante**:

```
1  public class RegistraAluno {
2      private String nome;
3      private String endereco;
4      private int idade;
5      private double notaMatematica;
6      private double notaPortugues;
7      private double notaGeografia;
8
9      private static int contadorEstudante;
10
11 }
```

As classes são compostas por atributos e métodos. O primeiro já resolvemos; vamos agora implementar os comportamentos que desejamos que sejam executados pelo nosso objeto, os métodos.

Vamos completar o conceito de **encapsulamento**. Nós começamos explicando que devemos declarar nossos atributos dos objetos como **private**, pois assim os protegemos e somente a própria classe pode manipulá-los. Mas então como faremos para modificá-los? Por meio de métodos públicos (esses, sim, todos que quiserem podem acessar), chamados **métodos acessores**. O método acessor tem o modificador **public** e recebe o nome **get** seguido do nome do atributo que será acessado através dele. Ele sempre retorna um valor do mesmo tipo do atributo acessado.

No nosso caso, teremos métodos assessores para nome, endereço e idade. Além disso, implementaremos um método **get** para recuperar não cada nota, mas sim a média delas, o método **getMedia**( ). Quando

definimos um método que recupera algum valor por meio de alguma operação com os atributos da classe, esse método também deve ter seu nome iniciado com **get.** Assim, faremos o mesmo para o método que retorna a quantidade de estudantes inscritos, o método **getQuantidadeAlunos**( ). Veja todos esses métodos implementados no código da classe a seguir:

```
1  public class RegistraAluno {
2      private String nome;
3      private String endereco;
4      private int idade;
5      private double notaMatematica;
6      private double notaPortugues;
7      private double notaGeografia;
8
9      private static int contadorEstudante;
10
11     // retorna o nome do estudante
12     public String getNome(){
13         return nome;
14     }
15
16     // retorna o endereço do estudante
17     public String getEndereco(){
18             return endereco;
19     }
20
21     // retorna a idade do estudante
22     public int getIdade(){
23             return idade;
24     }
25
26     // retorna a média do estudante
27     public double getMedia(){
28         double resultado = 0;
29         resultado = (notaMatematica + notaPortugues + notaGeografia) / 3;
30         return resultado;
31     }
32
33     // retorna a quantidade de estudantes cadastrados
34     public static int getQuantidadeAlunos(){
35         return contadorEstudante;
36     }
37
38 }
```

Está tudo bem. Mas, se você prestar atenção, perceberá que temos um pequeno mas importante detalhe: já conseguimos recuperar as informações; mas onde elas foram definidas?

Em lugar algum; na verdade, vamos ver agora esse conceito: para recuperar algum valor de alguma variável, ele precisa ser definido antes. Entretanto, nossas variáveis não foram inicializadas, até porque isso não faria sentido, pois um aluno pode ter qualquer valor para idade, nome ou endereço; sendo assim, temos métodos de definição, que definem valores para variáveis, os métodos **set.**

E assim completamos nosso conceito de encapsulamento, em que temos classes dos objetos de negócio com atributos privados (acessíveis somente pela própria classe) e métodos públicos (acessíveis por quaisquer classes), **get** para recuperar os valores dos atributos e métodos públicos **set** para defini-los. No nosso caso, temos métodos **set** para os atributos nome, endereço, idade e para cada uma das notas, como você pode ver neste código:

```
1  public class RegistraAluno {
2      private String nome;
3      private String endereco;
4      private int idade;
5      private double notaMatematica;
6      private double notaPortugues;
7      private double notaGeografia;
8
9      private static int contadorEstudante;
10
11     // retorna o nome do estudante
12     public String getNome(){
13         return nome;
14     }
15
16     // define ou altera o nome do estudate
17     public void setNome(String temp){
18         nome = temp;
19     }
20
21     // retorna o endereço do estudante
22     public String getEndereco(){
23             return endereco;
24     }
25
26     // define ou altera o endereco do estudate
27         public void setEndereco(String temp){
28             endereco = temp;
29     }
30
31     // retorna a idade do estudante
32     public int getIdade(){
33             return idade;
34     }
35
36     // define ou altera idade do estudate
37         public void setEndereco(int temp){
38             idade = temp;
39     }
40
```

```
41      // define ou altera as notas
42          public void setNotaMatematica(double temp){
43              notaMatematica = temp;
44      }
45
46      public void setNotaPortugues(double temp){
47              notaPortugues = temp;
48      }
49
50      public void setNotaGeografia(double temp){
51              notaGeografia = temp;
52      }
53
54      // retorna a média do estudante
55      public double getMedia(){
56          double resultado = 0;
57          resultado = (notaMatematica + notaPortugues + notaGeografia) / 3;
58          return resultado;
59      }
60
61      // retorna a quantidade de estudantes cadastrados
62      public static int getQuantidadeAlunos(){
63          return contadorEstudante;
64      }
65
66  }
```

Por fim, vamos implementar uma classe (classe **AppRegistraAluno**), que representa uma aplicação que utiliza nosso objeto aluno do mundo real, representado computacionalmente pela classe **RegistraAluno**:

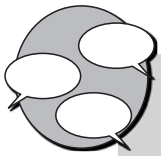
```
1   public class AppRegistraAluno {
2       public static void main(String args[]){
3
4           // cria 3 objetos RegistraAluno
5           RegistraAluno ana = new RegistraAluno();
6           RegistraAluno beto = new RegistraAluno();
7           RegistraAluno carlos = new RegistraAluno();
8
9           ana.setNome("Ana Machado");
10          beto.setNome("Roberto da Silva");
11          carlos.setNome("Carlos Alberto");
12
13          System.out.println(ana.getNome());
14
15          System.out.println("Contador: "+RegistraAluno.getQuantidadeAlunos());
16      }
17
18  }
```

Como foi pedido na aplicação, imprimimos o nome da Ana pelo método **getNome**() aplicado ao objeto ana, além do contador por meio do método **getQuantidadeAlunos**( ).

Atividade prática **4**

Abra a sua **IDE NetBeans** e implemente as classes **RegistraAluno** e **AppRegistraAluno** que já foram explicadas. Depois teste os exemplos.

Caso ainda tenha dúvidas sobre esses conceitos, interaja com seu tutor ou coloque suas dúvidas no fórum.



INFORMAÇÕES SOBRE FÓRUM

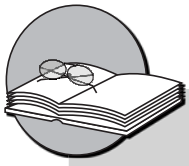
O fórum está aberto para discutirmos as dúvidas; participe, troque informações.
Também não perca o ***chat*** da semana. O horário e o dia serão colocados na página do fórum.

RESUMO

Vamos rever os principais conceitos abordados nesta aula.

- A **Programação Orientada a Objeto (POO)** tem como pilar teórico o conceito de **objeto**.
- Um Sistema Orientado a Objeto é um conjunto de objetos que representam os seres e coisas do mundo real, interagindo computacionalmente com as mesmas características e comportamentos reais.
- Classes são estruturas que definem as características e comportamentos dos seres ou coisas do mundo real; quando implementados em Java nas classes, essas características passam a se chamar atributos e os comportamentos se transformam nos nossos métodos.
- Objetos são as instâncias das classes, os seres que existem no meu mundo real quando são inseridos no mundo computacional.
- Atributos são as características inerentes ao meu ser ou coisa do mundo real quando são definidas no mundo computacional.
- Métodos são os comportamentos ou funções de um objeto do mundo real na forma como ele é tratado no mundo computacional.

- Um método é um trecho de código que realiza uma função específica e pode ser chamado por qualquer outro método ou classe para realizar a referida função num determinado contexto.
- **Casting** ou **typecasting** é o processo no qual fazemos a conversão de tipos de um atributo ou dado para outro tipo de dado. O **casting** pode ser implícito ou explícito.
- **Encapsulamento** é o conceito no qual protegemos os atributos de nossas classes através do modificador de acesso **private**; com isso, somente a própria classe pode acessá-lo.
- Classes externas podem interagir com a classe protegida (o que é desejável) por meio de **métodos acessores**, os métodos **get**, que são **public** e capturam o conteúdo de um atributo.
- Além disso, para definir os valores desses atributos, utilizamos **métodos de definição**, os métodos **set**, que também são **public** e assim alcançados por qualquer classe do sistema.



Informação sobre a próxima aula

Na próxima aula, continuaremos a estudar os conceitos de Orientação a Objeto com base em exemplos práticos que ajudarão a entender melhor esse paradigma que domina a programação de sistemas modernos.

AULA 6

Orientação a Objeto II

Meta da aula

Continuar a apresentar os conceitos de Orientação a Objeto em Java.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. criar e reutilizar suas próprias classes;
2. entender e utilizar sobrecarga de métodos;
3. organizar projeto de classes por meio de pacotes;
4. reconhecer os modificadores de acesso utilizados em Java;
5. reconhecer e aplicar os conceitos de Herança;
6. reconhecer e aplicar os conceitos de Polimorfismo;
7. reconhecer e aplicar os conceitos de Classes Abstratas;
8. reconhecer e aplicar os conceitos de Interfaces.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você: tenha um computador com Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; tenha entendido bem as aulas estudadas até agora e realizado com aproveitamento todas as atividades, já que entraremos numa nova fase do aprendizado; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

UTILIZANDO O *THIS* NAS SUAS CLASSES

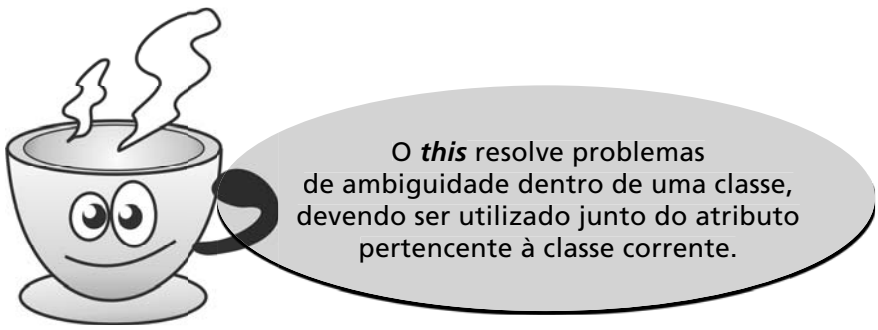
A palavra reservada ***this*** é utilizada para resolver ambiguidades que haja entre os elementos de uma classe em relação ao seu nome, não os confundindo com outros elementos que tenham vindo de outra classe que tenham o mesmo nome. Normalmente, é utilizado para acessar atributos ou métodos da classe ativa.

Vamos analisar no trecho de classe a seguir o método *setIdade*:

```
...
int idade;
public void setIdade(int idade) {
    idade = idade;
}

...
```

No código anterior, temos um atributo ***int idade*** pertencente à classe corrente e no parâmetro do método ***setIdade***(***int idade***), outro atributo ***idade***, sendo este último idade, o que será passado como parâmetro por outro objeto que utilizará o método da nossa classe. Assim, na próxima linha, onde temos *idade* = *idade*, está ocorrendo uma ambiguidade, que idade é de quem? Para resolver esse tipo de dúvida, o elemento pertencente à classe corrente recebe o prefixo *this* seguido de ponto antes do atributo. Agora o código fica da seguinte forma:

```
...
int idade;
public void setIdade(int idade) {
    this.idade = idade;
}

...
```



SOBRECARGA DE MÉTODOS

Algumas vezes, precisamos que métodos, por alguma questão de modelagem, tenham o mesmo nome, mas implementem comportamentos diferentes de acordo com o argumento que é passado. Quando isso acontece, dá-se o que chamamos de **sobrecarga de métodos**.

Métodos com o mesmo nome, mas com comportamentos diferentes de acordo com os argumentos passados, é o conceito chamado de **sobrecarga de métodos**. A sobrecarga pode ser de quantidade de argumentos (quantidades diferentes para métodos diferentes) ou de tipos de dados diferentes ou retorno de valores diferentes.

Relembre uma classe da aula anterior, a classe *RegistraAluno*. Nela, vamos criar dois métodos sobrecarregados por quantidade de parâmetros. Nosso primeiro método *imprimir*( ) não receberá parâmetro algum e imprimirá nome, endereço e idade do aluno:

```
public void imprimir(){
    System.out.println("Nome: "+nome);
    System.out.println("Endereço: "+endereco);
    System.out.println("Idade: "+idade);
}
```

No outro método sobrecarregado, ele recebe como parâmetro três argumentos *double* representando as três notas do aluno. A máquina virtual sabe interpretar em tempo de execução que esse é o método correto que foi chamado, se for o caso, imprimindo então as três notas além do nome do aluno:

```
public void imprimir(double mNota, double pNota, double gNota){
    System.out.println("Nome: "+nome);
    System.out.println("Nota Matemática: "+mNota);
    System.out.println("Nota Português: "+pNota);
    System.out.println("Nota Geografia: "+gNota);
}
```

E assim, podemos utilizar esses métodos por meio da classe principal *AppRegistraAluno* (programa, aplicação), que utiliza nossa classe auxiliar *RegistraAluno*, nossa classe que representa nosso objeto de negócio:

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

        RegistraAluno ana = new RegistraAluno();

        ana.setNome("Ana Machado");
        ana.setEndereco("Av. Brasil, 1000");
        ana.setIdade(15);

        ana.setNotaMatematica(9.7);
        ana.setNotaPortugues(8.2);
        ana.setNotaGeografia(7.4);

        ana.imprimir();

    }

}
```

Os dois métodos *imprimir* estão implementados em *RegistraAluno*, mas ao fazer a chamada para ele em AppRegistraAluno para o objeto ana, a JVM sabe que deve executar as ações do método *imprimir* sem parâmetros, e a saída será:

Vamos agora modificar *AppRegistraAluno* para imprimir também as notas. Isso será feito simplesmente passando as notas como argumento do método *imprimir*:

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

        RegistraAluno ana = new RegistraAluno();

        ana.setNome("Ana Machado");
        ana.setEndereco("Av. Brasil, 1000");
        ana.setIdade(15);

        ana.setNotaMatematica(9.7);
        ana.setNotaPortugues(8.2);
        ana.setNotaGeografia(7.4);

        ana.imprimir(ana.getNotaMatematica(),
            ana.getNotaPortugues(),
            ana.getNotaGeografia());
    }

}
```

E a saída será, conforme previsto, com a chamada do método *imprimir* mostrando as notas na tela, em vez do endereço e idade do aluno:

Atividade prática 1

Abra sua **IDE NetBeans** e implemente a classe **RegistraAluno** com os dois métodos sobrecarregados *imprimir*, como foi mostrado.

Após isso, crie duas classes **AppRegistraAluno** e **AppRegistraAluno2**. Na primeira classe, instancie um objeto que representa um aluno com seus respectivos dados e imprima essas informações utilizando o método *imprimir* sem parâmetros, que imprime apenas as informações pessoais do aluno. Na segunda classe, instancie um outro aluno com seus dados, mas dessa vez utilize o método imprimir que recebe como parâmetros as notas do aluno, imprimindo-as na saída do sistema junto com o nome do aluno.

PACOTES

Para organizar as classes e recursos de um projeto com o objetivo de facilitar a análise, o projeto e a manutenção do sistema, a Orientação a Objeto criou uma abstração lógica chamada **pacote**, cuja representação física é feita pelos diretórios do sistema. Assim, o gerenciamento e a manutenção do projeto ficam muito mais organizados e facilitados, com estrutura padronizada, evitando conflito de nomes e origens.

Para usar classes externas ao pacote que está sendo utilizado, devemos escrever a palavra reservada *import* seguida do nome do pacote mais "." e os subpacotes separados também por ".", terminando com a classe a ser utilizada:

```
import  <pacote>.<subpacote>.<classe>;
```

Por exemplo, para importar a classe *Color de java.awt* (*java* é o pacote e *awt* o subpacote), devemos escrever a seguinte linha na nossa classe:

```
import java.awt.Color;
```

As classes do pacote *java.lang* (que estudaremos futuramente com mais detalhes) são importadas pela JVM automaticamente, sem necessidade do *import*; por isso, não precisamos importar a classe *String* quando a utilizamos, pois ela pertence ao pacote *java.lang*.

Para criar seus próprios pacotes, você deve escrever na classe a palavra reservada *package* seguida da hierarquia de pacotes, ou seja, o pacote seguido dos subpacotes subsequentes separados por "." :

```
package  <pacote>.<subpacote>;
```

Quando um pacote é criado pela IDE e dentro desse pacote você cria uma classe, o comando *package* com o empacotamento correto é inserido automaticamente no código. Essa é a forma mais simples e correta de utilizar o conceito de pacotes.

MODIFICADORES DE ACESSO

Algumas vezes, é interessante implementar restrições quanto ao acesso aos elementos da classe. Esses níveis de restrições são implementados por meio do conceito de **modificadores de acesso**.

Existem quatro modificadores de acesso em Java: ***public***, ***private***, ***protected*** e ***default***; este último é acionado quando não se escreve nenhum dos outros modificadores de acesso antes das definições de classes, atributos ou métodos.

- Quando não é usado nenhum modificador de acesso (modificador ***default***), os elementos da classe são acessíveis somente pelos métodos internos da classe e das suas subclasses.
- Quando é utilizado o modificador de acesso ***public***, os elementos da classe são acessíveis tanto pela própria classe quanto por qualquer classe que tente acessá-los.
- Quando utilizado o modificador de acesso ***protected***, os elementos da classe são acessíveis somente por classes no mesmo pacote da classe dos elementos.
- Quando utilizado o modificador de acesso ***private***, os elementos da classe são acessíveis somente pela própria classe que os definiu.

HERANÇA

Herança é uma técnica da Orientação a Objeto cujo objetivo é criar um modelo com objetos do mundo real. Aqueles objetos que tem características mais genéricas são construídos de forma que os que são mais específicos possam receber os atributos e métodos dos genéricos sem a necessidade de reescrita de código, aumentando o reúso e facilitando a manutenção.

A ***herança*** em Java é definida pela palavra reservada ***extends*** utilizada pela classe que quer herdar as características. A essa classe damos o nome de ***subclasse***. A classe que está sendo utilizada na herança recebe o nome de ***superclasse***.

Veja o exemplo da classe ***Pessoa*** e depois da classe ***Aluno***, que herda de Pessoa:

```
public class Pessoa {
    protected String nome;
    protected String endereco;

    public String getNome(){
        return nome;
    }

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

    public String getEndereco(){
            return endereco;
        }

        public void setEndereco(String endereco){
            this.endereco = endereco;
    }
}
```

Na classe anterior, definimos dois atributos: ***nome*** e ***endereco***, e os métodos ***get***, para acessar, e ***set***, para definir cada um dos atributos.

Veja a classe ***Aluno***:

```
public class Aluno extends Pessoa {
    int nota;

    public int getNota(){
            return nota;
        }

        public void setNota(int nota){
            this.nota = nota;
    }
}
```

Na classe Aluno, o fato de termos o código ***extends Pessoa*** faz com que ela herde todos os atributos e métodos de ***Pessoa***; afinal, ***Aluno*** é um tipo de ***Pessoa***.

Vamos explicar como funciona a lógica de execução do mecanismo de herança por intermédio da classe a seguir:

```
public class TesteAluno {
    public static void main(String args[]){
        Aluno ana = new Aluno();
        ana.setNome("Ana");
        String nomeAtual;
        nomeAtual = ana.getNome();
        System.out.println(nomeAtual);
    }
}
```

Quando a classe anterior instancia ***Aluno***, o construtor de ***Pessoa*** é invocado e os atributos deste são criados para poderem ser utilizados por ***Aluno***, que herda de ***Pessoa***. Veja que utilizamos os métodos ***setNome( )*** e ***getNome( )***, que não são de ***Aluno*** mas de ***Pessoa***; com isso, não precisamos criar esses métodos em ***Aluno***, pois estamos reutilizando-os através de ***Pessoa***.

Atividade prática 2

Abra sua **IDE NetBeans** e crie dois pacotes, um chamado **cedjava.app** e outro chamado **cedjava.objetos**.

No pacote **cedjava.objetos,** implemente a classe **Pessoa** e depois a classe **Aluno** herdando os dados de **Pessoa**.

Por fim, no pacote **cedjava.app** implemente a classe **TesteAluno** para testar se a herança funcionou corretamente.

Caso você não consiga fazer ou tenha alguma dúvida em qualquer uma das atividades, coloque sua dúvida ou seu problema no Fórum, insira a dúvida além do código que está tentando implementar e, se estiver dando algum erro, faça uma cópia da tela numa figura e coloque no Fórum. Com certeza algum colega ou o tutor irá ajudar a esclarecer o problema.

Uma observação importante: quanto mais você praticar ou modificar os exemplos que estamos propondo nas aulas, inclusive tentando criar seus próprios exemplos, mais intimidade e facilidade você terá com a linguagem e principalmente com o manuseio das ferramentas utilizadas para programar, com foco principal na IDE NetBeans. Por isso, não deixe de praticar sempre que puder.

POLIMORFISMO

Vamos voltar ao exemplo anterior, no qual temos uma superclasse ***Pessoa*** e uma subclasse ***Aluno*** herdando de ***Pessoa***. Suponha que tenhamos também uma nova subclasse herdando de ***Pessoa***, a classe ***Empregado***.

Suponha agora que as três classes tenham um método ***getNome( )***; isso pode ocorrer, visto que eu posso querer que em um determinado momento, no meu sistema, o método a ser chamado seja o ***getNome( )*** de alguma das subclasses, pois eles podem executar comportamentos diferentes. Isso é o que chamamos de ***sobrescrita de métodos***. Na execução, a Máquina Virtual Java sempre procura pelos métodos na classe original (subclasse). Caso não os encontre, ela procura na superclasse pelo método com aquele nome referido.

Veja as classes **Pessoa, Aluno** e **Empregado** utilizando o conceito de sobrescrita de métodos:

Classe Pessoa

```
public class Pessoa {
    protected String nome;

    public String getNome(){
        System.out.println("Nome da Pessoa: "+nome);
        return nome;
    }

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

}
```

Classe Aluno

```
public class Aluno extends Pessoa {

    public String getNome(){
        System.out.println("Nome do Aluno: "+nome);
        return nome;
    }

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

Classe Empregado

```
public class Empregado extends Pessoa {

    public String getNome(){
        System.out.println("Nome do Empregado: "+nome);
        return nome;
    }

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

Nesse caso, para cada objeto que for utilizado será impresso o nome do objeto com sua categoria (Pessoa, Aluno ou Empregado), como você pode observar nesta classe:

```
public class TesteAluno2 {
    public static void main(String args[]){
            Pessoa joao = new Pessoa();
            joao.setNome("Joao");
            joao.getNome();

            Aluno ana = new Aluno();
            ana.setNome("Ana");
            ana.getNome();

            Empregado mario = new Empregado();
            mario.setNome("Mario");
            mario.getNome();

    }
}
```

Atividade prática 3

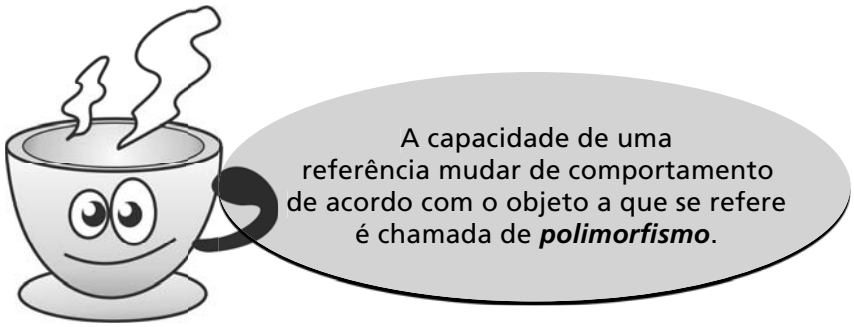
Através do ***NetBeans***, no pacote ***cedjava.objetos,*** modifique a classe ***Pessoa*** e depois a classe Aluno, além de criar a nova classe ***Empregado***, conforme já foi mostrado.
No pacote ***cedjava.app,*** implemente a classe ***TesteAluno2*** para testar e entender o conceito de ***sobrescrita de métodos***.

Veja agora outro conceito importante modificando um pouco nossa classe ***TesteAluno2***, que foi utilizada para visualizar a sobrescrita de nossos objetos. Para isso, iremos modificá-la um pouco.

Vamos criar um atributo do tipo Pessoa, mas sem chamar o construtor com ***new***. Em vez disso, vamos atribuir a referência de ***Pessoa*** aos objetos ***Aluno*** e ***Empregado***:

```
public class TesteAluno3 {

    public static void main(String[] args) {
        Pessoa ref;

        Aluno alu = new Aluno();
        alu.setNome("Beatriz");

        Empregado emp = new Empregado();
        emp.setNome("Carlos");

        ref = alu;
        ref.getNome();

        ref = emp;
        ref.getNome();
    }

}
```

Apesar de o atributo ref ser do tipo ***Pessoa***, ao ser impresso o nome por meio dos métodos ***getNome( )***, vimos que a referência ***ref*** do tipo ***Pessoa*** aponta para um objeto do tipo ***Aluno*** e depois para um objeto do tipo ***Empregado***, assumindo assim o comportamento destes. Isso é o que chamamos de ***polimorfismo***.



CLASSES ABSTRATAS

Vamos supor que estamos modelando uma hierarquia de um sistema para implementação de animais cuja linha de herança tenha no seu topo a classe mais genérica (a superclasse) SerVivo. Nessa classe, vamos definir alguns métodos comuns aos seres vivos, como respirar, dormir e andar. Mas temos de tomar cuidado ao generalizar todos os métodos de seres vivos, pois existem comportamentos que são executados de forma diferente conforme a espécie. Além disso, SerVivo nunca será objeto, já que um tipo de ser vivo deve estar associado a uma espécie; por isso, devemos tomar algum cuidado para que um programador desatento não instancie SerVivo. A forma de implementar esses conceitos com esses cuidados é por meio do que chamamos ***Classes Abstratas***.

Uma classe abstrata é uma classe que não pode ser instanciada como objeto, mas serve como base para criação de outras classes que irão herdar dela. Geralmente, a classe abstrata está no topo de uma hierarquia de classes como sua superclasse.

Para definir que uma classe é abstrata, devemos colocar a palavra reservada ***abstract*** antes de ***class*** na sua definição. Além disso, vimos que alguns métodos das classes abstratas não devem ser implementados, pois podem ter comportamentos diferentes de acordo com as subclasses. Esses são os ***métodos abstratos***, e devem vir igualmente com a palavra ***abstract*** após o modificador de acesso, e são de implementação obrigatória por quem herdar da classe abstrata.

Veja este o exemplo, no qual criamos a superclasse abstrata SerVivo e duas subclasses concretas a partir dessa classe: Cachorro e Humano:

Classe SerVivo

```
public abstract class SerVivo {
    public abstract void andar();
}
```

Classe Humano

```
public class Humano extends SerVivo {
    public void andar(){
        System.out.println("Ser humano andando com 2 pernas...");
}
```

Classe Cachorro

```
public class Cachorro extends SerVivo {
    public void andar(){
        System.out.println("Cachorro andando com 4 patas...");
}
```

Como explicamos, temos a superclasse ***SerVivo*** e duas subclasses que herdam desta, as classes ***Humano*** e ***Cachorro***. Porém, a classe ***SerVivo*** é abstrata; isso faz com que ela não possa ser instanciada por uma aplicação. Além disso, ela possui um método abstrato, sem implementação, somente sua assinatura, isso faz com que as duas classes que herdam de ***SerVivo*** sejam obrigadas a implementar o método abstrato de ***SerVivo***, conforme está feito nas classes anteriores.

INTERFACES

Interfaces são classes especiais; por uma decisão de arquitetura e modelagem, foi definido que tais classes teriam somente métodos abstratos.

Interfaces definem um padrão e o caminho público para especificação do comportamento de classes. Permitem que classes, indepen-

dentemente de sua localização na estrutura hierárquica, implementem comportamentos comuns.

Interfaces são utilizadas quando queremos obrigar a implementação de métodos que não têm relacionamento de herança entre as classes que irão utilizá-los.

A principal diferença entre uma interface e uma classe abstrata é que a classe abstrata pode possuir métodos implementados (reais) ou não implementados (abstratos). Na interface, todos os métodos são abstratos e públicos, tanto que, para esta, a palavra-chave *abstract* ou *public* é opcional.

Suponha uma classe Linha que contém métodos que calculam o tamanho da linha e compara essa linha com outros objetos do tipo Linha, indicando sua relação de grandeza. Imagine outro objeto MeusNumeros que compara atributos dessa classe com outros objetos MeusNumeros. Apesar de diferentes, essas classes implementam métodos semelhantes quanto à assinatura. Para garantir que esses métodos sejam implementados e para padronizar sua assinatura, utilizamos interfaces.

A relação entre uma classe e a interface que ela implementa é como a de um contrato que deve ser assinado por quem quer implementar determinado comportamento. A leitura que se faz é a seguinte: "A classe que quiser realizar tal função deve assinar tal contrato (implementar a interface) e, consequentemente, implementar seus métodos abstratos".

Veja os exemplos da classe Linha e da interface Relação que define os métodos abstratos a serem implementados.

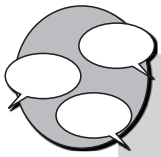
Na interface definimos apenas as assinaturas dos métodos que devem ser implementados por todas as classes que querem executar tal comportamento. Uma interface deve ter a palavra reservada *interface* na sua definição, em vez de *class*, como você vê abaixo:

```
interface Relacao {
    boolean MaiorQue(Object a, Object b);
    boolean MenorQue(Object a, Object b);
    boolean Igual(Object a, Object b);
}
```

Por sua vez, a classe que quer fazer relação de grandeza com objetos deve assinar um contrato com a interface Relacao (*implements* Relacao) e implementar obrigatoriamente todos os seus métodos, como está na classe Linha:

```
public class Linha implements Relacao {
    private double x1;
    private double x2;
    private double y1;
    private double y2;

    public Linha(double x1, double x2, double y1, double y2) {
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }

    public double getComprimento(){
        double comprimento = Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
        return comprimento;
    }

    public boolean MaiorQue(Object a, Object b) {
        double aComp = ((Linha)a).getComprimento();
        double bComp = ((Linha)b).getComprimento();
        return (aComp > bComp);
    }

    public boolean MenorQue() {
        double aComp = ((Linha)a).getComprimento();
        double bComp = ((Linha)b).getComprimento();
        return (aComp < bComp);
    }

    public boolean Igual() {
        double aComp = ((Linha)a).getComprimento();
        double bComp = ((Linha)b).getComprimento();
        return (aComp == bComp);
    }
```

Uma observação importante é que interface é a forma que o Java tem de criar uma abstração para implementar herança múltipla, já que diretamente, por herança via *extends*, isso não é permitido em Java, o que faz todo sentido, pois dessa forma se criaria um acoplamento muito grande entre as classes herdadas; com interface, isso é amenizado. Resumindo, uma subclasse pode fazer *extends* com apenas uma classe, mas pode fazer *implements* com muitas interfaces.



INFORMAÇÕES SOBRE FÓRUM

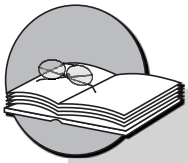
O fórum é o espaço para discutirmos suas dúvidas. Participe, troque informações.
Também não perca o ***chat*** da semana. O horário e o dia serão colocados na página do fórum.

RESUMO

Reveja os principais conceitos apresentados nesta aula.

- A palavra reservada ***this*** deve ser utilizada para resolver problemas de ambiguidade dentro de uma classe, devendo ser utilizado junto ao atributo pertencente à classe corrente.
- Quando precisamos que métodos, por alguma questão de modelagem, tenham o mesmo nome mas implementem comportamentos diferentes de acordo com o argumento que é passado, precisamos utilizar o que chamamos ***sobrecarga de métodos***.
- Métodos com o mesmo nome, mas com comportamentos diferentes de acordo com os argumentos passados é o conceito chamado ***sobrecarga de métodos***. A sobrecarga pode ser de quantidade de argumentos (quantidades diferentes para métodos diferentes) ou de tipos de dados diferentes ou retorno de valores diferentes.
- Para organizar as classes e recursos de um projeto com o objetivo de facilitar a análise, o projeto e a manutenção do sistema, a Orientação a Objetos criou uma abstração lógica chamada ***pacote***, cuja representação física é feita através dos diretórios do sistema. Assim, o gerenciamento e a manutenção do projeto ficam muito mais organizados e facilitados com uma estrutura padronizada, evitando conflito de nomes e origens.
- Quando queremos implementar restrições nos recursos de nossas classes, precisamos utilizar os chamados modificadores de acesso. Existem quatro em Java: ***public***, ***private***, ***protected*** e ***default***; o ***default*** é quando não se escreve nenhum dos outros modificadores de acesso antes das definições de classes, atributos ou métodos.

- ***Herança*** é uma técnica da Orientação a Objeto cujo objetivo é criar um modelo com objetos do mundo real. Aqueles objetos que tem características mais genéricas são construídos de forma que os que são mais específicos possam receber os atributos e métodos dos genéricos sem a necessidade de reescrita de código, aumentando o reúso e facilitando a manutenção. A ***herança*** em Java é definida pela palavra reservada ***extends*** utilizada pela classe que quer herdar as características. A essa classe damos o nome de ***subclasse***. A classe que está sendo utilizada na herança recebe o nome de ***superclasse***.



Informação sobre a próxima aula

Na próxima aula, iremos estudar como é o tratamento de erros em Java, através de um conceito bem interessante e diferente do que conhecemos nas linguagens procedurais, as ***exceções***. Até lá...

AULA 7

Tratamento de exceções em Java

Meta da aula

Ensinar a forma de tratamento de erros em Java, feita por intermédio da classe *Exceptions* (exceções) e suas subclasses.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. definir o que são exceções;
2. explicar o mecanismo de tratamento de erros em Java por meio de exceções;
3. empregar tratamento de exceções utilizando estruturas com *try*, *catch* e *finally*.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; tenha compreendido bem todos os conceitos de Orientação a Objeto aprendidos nas duas últimas aulas, importantes para continuarmos o aprendizado de Java; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

O QUE SÃO EXCEÇÕES?

Exceções em Java têm o mesmo sentido semântico da palavra em português, ou seja, uma exceção acontece quando a regra não é respeitada, seguida; a regra, nesse caso, é o fluxo normal (fluxo principal) do programa. Portanto, ocorre uma exceção sempre que algo der errado ou fugir do fluxo esperado na nossa aplicação, por isso o nome exceção para tratamento de erro.

O **tratamento de exceções**, como também é chamada essa técnica em Java (termo originado de tratamento de erros), é bem diferente do que fazíamos quando nossa programação não era Orientada a Objeto. Antes, o que fazíamos para tratar um fluxo fora do normal era tratar o retorno do método que executava esse fluxo. Isso era feito por meio de uma **marcação booleana**, em que o método retornava *true* se fosse executado, quando então a aplicação imprimia uma mensagem de "método executado com sucesso" ou algo parecido, e retornava *false* se ocorresse algum problema, caso em que a aplicação imprimia uma mensagem de que algum problema havia ocorrido.

Essa forma de tratamento de erro tem vários problemas, e entre eles o mais criticado é que se consegue tratar apenas duas possibilidades: ou o fluxo funcionou ou não funcionou. Além disso, nesse caso nunca se poderá definir a causa exata, pois um determinado fluxo pode dar errado por várias razões.

O que podemos fazer para resolver isso?

Alguns programadores podem ficar tentados a ter a "maravilhosa ideia" de, no lugar de retornar um *booleano*, retornar um inteiro e tratá-lo num bloco *if-else-if* e para cada inteiro retornar uma mensagem diferente relativa ao erro.

Isso está correto? **NÃO**... isso está errado, isso é uma má prática de programação (*anti-pattern*) chamada *Magic Numbers*. Nela, o programador cria números mágicos para cada erro. Esse tipo de prática tem manutenção extremamente complicada, pode incorrer numa quantidade grande de erros e não dá semântica alguma ao código, que precisa retratar e explicar por si só o que está ocorrendo no mundo real por meio do nome da classe, dos seus atributos e dos seus métodos. Esse tipo de técnica, com "os números mágicos", vai contra os princípios da Orientação a Objeto, pois precisaremos ficar com uma tabela na mesa para sabermos que erro foi aquele que ocorreu.

Vou dar um exemplo. Imagine que você está implementando um sistema bancário no qual temos uma classe **Conta** que possui um **método** sacar que recebe o valor do saque, executa as operações de saque (conexão com banco, validação de usuário, verificação de saldo e limite etc.) e retorna *true* ou *false* conforme o caso:

```
public class Conta {
    private double saldo;
    private long numConta;
    ...
    public boolean sacar(double valor) {
    ...
```

Se quisermos fazer o tratamento dos possíveis fluxos que possam ocorrer por ocasião do saque, só conseguiremos tratar dois casos: ou o saque funcionou ou não funcionou:

```
public class TestaSaque {
    public static void main(String args[]){
        Conta c1 = new Conta();
        boolean resultadoDoSaque;
        resultadoDoSaque = c1.sacar(1000.0);
        if (resultadoDoSaque){
            System.out.println("Ok, seu saque deu certo");
        } else {
            System.out.println("Ops, o saque não foi possível");
        }
    }
}
```

Então, acho que você já se convenceu de que esse tipo de tratamento não é uma boa ideia.

A OO desenvolveu formas mais elegantes de tratar os fluxos anormais do sistema. Em Java, esse fluxo anormal se chama ***Exceptions*** (exceções), e a técnica utilizada para trabalhar erros de forma correta é o chamado ***Tratamento de Exceções***.

No tratamento de exceções, a ideia é que você saiba quais tipos de erro um determinado trecho de código pode lançar, e então ele tem a obrigação de identificar esse erro caso ele ocorra, enviando uma mensagem adequada e tratada para o usuário, nesse caso. Parece difícil, quer dizer que é preciso conhecer todas as exceções do Java? Isso não fica muito complicado, pois, uma vez que a exceção esteja contida no código, a própria JVM se responsabiliza por desviar a execução para esse tratamento adequado, caso ele ocorra.

Caso a JVM não consiga encontrar o devido tratamento para aquela exceção que foi lançada (ou seja, você esqueceu ou não percebeu que poderia ocorrer aquele erro naquela classe), ela interrompe o fluxo de execução e mostra o erro na saída do sistema. Isso não parece muito adequado, pois o usuário não vai entender nada. É melhor você mesmo interceptar e tratar esse erro. Na verdade, essa interrupção de execução da JVM é uma proteção para garantir que seu programa não vá executar uma operação errada, num fluxo que não é o normal, como já dissemos.

Então, para evitar que a JVM "morra" devido a uma exceção não tratada, devemos tentar (***try***) executar o trecho de código suscetível ao erro e, caso não consigamos, devemos pegá-lo (***catch***), discriminando qual foi o erro.

A palavra reservada inserida num bloco para tentar executar o código é ***try*** (tentar). Para pegar a exceção e discriminá-la a palavra reservada utilizada é o ***catch*** (pegar).

Se uma determinada exceção foi pega e identificada (ou seja, era esperada), a execução continua, já que a JVM supõe que você tomou as providências cabíveis.

Na verdade, ***Exception*** é uma classe que tem muitas "filhas", como pode ser observado na **Figura 7.1**.

Vamos dividir as subclasses filhas de ***Exception*** em dois grupos: de um lado, ***RuntimeException*** e suas subclasses; no outro, todas as demais subclasses de ***Exception***.

Todas as exceções filhas de ***RuntimeException*** são chamadas ***Unchecked Exceptions***; seu tratamento não é obrigatório, uma vez que você deve resolver esse tipo de erro no código.

Exemplos de exceções de Runtime:

- ArithmeticException;
- NumberFormatException;
- NullPointerException;
- ArrayIndexOutOfBoundsException.

Todas as exceções que não são filhas de ***RuntimeException*** (Unchecked Exceptions) são chamadas ***Checked Exceptions***. Nesse caso, temos duas opções: tratá-las num bloco ***try/catch*** ou lançá-las para serem tratadas à frente (numa outra classe que chamar essa que tem o erro), por meio de ***throws***.

Veja a hierarquia de classes ***Exception***:

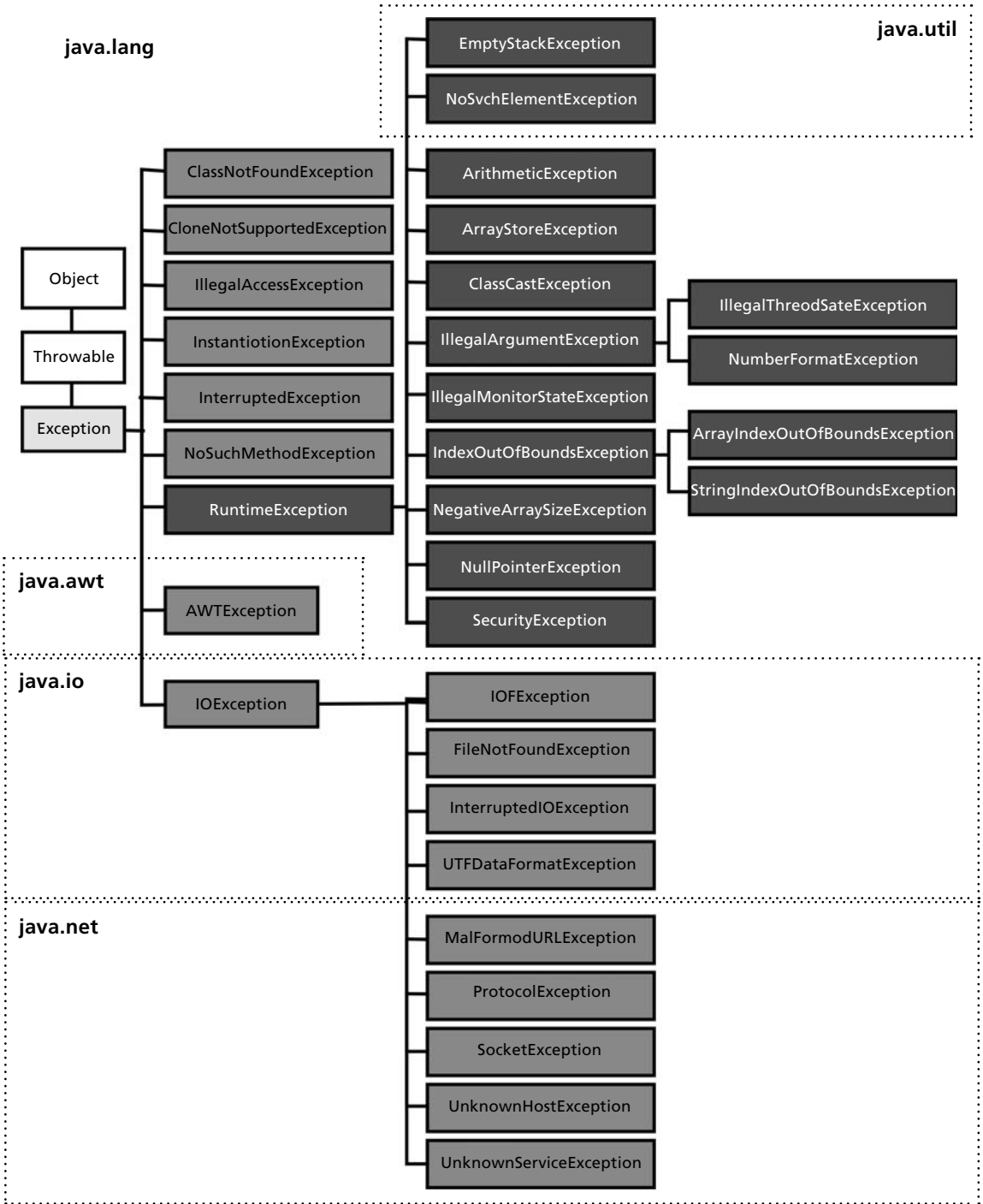


Figura 7.1: Hierarquia das classes a partir de Exception.

Veja a classe a seguir. Ela tem algum problema? Por quê? Como resolver?

```
public class TesteException {
    public static void main(String args[]){
        new java.io.FileReader("arquivoDeTeste.txt");
    }
}
```

No caso dessa classe, estamos utilizando a classe ***FileReader*** de ***java.io*** para fazer a leitura de um arquivo que está num determinado diretório; assim, você deve esperar, por exemplo, um erro tal que o arquivo não possa ser encontrado, o que é um tipo de ***IOException***, do tipo ***FileNotFoundException***, que não é filha de ***RunTimeException***. É, assim, uma ***Checked Exception***, e você é obrigado a tratá-la, caso contrário a classe não irá nem compilar, lançando qual exceção levou ao erro: ***unreported exception java.io.FileNotFoundException***; ***must be caught or declared to be thrown*** (erro apresentado pelo compilador).

Veja como é fácil perceber qual exceção deixou de ser tratada: o compilador mostra na sua janela de erro (com o tempo e a experiência, você aprenderá sobre elas).

Outro detalhe que merece atenção: já que uma classe pode fazer várias coisas, podemos ter mais de uma exceção e de repente todas *checked*.

Assim, podemos encadear tratamentos de exceção, mas cada um em um único bloco ***catch***, já que as medidas podem ser diferentes para cada erro e os erros ocorrerão um de cada vez.

```
public class TesteException2{
    public static void main(String args[]){
        System.out.println("vários catches...");
        try{
            //métodos que podem lançar várias exceptions...
        } catch(IllegalAccessException e){
            //...
        } catch(SQLException e){
            //...
        } catch(IOException e){
            //...
        }
    }
}
```

Vários erros também podem ser passados para a frente (nos dois casos deve ser dado ***import*** das classes referentes aos erros), mas separados por vírgula.

```
public class TesteException5{
    public static void main(String args[]) throws
    IllegalAccessException,SQLException,IOException {
        System.out.println("vários catches...");
        //nesse caso a classe preferiu passar
        //o tratamento das checked exceptions
        //para quem chamá-la
    }
}
```

TRATANDO AS EXCEÇÕES

Veja agora um exemplo de aplicação utilizando esses conceitos de exceções e como devemos tratá-los.

```
public class TesteException2 {
    public static void main(String args[]){
        try {
            double num1 = 20;
            double num2 = 25;
            System.out.println("Soma = "+(num1+num2));
            System.out.println("Subtração = "+(num1-num2));
            System.out.println("Multiplicação = "+(num1*num2));
            System.out.println("Divisão = "+(num1/num2));
        }

        catch(ArithmeticException e){
            System.out.println("Erro de divisão por zero!");
        }

        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Número de argumentos inválidos!");
        }

        catch(NumberFormatException e){
                    System.out.println("Entre somente com números!");
        }
    }
}
```

Na classe anterior, utilizamos o conceito de exceções para tratar alguns problemas que poderiam ocorrer no nosso sistema. Por exemplo, imagine que alguém insira um valor zero na variável numérica **num2**; quando for efetuada a divisão, estará ocorrendo uma divisão por zero, o que não é possível, matematicamente; com isso, podemos tentar capturar um erro de ***ArithmeticException***, como foi feito no primeiro bloco *catch*.

Outra exceção tratada foi a ***ArrayIndexOutOfBoundsException***, que ocorre quando inserimos mais argumentos do que uma operação ou espaço alocado suporta. No nosso caso, suponha que alguém queira realizar uma de nossas operações, que foi programada para ocorrer sempre duas a duas, com apenas um argumento. Isso não iria funcionar; por isso, no exemplo resolvemos tratar essa exceção.

No último bloco ***catch***, tratamos um erro que poderia ocorrer caso o usuário inserisse algum outro argumento diferente de números; isso lançaria a exceção ***NumberFormatException***.

Até aí tudo bem, mas se você olhar com mais cuidado a nossa figura que mostra a família de exceções, perceberá que ***ArithmeticException, ArrayIndexOutOfBoundsException*** e ***NumberFormatException*** são filhas de ***RunTimeException***, ou seja, são ***Unchecked Exception***. Sendo assim, não precisam ser tratadas.

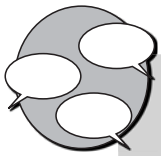
Pode surgir aí uma dúvida: por que existem exceções que devem ser tratadas e outras que não são obrigatórias?

Na verdade, tem a ver com o sentido semântico do erro e sua correção na regra implementada. Veja que faz muito mais sentido tratar uma exceção aritmética (***ArithmeticException***), de limitação de espaço ou de argumentos (***ArrayIndexOutOfBoundsException***) ou de formatação de tipo de dado (***NumberFormatException***) no próprio código a partir de blocos ***if-else*** do que deixar para a estrutura da JVM tratar esse erro. Afinal, são erros de dados do negócio, todos eles estão contidos nas ***RunTimeException***. Esses tipos de possíveis erros são muito mais bem tratados por meio de validações do que de exceções (que interrompem a **JVM**!), pois você quer que o usuário acerte.

Por outro lado, faz mais sentido deixar por conta da **JVM** os erros que não são ***RunTimeException***; você faz apenas a interceptação desses erros, pois são associados à **infraestrutura** do sistema; são as **Checked Exceptions**. Seria, por exemplo, um erro na rede, um problema com a conexão de banco de dados ou outro erro que não faz parte do erro do usuário.

Na classe anterior, poderíamos então implementar a mesma classe, mas sem a estrutura ***try/catch***; o compilador não reclamaria e o programa funcionaria sem problemas:

```
public class TesteException3 {
    public static void main(String args[]){

            double num1 = 20;
            double num2 = 25;
            System.out.println("Soma = "+(num1+num2));
            System.out.println("Subtração = "+(num1-num2));
            System.out.println("Multiplicação = "+(num1*num2));
            System.out.println("Divisão = "+(num1/num2));

    }
}
```



INFORMAÇÕES SOBRE FÓRUM

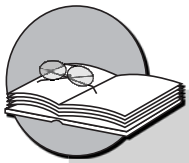
Entre no fórum e apresente suas dúvidas. Participe, troque informações. Quem sabe você não pode colaborar com seus colegas?
Também não perca o ***chat*** da semana. Veja o horário e dia na página do fórum.

RESUMO

Reveja os principais conceitos estudados nesta aula.

- Nas linguagens procedurais, o que fazíamos para tratar um fluxo fora do normal era tratar o retorno do método que executava esse fluxo. Isso era feito por meio de uma **marcação booleana** em que o método retornava *true* se fosse executado, quando a aplicação chamadora imprimia uma mensagem de "método executado com sucesso", ou algo parecido, ou retornava *false* se ocorresse algum problema, caso em que a aplicação chamadora imprimia uma mensagem de que algum problema havia ocorrido.
- A Orientação a Objeto desenvolveu formas mais elegantes de tratar os fluxos anormais do sistema. Em Java, esse fluxo anormal se chama ***Exceptions*** (exceções), e a técnica utilizada para trabalhar erros de forma correta é o chamado **Tratamento de Exceções**.

- No **Tratamento de Exceções**, caso a JVM não consiga encontrar o devido tratamento para aquela exceção que foi lançada (ou seja, você esqueceu ou não percebeu que poderia ocorrer aquele erro naquela classe), ela interrompe o fluxo de execução e mostra o erro na saída do sistema, o que não parece muito adequado, pois o usuário não vai entender nada; é melhor você mesmo interceptar e tratar esse erro.
- Para evitar que a JVM "morra" devido a uma exceção não tratada, devemos tentar (***try***) executar o trecho de código suscetível ao erro; caso não consigamos, devemos pegá-lo (***catch***), discriminando qual foi o erro. As palavras reservadas entre as quais o código é inserido num bloco para tentar executar o código são ***try*** (tentar) e, para pegar a exceção e discriminá-la, ***catch*** (pegar).
- As subclasses filhas de ***Exception*** são divididas em dois grupos: ***RuntimeException*** e suas subclasses num grupo; e todas as demais subclasses de ***Exception*** no outro grupo.
- Todas as exceções filhas de ***RuntimeException*** são chamadas de ***Unchecked Exceptions***; seu tratamento não é obrigatório, uma vez que você deve resolver esse tipo de erro no código; são exceções do negócio implementado.
- Todas as exceções que não são filhas de ***RuntimeException*** (***Unchecked Exceptions***) são as chamadas ***Checked Exceptions***. Neste caso, temos duas opções: tratar as exceções com um bloco ***try/catch*** ou lançá-las para serem tratadas à frente pela classe que as utilizou por meio de ***throws***; isso é feito para erros do sistema, de infraestrutura, como um erro na rede, um problema com a conexão de banco de dados ou outro erro que não faz parte do erro do usuário.



Informação sobre a próxima aula

Na próxima aula, você irá estudar como trabalhar com algumas das principais bibliotecas de Java incluídas no JSE; vamos utilizar algumas de suas classes para ajudar na nossa programação do dia a dia. Até lá...

Bibliotecas do Java Standard Edition (JSE)

AULA 8

Meta da aula

Ensinar a utilização de bibliotecas em Java por meio de algumas das principais bibliotecas implementadas no Java Standard Edition.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. entender o funcionamento de bibliotecas em linguagens orientadas a objeto;
2. explicar como utilizar bibliotecas de classes em Java;
3. utilizar as principais bibliotecas do Java Standard Edition para auxiliar na criação de suas classes.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você tenha um computador com o Windows XP ou Windows Vista instalado e pelo menos 512 MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; tenha compreendido bem todos os conceitos de orientação a objetos aprendidos nas últimas aulas, importantes para continuarmos o aprendizado de Java; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

INTRODUÇÃO

Uma das principais premissas da programação orientada a objeto é a reutilização de classes que já estejam prontas para serem utilizadas.

E assim, como não podia ser diferente, como *core* da plataforma e de qualquer sistema escrito em Java, o Java Standard Edition possui mais de 3.500 classes para serem reutilizadas e auxiliar nas mais diversas tarefas do dia a dia do programador. Imagine se você tivesse de começar cada função, cada tarefa, cada funcionalidade do seu sistema do zero. Seria impossível...

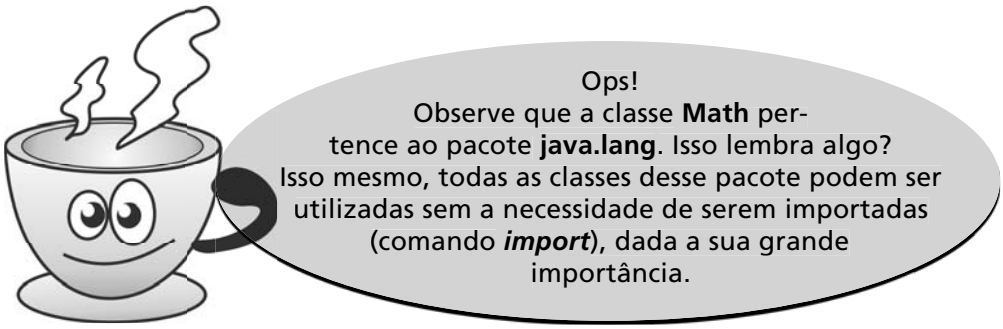
Dentro dessas milhares de classes, existe um conjunto tão básico a ser utilizado que você não precisa nem fazer o **import** dessas classes, – ele é feito automaticamente. São as classes do pacote **java.lang**. Nesse pacote, as duas classes mais importantes para qualquer nível de desenvolvedor, desde o iniciante até o mais experiente, são com certeza **Math** e **String**.

Vamos, no decorrer desta aula, explicar um pouco mais sobre essas classes e naturalmente você entenderá por que elas são tão importantes. Além disso, vamos também falar de outras classes complementares que o programador Java pode querer utilizar.

JAVA E OPERAÇÕES MATEMÁTICAS COM java.lang.Math

Invariavelmente o programador de qualquer linguagem precisa manipular números e efetuar operações matemáticas. Algumas dessas operações dariam muito mais trabalho se não tivéssemos métodos prontos para utilizar.

Em Java, a classe que implementa essas funcionalidades e é responsável por auxiliar nas operações com números é a classe **Math** do pacote **java.lang**



Vamos ver um raio x dos principais métodos de **Math**:

- **public static double abs(double a):**

✓ recebe como parâmetro um valor numérico do tipo double e retorna seu valor absoluto (sem sinal).

- **public static double random():**

✓ retorna um valor numérico do tipo double maior ou igual a 0.0 e menor que 1.0. Utilizado para geração de números aleatórios.

- **public static double max(double a, double b):**

✓ recebe como parâmetro dois valores numéricos do tipo double e retorna o maior dos dois.

- **public static double min(double a, double b):**

✓ recebe como parâmetro dois valores numéricos do tipo double e retorna o menor dos dois.

- **public static double ceil(double a):**

✓ é a chamada função teto. Recebe como parâmetro um valor numérico do tipo double e retorna o inteiro mais próximo maior ou igual a esse número, mas convertido para double.

- **public static double floor(double a):**

✓ é a chamada função piso. Recebe como parâmetro um valor numérico do tipo double e retorna o inteiro mais próximo menor ou igual a esse número, mas convertido para double.

- **public static double exp(double a):**

✓ retorna a constante de Euler elevada à potência passada como parâmetro, devendo esse número ser um valor do tipo double.

- **public static double log(double a):**

✓ retorna o logaritmo natural de um numérico do tipo double.

- **public static double pow(double a, double b):**

✓ retorna um numérico do tipo double resultado do primeiro argumento (numérico do tipo double) elevado à potência do segundo argumento (também do tipo double).

- **public static long round(double a):**
  - ✓ retorna o número inteiro mais próximo do argumento numérico do tipo double que foi passado.
- **public static double sqrt(double a):**
  - ✓ retorna um numérico do tipo double cujo valor é a raiz quadrada do numérico double passado como argumento.
- **public static double sin(double a):**
  - ✓ retorna um valor numérico do tipo double cujo valor é o seno trigonométrico do ângulo passado como argumento.
- **public static double cos(double a):**
  - ✓ retorna um valor numérico do tipo double cujo valor é o cosseno trigonométrico do ângulo passado como argumento.
- **public static double toDegrees(double angRad):**
  - ✓ converte o valor de um ângulo medido em radianos para seu valor equivalente em graus.
- **public static double toRadians(double angDeg):**
  - ✓ converte o valor de um ângulo medido em graus para seu valor equivalente em radianos.

JAVADOCS

Formato de documentação oficial implementado pelo Java, mostrando as funcionalidades de uma classe por meio de páginas navegáveis (html), sendo acessíveis todas as informações relativas a seus métodos, campos e demais informações que o criador da classe ou API achou importante que ficasse documentado para quem fosse utilizá-la.

Como dito no começo, esses são os principais métodos da classe Math, mas existem outros que podem ser úteis para o seu trabalho caso não encontre a funcionalidade matemática desejada nesse conjunto que foi apresentado.

Atividade prática 1

Vamos pesquisar um pouco?

Acesse no site da Sun os JAVADOCS da classe **Math** no pacote **java.lang** e aprenda um pouco mais sobre essa importante ferramenta na hora de trabalharmos com programação matemática. Veja os outros métodos, constantes e demais informações sobre essa classe. Acesse em: http://java.sun.com/javase/6/docs/api/java/lang/Math.html

JAVA E MANIPULAÇÃO TEXTUAL COM java.lang.String

Como vimos desde o início, uma sequência de caracteres alfabéticos em Java é uma classe chamada *String*.

Dessa forma, as variáveis desse tipo armazenam parâmetros como referências a objetos e não um valor definido como nos tipos primitivos.

Atividade prática 2

Para entender melhor a diferença de acesso por referência e por valor em Java, o que ocorre no caso de *String*, implemente o código da classe **TestaString** a seguir e observe atentamente qual será a saída do programa.

```
1  public class TestaString {
2      public static void main(String args[]){
3          String a = new String("Java na Universidade");
4          String b = new String("Java na Universidade");
5          if (a == b) {
6              System.out.println("a e b sao objetos iguais (mesmo objeto)");
7          } else {
8              System.out.println("a e b sao objetos diferentes");
9          }
10     }
11 }
```

Como pode ser observado na classe da Atividade 2, mesmo que duas variáveis tenham o mesmo conteúdo, ao compará-las com o operador de igualdade "==", o resultado é que elas são diferentes, o que faz sentido, já que são objetos diferentes, ou seja, com referências diferentes. O mesmo não ocorre com variáveis primitivas, nas quais a igualdade é testada por valor e não por referência. Para o programador iniciante, isso pode parecer um pouco confuso, mas para você que já sabe o que é polimorfismo, é perceptível a importância do Java poder diferenciar entre variáveis de referência e por valor.

Mas surge uma dúvida, apesar de não ser o comportamento padrão comparar o conteúdo de dois objetos em Java, isso me parece uma operação frequente em qualquer linguagem de programação, então, como fazemos para comparar realmente o conteúdo dos atributos de dois objetos em Java? Faça a Atividade 3 e você terá a resposta.

Atividade prática 3

Como é uma operação muito comum em qualquer linguagem de programação, o Java possui uma forma de comparar o conteúdo de dois objetos, já que vimos que a simples comparação da referência do objeto é outra coisa e pode não dar o resultado esperado. Essa comparação do conteúdo é feita pelo método **equals( )**, conforme apresentado na classe a seguir. Teste a classe **TestaString2** e veja como agora conseguimos comparar o conteúdo de duas *Strings*.

```
1  public class TestaString2 {
2      public static void main(String args[]){
3          String a = new String("Java na Universidade");
4          String b = new String("Java na Universidade");
5          if (a.equals(b)) {
6              System.out.println("a e b tem conteudos iguais");
7          } else {
8              System.out.println("a e b tem conteudos diferentes");
9          }
10     }
11 }
```

Além de compararmos o conteúdo de *Strings*, muitas outras operações com cadeias de caracteres são comuns em programação, principalmente quando fazemos programas com grande quantidade de textos como é o caso de cadastros, de listas, de agendas, de pesquisas e muitos outros tipos de *softwares* comumente desenvolvidos no mercado de desenvolvimento de sistemas. Assim, não seria diferente com Java e a nossa linguagem provê muitos métodos importantes para manipulação textual por meio da classe *String*. Iremos mostrar os mais importantes, mas lembro que existem outros métodos nesta importante classe. Consulte os Javadocs da classe *String* para conhecer os detalhes de todos os métodos.

- **public String[ ] split(String regex):**
  - ✓ recebe como parâmetro uma *String* cuja expressão será a regra de separação da frase *String* ao qual o método *split* foi aplicado, retornando um *array* de *Strings*, onde cada posição do *array* é o fragmento de frase resultante da divisão da frase original, separado pela expressão.

• **public int compareTo(String outraString):**

✓ compara os caracteres *unicode* de duas *Strings* retornando 0 (zero) se as *Strings* forem iguais e diferente de 0 (zero) se forem diferentes.

• **public int compareToIgnoreCase(String outraString):**

✓ é um método bem interessante principalmente para utilização em pesquisas de nomes em cadastros e aplicações correlatas. Ele compara duas *Strings*, assim como no método *compareTo*, mas ignorando maiúsculas e minúsculas.

• **public String toUpperCase( ):**

✓ é um método muito conhecido das outras linguagens de programação e em Java executa a mesma operação, ou seja, converte todos os caracteres de uma *String* para maiúsculos.

• **public String toLowerCase( ):**

✓ é um método muito conhecido das outras linguagens de programação e em Java executa a mesma operação, ou seja, converte todos os caracteres de uma *String* para minúsculos.

• **public char charAt(int index):**

✓ método da classe String que recebe um número inteiro e retorna o caractere naquela posição da *String*. Vale lembrar que os índices dos caracteres de uma *String* se iniciam em 0 (zero).

• **public int length( ):**

✓ retorna o comprimento de uma *String*, que é igual ao seus número de caracteres. Lembre-se de que espaço vale como caracter de uma *String*. Além disso, cuidado para não confundir: o índice de um caracter numa *String* começa com 0 (zero), mas o tamanho de uma *String* com um único caractere é 1 (um), mesmo que ele seja somente um espaço.

• **public String substring(int beginIndex):**

✓ método que recebe um número inteiro que representa o índice do caractere inicial do trecho a ser dividido e retorna uma *String* que é um fragmento da *String* original a partir do índice inserido como parâmetro, inclusive este caractere.

• **public String trim( ):**

✓ método que retorna a *String* original retirados os espaços em branco no início e no fim de uma sequência de caracteres. Método muito interessante de ser utilizado no caso de captura de dados por meio de um formulário, em que o usuário pode equivocadamente inserir espaços antes ou depois de um campo.

• **public int indexOf(char ch):**

✓ método que recebe um caracter de uma *String* e retorna o índice (número inteiro) correspondente à posição da primeira incidência daquele caracter no *String*.

• **public int lastIndexOf(char ch):**

✓ método que recebe um caracter de uma *String* e retorna o índice (número inteiro) correspondente à posição da última incidência daquele caractere no *String*.

• **public boolean isEmpty( ):**

✓ método que retorna *true* caso o comprimento da *String* seja 0 (zero) e *false* caso contrário.

• **public boolean contains(SequenceChar s):**

✓ método que retorna *true* caso a *String* contenha a sequência de caracteres inserida como parâmetro do método. É um método para fazer uma busca de uma substring dentro de uma *String*.

• **public String replace(char oldChar, char newChar):**

✓ método que retorna a *String* original substituindo o caracter *oldChar* pelo caracter *newChar* passados como parâmetros do método.

Atividade prática 4

Vamos começar a fazer uma série de atividades de implementação de classes para entender melhor o funcionamento dos métodos das classes vistas até agora.

Para começar, implemente a classe *TestaMath* a seguir, que tem uma pequena amostra de como utilizar a biblioteca matemática do Java. Veja que não foi necessário instanciar nenhum objeto *Math*. Isso ocorre porque seus métodos dessa classe são todos estáticos, ou seja, sua instância é da classe e não precisamos de um objeto para utilizá-los.

```
1  public class TestaMath {
2      public static void main(String args[]){
3          int i = -10;
4          System.out.println(i);
5          int j = Math.abs(i);
6          System.out.println(j);
7
8          double d1 = 4.15;
9          System.out.println(d1);
10         //int k = Math.round(d1);
11         double d2 = Math.round(d1);
12         System.out.println(d2);
13     }
14 }
```

Atividade de reflexão 1

Vamos agora trabalhar com a classe **String**. Com as atividades a seguir perceberemos que essa importante classe serve para fazermos coisas muito mais interessantes do que simplesmente armazenar cadeias de caracteres, a única funcionalidade que estávamos utilizando.

Implemente e execute a classe **TestaString3** e escreva nas linhas a seguir quais foram os resultados nesse programa das operações sobre os métodos **split**, **compareTo**, **compareToIgnoreCase**, **toLowerCase** e **toUpperCase**. Anote também alguma observação que achou importante para discutirmos no fórum da semana.

```
1  public class TestaString3 {
2      public static void main(String args[]){
3          String a = "Java na Universidade";
4          String b[] = a.split(" ");
5          System.out.println(b[0]);
6          System.out.println(b[1]);
7          System.out.println(b[2]);
8
9          String c = "java na universidade";
10         int i = a.compareTo(a);
11         System.out.println(i);
12         int j = a.compareTo(c);
13         System.out.println(j);
14         int k = a.compareToIgnoreCase(c);
15         System.out.println(k);
16
17         a.toLowerCase();
18         System.out.println(a);
19         String d = a.toLowerCase();
20         System.out.println(d);
21         String e = a.toUpperCase();
22         System.out.println(e);
23     }
24 }
```

Atividade de reflexão **2**

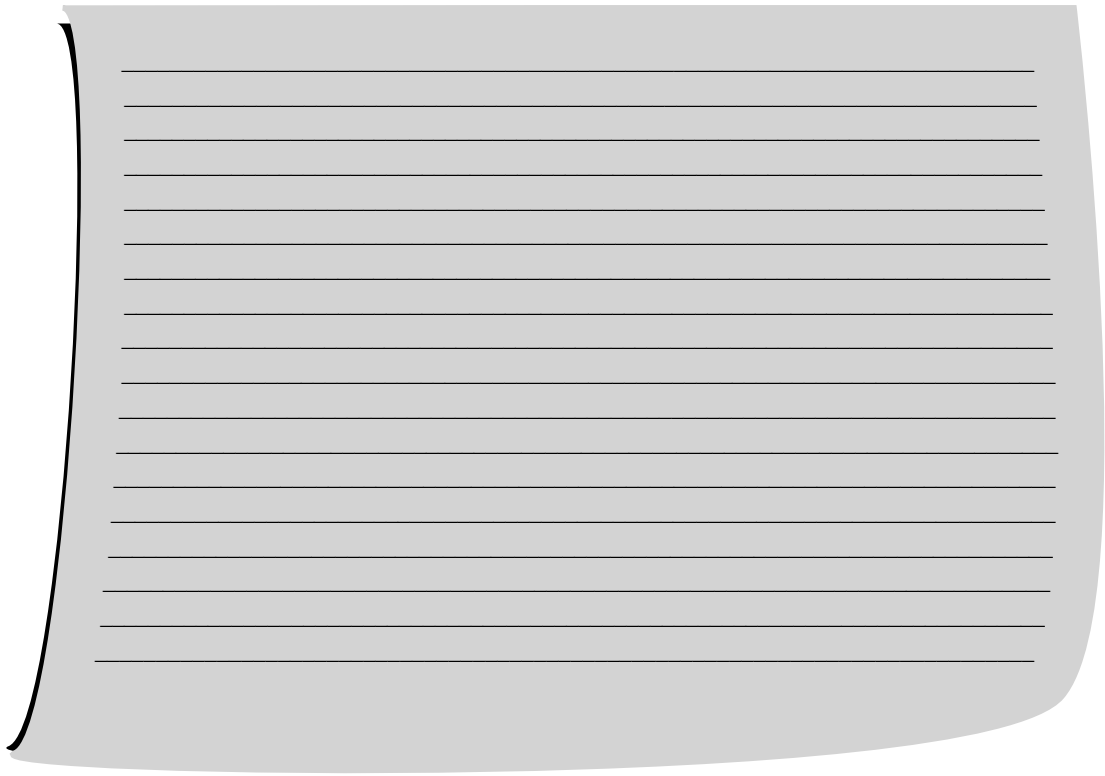
Continuando os trabalhos com a classe **String**, implemente e execute a classe **TestaString4** e escreva nas linhas a seguir quais foram os resultados nesse programa das operações sobre os métodos **charAt**, **length**, **substring**, **trim**, **indexOf** e **lastIndexOf**. Anote também alguma observação que achou importante para discutirmos no fórum da semana.


```
1  public class TestaString4 {
2      public static void main(String args[]){
3          String a = "Java na Universidade";
4          System.out.println(a);
5          char c = a.charAt(9);//zero base
6          System.out.println(c);
7          int i = a.length(); //nzero base
8          System.out.println(i);
9          String na = a.substring(8);//zero base
10         System.out.println(na);
11         String ss = a.trim();
12         System.out.println(ss);
13         int j = a.indexOf("U");//zero base
14         System.out.println(j);
15         int k = a.indexOf("u");//zero base
16         System.out.println(k);
17         int l = a.indexOf("a");//zero base
18         System.out.println(l);
19         int m = a.lastIndexOf("a");//zero base
20         System.out.println(m);
21     }
22 }
```

Atividade de reflexão 3

Implemente e execute essa última classe com métodos de **String**, a classe **TestaString5** e escreva nas linhas a seguir quais foram os resultados nesse programa das operações sobre os métodos **isEmpty**, **contains** e **replace**. Anote também alguma observação que achou importante para discutirmos no fórum da semana.


```
1  public class TestaString5 {
2      public static void main(String args[]){
3          String a = "Java na Universidade";
4          //String a = "";
5          //String a = " ";
6          boolean q1 = a.isEmpty();
7          System.out.println(q1);
8          //boolean q2 = a.contains("Java");
9          boolean q2 = a.contains("Java2");
10         System.out.println(q2);
11         a.replace("a","o");
12         System.out.println(a);
13         String b = a.replace("a","o");
14         System.out.println(b);
15     }
16 }
```



OUTRAS CLASSES IMPORTANTES DO JAVA STANDARD EDITION

Classes Wrapper para conversão de tipo

Um outro problema importante na programação pelo qual já passamos, mas que só encontramos algumas soluções nas classes das bibliotecas do Java, é o de conversão de tipos.

As classes responsáveis por conversão de tipo são chamadas classes **Wrapper**, já que elas "embrulham" a variável no tipo adequado que foi solicitada à conversão.

Você pode utilizar as classes **Wrapper** para efetuar conversões de *String* ou para utilizar tipos primitivos como classes.

Lembre que para transformarmos um tipo **int**, por exemplo, numa *String* basta utilizarmos o operador de concatenação "+" junto da *String* que está sendo utilizada.

Já para realizarmos a operação inversa, ou seja, transformarmos *String* em tipos primitivos, utilizamos as classes **Wrapper** com seus respectivos métodos de **parse** de acordo com o tipo solicitado.

Veja este exemplo, onde transformamos um **int** numa **String** somente concatenando-a com uma outra *String* e depois transformamos uma **String** num **int** por meio da classe **Wrapper Integer,** utilizando seu método **parseInt.** Sempre que quisermos realizar uma operação matemática com alguma *String* que foi recebida, devemos transformá-la num numérico antes de realizar o cálculo.

```
1  public class IntToString {
2      public static void main(String args[]){
3          int i = 1;
4          //String s = i;
5          String s1 = ""+i;
6          System.out.println(s1);
7
8          String s2 = "11";
9          //int j = i + s2;
10         int k = Integer.parseInt(s2);
11         int j = k + i;
12         System.out.println(j);
13     }
14 }
```

Existem classes *Wrapper* para todos os tipos primitivos, conforme mostrado na tabela abaixo:

Tipos primitivos de dados	Classes wrapper correspondentes
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Uma observação importante é que as classes **Wrapper** fazem parte do pacote **java.lang,** assim não precisam ser importadas para sua utilização.

Classes do pacote java.io para tratamento de entrada e saída

Continuando nosso passeio pelas bibliotecas nativas que já são contempladas pelo JDK, vamos ver um outro pacote muito importante, o **java.io,** que trata do processamento de entrada e saída em Java.

Em Java, todo tratamento de entrada e saída é feito como um fluxo de *bytes*, o que dá um poder polimórfico muito grande, visto que esse fluxo pode ser de qualquer coisa como um caractere, uma *String* ou uma linha de caracteres, e sua origem pode ser a padrão do sistema, um arquivo, um *socket*, um *post* ou qualquer outra coisa, mas tanto faz, pois serão tratados pelas mesmas classes. Afinal, temos o polimorfismo!

Num **fluxo de saída** em Java, podemos **escrever bytes** (**num arquivo, na tela etc.**), e a classe abstrata **OutputStream** define o comportamento desse fluxo.

Num **fluxo de entrada** em Java, podemos **ler bytes** (**de um arquivo, do teclado etc.**), e a classe abstrata **InputStream** define o comportamento desse fluxo.

Atividade prática 5

Vamos começar a executar as atividades dos programas referentes a operações de entrada e saída em Java.

Nessa primeira atividade você irá implementar uma classe que faz a leitura de uma linha inserida pelo teclado. Preste atenção nos comentários para aprender o que cada uma das classes utilizadas faz nesse programa. Por fim, execute essa classe na sua IDE e utilize a parte inferior no console para inserir uma frase qualquer e testar seu funcionamento.

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStream;
4  import java.io.InputStreamReader;
5
6  public class LeStringTeclado {
7      public static void main(String args[])throws IOException{
8          System.out.println("Digite uma tecla: ");
9          InputStream is = System.in; //IS sabe capturar um byte
10         //nesse caso recebendo-o da entrada padrao do sistema
11         InputStreamReader isr = new InputStreamReader(is);
12         //ISR sabe receber um byte e transforma-lo em char
13         BufferedReader br = new BufferedReader(isr);
14         //BR sabe receber varios char e armazena los num
15         //buffer transformando-os em String
16         String s;
17         s = br.readLine();
18         System.out.println("Voce digitou a frase: "+s);
19     }
20 }
```

Atividade prática 6

Vamos ser mais ousados agora. Nossa classe não lerá mais do teclado, o que estava fácil, já que o teclado é a entrada padrão do sistema. Agora queremos ler de um arquivo, então precisamos de uma filha concreta de **InputStream** que saiba ler um arquivo. Essa classe existe e se chama **FileInputStream**.

Crie um arquivo **teste.txt** como o da figura a seguir na raiz de seu projeto no NetBeans:

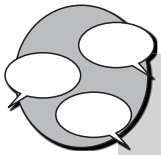
Implemente agora a classe para ler esse arquivo conforme o código a seguir e anote as observações para discutirmos no fórum da semana. O que precisamos fazer para lermos não apenas uma única linha, mas todas como na classe implementada?

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStream;
4  import java.io.InputStreamReader;
5  import java.io.FileInputStream;
6
7  public class LeArquivo {
8      public static void main(String args[])throws IOException{
9          InputStream is = new FileInputStream("teste.txt");
10         //Faz a leitura de um arquivo e o retorna como um stream de bytes
11         InputStreamReader isr = new InputStreamReader(is);
12         //ISR sabe receber um byte e transforma-lo em char
13         BufferedReader br = new BufferedReader(isr);
14         //BR sabe receber varios char e armazena-los num
15         //buffer transformando-os em String
16         String s;
17         s = br.readLine(); //só lê uma linha
18         System.out.println(s+"\n");
19
20         while(s != null) {
21             System.out.println(s);
22             s = br.readLine();
23         }
24     }
25 }
```

Atividade prática 7

Por fim, analogamente aos exemplos de leitura ou entrada de dados, implementados com a classe **InputStream**, vamos fazer a classe para realizar a escrita ou saída de dados, onde a classe a ser trabalhada é a **OutputStream**. Implemente a classe a seguir e veja que ao atualizar seu projeto com F5, o arquivo **escrita.txt** é criado e, ao abri-lo, você verá o conteúdo que foi escrito no seu interior com o método **write**.

```
1  import java.io.BufferedWriter;
2  import java.io.IOException;
3  import java.io.OutputStream;
4  import java.io.OutputStreamWriter;
5  import java.io.FileOutputStream;
6
7  public class EscreveArquivo {
8      public static void main(String args[])throws IOException{
9          OutputStream os = new FileOutputStream("escrita.txt");
10         OutputStreamWriter osw = new OutputStreamWriter(os);
11         BufferedWriter bw = new BufferedWriter(osw);
12         bw.write("Java na Universidade");
13         bw.close();
14         System.out.println("Escrita realizada com sucesso!");
15     }
16 }
```



INFORMAÇÕES SOBRE FÓRUM

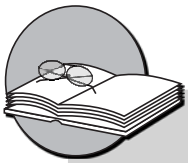
Vamos entrar no fórum para discutirmos as dúvidas de vocês. Participem, troquem informações.
Também não percam o ***chat*** da semana o horário e o dia serão colocados na página do fórum.

RESUMO

Vamos rever os principais conceitos vistos nesta aula.

- Uma das principais premissas da programação orientada a objetos é a reutilização de classes que já estejam prontas para serem utilizadas; assim, o Java Standard Edition possui mais de 3.500 classes para serem reutilizadas pelo programador.
- Dentre elas, as mais importantes para qualquer nível de desenvolvedor são as classes **Math** e **String**, que pertencem ao pacote **java.lang** e portanto não precisam ser importadas.

- Todos os detalhes dos métodos e outras informações importantes das classes **Math** e **String** podem ser obtidas por meio dos **javadocs** das respectivas classes no *site* da Sun, podendo também ser baixados para o seu computador para ser consultado.
- **Javadocs** é o formato de documentação oficial implementado pelo Java por meio de páginas navegáveis (html).
- As variáveis do tipo *String* armazenam parâmetros como referências a objetos e não um valor definido como nos tipos primitivos; assim, não podemos comparar os conteúdos de *Strings* com o operador de igualdade "==".
- Mas como é uma operação muito comum, o Java possui uma forma de comparar o conteúdo de dois objetos por meio do método **equals( )**.
- Um outro problema importante na programação é o de conversão de tipos, executados pelas classes **Wrapper**, que efetuam conversões de *String* ou para utilizar tipos primitivos como classes.
- Para transformarmos um tipo **int** numa *String*, basta utilizarmos o operador de concatenação "+" junto da *String* que está sendo utilizada.
- Já para realizarmos a operação inversa, utilizamos as classes **Wrapper** com seus respectivos métodos de **parse** de acordo com o tipo solicitado. Por exemplo, transformamos uma **String** num **int** por meio da classe **Wrapper Integer**, utilizando seu método **parseInt**.
- Sempre que quisermos realizar uma operação matemática com alguma *String* que foi recebida, devemos transformá-la num numérico antes de realizar o cálculo, mas também existem classes **Wrapper** para todos os outros tipos primitivos.
- Em Java, todo tratamento de entrada e saída é feito como um fluxo de *bytes* por meio das classes do pacote **java.io**, o que lhe concede alto nível de polimorfismo.
- Num **fluxo de saída** em Java, podemos **escrever bytes (num arquivo, na tela etc.)** e a classe abstrata **OutputStream** define o comportamento desse fluxo.
- Num **fluxo de entrada** em Java, podemos **ler bytes (de um arquivo, do teclado etc.)** e a classe abstrata **InputStream** define o comportamento desse fluxo.



Informação sobre a próxima aula

Na próxima aula, iremos começar uma das partes mais legais do Java SE, programação gráfica com AWT e Swing. Enfim, vamos começar a desenhar as famosas janelas *desktop* do Windows. Não percam.

AULA 9

Programação gráfica em Java I

Metas da aula

Introduzir os conceitos básicos e ensinar o aluno a desenvolver aplicações gráficas *desktop* em Java através das bibliotecas AWT e Swing.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. entender o funcionamento das bibliotecas AWT e Swing;
2. explicar o fluxo de informações e tratamento de eventos de aplicações gráficas *desktop* em Java;
3. criar aplicativos com recursos gráficos de janelas utilizando as bibliotecas gráficas do JDK.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512 MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; tenha compreendido bem todos os conceitos de Orientação a Objetos aprendidos nas últimas aulas, importantes para continuarmos o aprendizado de Java; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

INTRODUÇÃO

Apesar de você já conseguir projetar e executar uma série de aplicações em Java utilizando os conceitos aprendidos até agora, principalmente os relativos à orientação a objetos, desde o advento do sistema operacional Windows da Microsoft, mesmo nas suas versões mais antigas, um dos grandes atrativos da programação moderna é o desenvolvimento de aplicações gráficas baseadas em janelas, popularmente conhecido no mundo do desenvolvimento como PROGRAMAÇÃO DESKTOP.

PROGRAMAÇÃO DESKTOP

Ramo da programação voltado para o desenvolvimento de aplicativos baseados em janelas, que normalmente rodam localmente e por meio das GUI conseguem trazer facilidade de manipulação e riqueza de visualização para o usuário.

Assim, o Java provê no seu conjunto básico de desenvolvimento (JDK) uma série de bibliotecas para auxiliar na programação de aplicações *desktop*. As principais bibliotecas dessa categoria são a **Abstract Window Toolkit**, mais conhecida como **AWT**, e a **Swing**.

Vale lembrar que uma aplicação desenvolvida com recursos gráficos agrega muito valor a um sistema, fazendo com que ele ganhe em interatividade, usabilidade e experiência para o usuário, tudo isso por meio de recursos de interface gráfica com o usuário, também conhecido como **GUI** (**Graphical User Interface**).

No decorrer desta aula, vamos percorrer os principais componentes dessas bibliotecas por meio de vários exemplos além de aprendermos a forma como esses componentes interagem com a sua aplicação, conhecido como tratamento de eventos.

BIBLIOTECAS GRÁFICAS DO JDK

Dentro do JDK existe um conjunto de classes e bibliotecas que recebe o nome especial de *Java Foundation Classes* (JFC), sendo uma coleção de APIs que auxiliam no desenvolvimento de aplicações Java com GUI. Dessas APIs as duas principais são a **AWT** e a **Swing**.

APPLETS

São aplicações gráficas para *web* desenvolvidas com bibliotecas gráficas *desktop*, como **AWT** e **Swing**, e são inteiramente importadas para o usuário por ocasião de sua requisição dentro de uma página ou recurso na rede.

Além de aplicações *desktop*, essas bibliotecas também podem ser utilizadas para desenvolver aplicações web chamadas **APPLETS**.

Existem algumas diferenças importantes entre **AWT** e **Swing**. Na verdade, **AWT** é uma biblioteca mais antiga, existente no Java desde a sua versão 1.0, sendo que **Swing** seria sua atualização na parte de componentes gráficos. Vamos ver as principais características de cada uma dessas bibliotecas antes de utilizá-las.

Por questões de performance, como a AWT existe desde a primeira versão da JVM que ainda tinha alguns problemas quanto ao seu rendimento, foram desenvolvidos alguns componentes nativos, fazendo com que essa biblioteca seja dependente de plataforma para alguns pacotes. Assim, não só as classes como a aparência de sistema (*look and feel*) das janelas ficam dependentes do sistema operacional utilizado, reduzindo a portabilidade das aplicações feitas completamente com essa API.

Por outro lado, a biblioteca **Swing** é inteiramente escrita em Java, resolvendo os problemas de portabilidade da **AWT**, sendo portanto inteiramente independente de plataforma, podendo também o programador utilizar diferentes aparências em qualquer sistema operacional. A API **Swing** implementa vários componentes da **AWT**, mas também tem várias linhas de herança em comum.

A BIBLIOTECA ABSTRACT WINDOW TOOLKIT (AWT)

Aprendendo sobre os *Containers*

No desenvolvimento de aplicações gráficas, a técnica utilizada para montagem do sistema é a distribuição de componentes clássicos como botões, combos, áreas de texto e outros sobre tabuleiros onde serão distribuídos, formando uma arquitetura de mosaico. Esses tabuleiros, base para todos os outros componentes, são chamados de ***containers***, e constituem a base de toda a programação *desktop* em Java.

Os principais *containers* existentes em Java são:

- **Component**: é uma classe abstrata para objetos que interagem com o usuário; é a raiz de todos os outros componentes de **AWT**.
- ***Container***: subclasse concreta de **Component**, é um componente que pode conter outros componentes.
- **Panel**: subclasse de **Container**, é uma área de manipulação que pode ser colocada dentro de um **Window**, **Dialog** ou **Frame**.
- **Window**: subclasse de ***Container***, é uma janela simples que não tem borda ou barra de menu.
- **Dialog**: janela com barra de título e botão de fechar utilizada para criação de janelas interativas com o usuário.
- **Frame**: janela mais completa, com título, barra de menu, borda e outros atributos.

Assim, podemos utilizar algum desses ***containers*** mostrados para construirmos nossa aplicação de janelas.

Uma vez escolhido o tipo da janela, podemos configurar seu tamanho por meio do método **setSize** passando a largura e a altura como parâmetros:

void setSize(int width, int height)

Por padrão, a janela não é um objeto visível após sua execução. Para isso, normalmente a última operação executada na sua construção é a reconfiguração de seu parâmetro de visualização para ***true*** por meio do método **setVisible**:

void setVisible(boolean b)

Por ser o mais completo, o objeto *Frame* é o mais comumente utilizado em aplicações **AWT**, podendo ser mesclado com objetos **Panel**, como será mostrado futuramente.

Vamos ver nosso primeiro exemplo de janela feito em Java:

```
import java.awt.Frame;
public class Exemplo01 extends Frame{
    public static void main(String[] args) {
        Exemplo01 jf = new Exemplo01();
        //configura a largura e a altura da janela
        jf.setSize(100, 300);
        //faz a janela ficar visível, tente tirar!
        jf.setVisible(true);
    }
}
```

Nesse exemplo aqui exposto, instanciamos nosso primeiro objeto *Frame* configurando seu tamanho e fazendo-o aparecer na tela do computador por meio do método **setVisible**.

Atividade prática 1

Crie um novo projeto na IDE NetBeans somente para suas classes gráficas feitas com **AWT** e **Swing** e comece implementando a classe **Exemplo01** como mostrado no código anterior.

Por fim, execute e veja o resultado da sua primeira janela.

Vamos aprender a desenhar?

Dentro do pacote **AWT**, existe uma classe muito importante, responsável por executar operações de desenho numa janela: a classe **Graphics**.

Nessa classe, vários métodos nos auxiliam a executar operações de desenho como numa aplicação tipo *Paint*.

Na classe **Graphics**, podemos definir cores por meio de uma classe utilizada conjuntamente com ela, a classe **Color**, além de definir fontes e traços de desenhos como retângulos e polígonos.

Veja a classe **Exemplo02** para aprender uma pouco mais sobre a classe **Graphics** e seu funcionamento.

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Panel;
public class Exemplo02 extends Panel{
        Exemplo02(){
            //construtor de Exemplo02 utilizando Color
            setBackground(Color.BLUE);
        }
    public void paint(Graphics g) {
        //setColor: define cor no padrão RGB
        g.setColor(new Color(255, 0, 0));
        //setFont: define as configurações da fonte
        g.setFont(new Font("Arial", Font.BOLD, 16));
        g.drawString("Primeiro exemplo com Graphics", 30, 100);
        g.setColor(new Color(0, 1.0f, 0));
        g.fillRect(30, 100, 150, 10);
    }

    public static void main(String[] args) {
        Frame f = new Frame("Testando a classe Graphics!!!");
        Exemplo02 ex02 = new Exemplo02();
        f.add(ex02);
        f.setSize(400,200);
        f.setVisible(true);
    }
}
```

Atividade prática **2**

Crie uma nova classe **Exemplo02** para implementar nosso código já mostrado. Por fim, compile e execute para entender as funcionalidades da classe **Graphics**.

Componentes internos de AWT

Vamos agora aprender sobre os componentes internos de um ***container*** em **AWT**, aqueles que vão compor a sua aplicação.

Como já explicamos, você tem um tabuleiro (seu ***container***) sobre o qual irá arquitetar seu mosaico. As peças desse mosaico são compostas desses componentes internos ou microcomponentes de **AWT**. Os principais deles são:

- **Label**: componente que facilita a escrita de texto na aplicação, no formato de etiqueta.
- **TextField**: componente que serve para escrever um texto de entrada na aplicação, de apenas uma linha.
- **TextArea**: componente semelhante ao **TextField**, mas que suporta mais de uma linha.
- **Button**: componente que desenha um botão na aplicação da janela.

Além desses componentes básicos, temos outros componentes mais elaborados, como ***Checkbox*** (opções), ***Choice*** (combo), ***List*** (lista de escolha) e ***Scrollbar*** (barra de rolagem).

Veja no exemplo a seguir a utilização de muitos desses componentes internos de **AWT**:

```
import java.awt.*;
public class Exemplo03 extends Frame{
    public static void main(String[] args) {
        Exemplo03 ex03 = new Exemplo03();
        ex03.setLayout(new FlowLayout());
        ex03.setSize(600, 100);
        ex03.add(new Button("Click aqui!!!"));
        ex03.add(new Label("Texto de etiqueta!"));
        ex03.add(new TextField());
        CheckboxGroup cbg = new CheckboxGroup();
        ex03.add(new Checkbox("homem", cbg, true));
        ex03.add(new Checkbox("mulher", cbg, true));
        List list = new List(3, false);
        list.add("Java");
        list.add("C");
        list.add("C++");
        ex03.add(list);
        Choice chooser = new Choice();
        chooser.add("João");
        chooser.add("Maria");
        chooser.add("José");
        ex03.add(chooser);
        ex03.add(new Scrollbar());
        ex03.setVisible(true);
    }
}
```

Atividade prática 3

Crie uma nova classe **Exemplo03** para implementar nosso código já mostrado acima. Por fim, compile e execute para entender como criar componentes internos de **AWT**.

Diferentes formas para seu mosaico AWT

Agora que você já começou a preencher com componentes internos seus macrocomponentes ou ***containers***, você precisa aprender como organizar melhor as peças no seu tabuleiro, ou seja, organizar os microcomponentes no ***container***.

Para organizar as peças no tabuleiro, utilizamos estruturas chamadas **Gerenciadores de *Layout***, que são os responsáveis por determinar a posição e o tamanho dos componentes no *container*.

Os Gerenciadores de *Layout* em Java são:

- FlowLayout;
- BorderLayout;
- GridLayout;
- GridBagLayout;
- CardLayout;
- BoxLayout.

Vamos aprender o conceito e a manipulação dos **Gerenciadores de *Layout*** mais utilizados na programação desktop em Java.

A configuração do **Gerenciador de *Layout*** em Java é feita por meio do método **setLayout**, passando como parâmetro o gerenciador a ser utilizado.

void setLayout(LayoutManager mng)

Vamos começar com o gerenciador mais simples de todos, o **FlowLayout**, que é o gerenciador padrão da classe **Panel** e suas subclasses. Esse gerenciador posiciona os componentes da esquerda para a direita e de cima para baixo, começando no canto superior esquerdo. O que pode ser alterado no construtor é o alinhamento dos componentes, podendo ser central, alinhado à esquerda ou alinhado à direita.

Aceita três construtores diferentes, podendo ser definido o alinhamento, o *gap* horizontal e o *gap* vertical da estrutura, de acordo com o construtor e os valores de alinhamentos escolhidos.

Os possíveis construtores para **FlowLayout** são:

- **FlowLayout( );**
- **FlowLayout(int alinhamento);**
- **FlowLayout(int alinhamento, int gapHoriz, int gapVert).**

Os valores de alinhamento possíveis são:

- **FlowLayout.LEFT;**
- **FlowLayout.CENTER;**
- **FlowLayout.RIGHT.**

Observe o exemplo a seguir da classe Exemplo 04 e veja como os microcomponentes de botão são distribuídos com FlowLayout:

```
import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;
public class Exemplo04 extends Frame{
    public static void main(String[] args) {
        Exemplo04 ex04 = new Exemplo04();
        ex04.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 10));
        ex04.add(new Button("Botão 1"));
        ex04.add(new Button("Botão 2"));
        ex04.add(new Button("Botão 3"));
        ex04.setSize(150, 110);
        ex04.setVisible(true);
    }
}
```

Atividade prática **4**

Crie uma nova classe **Exemplo04** para implementar nosso código mostrado anteriormente. Por fim, compile e execute para entender o comportamento do **Gerenciador de *Layout* FlowLayout**.

Redimensione o tamanho da janela criada arrastando sua borda com o mouse e perceba qual será o comportamento dos microcomponentes de seu *container*.

Vamos agora aprender um outro **Gerenciador de *Layout***: o **BorderLayout.** Ele é o gerenciador padrão da classe **Window** e suas subclasses.

A maioria das ferramentas *desktop* que conhecemos é desenhada com esse gerenciador, que é um dos mais utilizados para programação de aplicações de janelas mais complexas.

O **BorderLayout** divide o objeto *container* em cinco campos onde objetos do tipo **Component** podem ser adicionados, inclusive objetos **Panel**, como veremos depois para compormos aplicações mais complexas. Esses campos são distribuídos de acordo com a orientação cardeal, com características específicas, conforme mostrado a seguir:

- **North**: espalhamento horizontal na parte superior do *container*;
- **South**: espalhamento horizontal na parte inferior do *container*;
- **East**: ajuste vertical na porção direita do *container*;
- **West**: ajuste vertical na porção esquerda do *container*;
- **Center**: ajuste em todas as direções.

Uma observação importante: só pode ser colocado um componente por região.

Os valores de regiões válidas são:

- **BorderLayout.NORTH**;
- **BorderLayout.SOUTH**;
- **BorderLayout.EAST**;
- **BorderLayout.WEST**;
- **BorderLayout.CENTER**.

Vamos ver o exemplo abaixo que trabalha com BorderLayout:

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;
public class Exemplo05 extends Frame{
    public static void main(String[] args) {
        Exemplo05 ex05 = new Exemplo05();
        ex05.setLayout(new BorderLayout(10, 10));
        ex05.add(new Button("Botão Norte!!!"), BorderLayout.NORTH);
        ex05.add(new Button("Botão Sul!!!"), BorderLayout.SOUTH);
        ex05.add(new Button("Botão Direito!!!"), BorderLayout.EAST);
        ex05.add(new Button("Botão Esquerdo!!!"), BorderLayout.WEST);
        ex05.add(new Button("Botão Central!!!"), BorderLayout.CENTER);
        ex05.setSize(200, 200);
        ex05.setVisible(true);
    }
}
```

Atividade prática 5

Crie uma nova classe **Exemplo05** para implementar nosso código mostrado anteriormente. Por fim, compile e execute para entender o comportamento do **Gerenciador de *Layout* BorderLayout**.

Redimensione o tamanho da janela criada arrastando o mouse e perceba qual será o comportamento dos microcomponentes de seu *container*.

Um outro Gerenciador de Layout muito interessante é o GridLayout, que posiciona os componentes da esquerda para a direita e de cima para baixo, mas dividindo o *container* em um número de linhas e colunas, como células.

Para isso esse gerenciador suporta um construtor no qual o programador define por ocasião da sua criação a quantidade de linhas e colunas desejada no *container*:

GridLayout(int nroLinhas, int nroColunas)

Vamos ver o exemplo da classe Exemplo06 para entendermos melhor o funcionamento desse interessante gerenciador que divide nosso *container* em células:

```
import java.awt.Button;
import java.awt.Frame;
import java.awt.GridLayout;
public class Exemplo06 extends Frame{
    public static void main(String[] args) {
        Exemplo06 ex06 = new Exemplo06();
        ex06.setLayout(new GridLayout(2, 3));
        ex06.add(new Button("Um"));
        ex06.add(new Button("Dois"));
        ex06.add(new Button("Três"));
        ex06.add(new Button("Quatro"));
        ex06.add(new Button("Cinco"));
        ex06.setSize(200, 200);
        ex06.setVisible(true);
    }
}
```

Atividade prática **6**

Crie uma nova classe **Exemplo06** para implementar nosso código mostrado anteriormente. Por fim, compile e execute para entender o comportamento do **Gerenciador de *Layout* GridLayout**.

Redimensione o tamanho da janela criada arrastando o mouse e perceba qual será o comportamento dos microcomponentes de seu *container*.

Perceba que, caso você adicione menos microcomponentes do que o número de linhas vezes o número de colunas, esse espaço ficará vazio.

Para desenvolvermos aplicações mais complexas, precisaremos combinar diferentes gerenciadores.

Além disso, uma técnica interessante é criarmos vários objetos **Panel** e nele inserirmos as porções que queremos desenhar no nosso mosaico, que será posteriormente inserido num ***Frame***, por exemplo, conforme podemos ver no exemplo a seguir:

```
import java.awt.*;

public class Exemplo07 extends Frame{
    public static void main(String[] args) {
        Exemplo07 ex07 = new Exemplo07();
        Panel painelNorte = new Panel();
        Panel painelCentral = new Panel();
        Panel painelSul = new Panel();

        painelNorte.add(new Button("Um"));
        painelNorte.add(new Button("Dois"));
        painelNorte.add(new Button("Três"));

        painelCentral.setLayout(new GridLayout(4, 4));
        painelCentral.add(new TextField("Primeiro"));
        painelCentral.add(new TextField("Segundo"));
        painelCentral.add(new TextField("Terceiro"));
        painelCentral.add(new TextField("Quarto"));
```

```
        painelSul.setLayout(new BorderLayout());
        painelSul.add(new Checkbox("Casado"), BorderLayout.CENTER);
        painelSul.add(new Checkbox("Solteiro"), BorderLayout.EAST);
        painelSul.add(new Checkbox("Separado"), BorderLayout.WEST);

        ex07.add(painelNorte, BorderLayout.NORTH);
        ex07.add(painelCentral, BorderLayout.CENTER);
        ex07.add(painelSul, BorderLayout.SOUTH);
        ex07.setSize(300, 300);
        ex07.setVisible(true);
    }
}
```

Atividade prática 7

Vamos ver se você entendeu como funciona a criação de *layouts* complexos em programação *desktop* Java?

Crie uma classe **Exemplo07** e implemente a classe do exemplo anterior, analisando-a com cuidado para entender todos os detalhes da criação e construção de painéis sobre um *container*.

Rumo à próxima geração: componentes da biblioteca Swing

Conforme explicamos no início desta aula, tínhamos alguns problemas e limitações em relação a alguns componentes do pacote da biblioteca AWT. Com isso, o Java evoluiu sua plataforma de aplicações gráficas para uma nova biblioteca chamada **Swing**, contida no pacote **javax.swing**.

Essa nova biblioteca é inteiramente escrita em Java e fornece componentes com mais funcionalidades e melhor acabamento.

Os componentes básicos do **AWT** foram apenas reescritos, mas praticamente continuam os mesmos, inclusive com o mesmo nome, sendo que nos nomes dos componentes da biblioteca **Swing** foi acrescentado o prefixo **J**, conforme pode ser visto na lista de seus principais componentes a seguir:

- **JComponent;**
- **JButton;**
- **JCheckBox;**
- **JFileChooser;**
- **JTextField;**
- **JFrame;**
- **JPanel;**
- **JApplet;**
- **JOptionPane;**
- **JDialog;**
- **JColorChooser.**

Vamos ver um exemplo simples de utilização das classes de *Swing*:

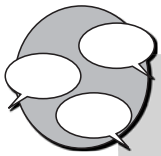
```
import javax.swing.JFrame;
public class Exemplo08 extends JFrame{
    public Exemplo08(){
        super("Minha primeira janela com Swing");
        this.setSize(200, 200);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new Exemplo08();
    }
}
```

Atividade prática **8**

Nesta atividade, temos duas tarefas.

Primeiro implemente a classe anterior para travar o primeiro contato com a biblioteca **Swing**, então compile e execute o exemplo para ver se tudo ocorreu sem problemas.

Após isso, reimplemente todos os nossos exemplos anteriores, mas agora utilizando os componentes da biblioteca **Swing**.



INFORMAÇÕES SOBRE FÓRUM

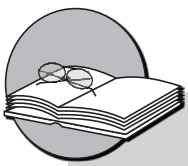
Vamos entrar no fórum para discutirmos as dúvidas de vocês. Participem, troquem informações.
Também não percam o *chat* da semana, o horário e dia serão colocados na página do fórum.

RESUMO

Vamos rever os principais conceitos vistos nesta aula:

- As principais bibliotecas para programação *desktop* (construídas em sistemas de janelas) em Java e que fazem parte do JDK são a **Abstract Window Toolkit**, mais conhecida como **AWT**, e a **Swing**.
- **AWT** e **Swing** fazem parte de um conjunto de classes e bibliotecas do JDK chamadas Java Foundation Classes (JFC).
- Além de aplicações *desktop*, essas bibliotecas também podem ser utilizadas para desenvolver aplicações *web* chamadas **Applets**;
- A **AWT** é uma biblioteca mais antiga, existente no Java desde a sua versão 1.0, sendo **Swing** sua atualização na parte de componentes gráficos.
- A biblioteca **AWT**, por questões de performance possui alguns componentes nativos, fazendo com que essa biblioteca seja dependente de plataforma para alguns pacotes.
- A biblioteca **Swing** é inteiramente escrita em Java, resolvendo os problemas de portabilidade da **AWT**, sendo portanto inteiramente independente de plataforma.
- No desenvolvimento de aplicações gráficas, a técnica utilizada para montagem do sistema é a distribuição de componentes clássicos (microcomponentes) como botões, combos, áreas de texto e outros sobre tabuleiros onde serão distribuídos, formando uma arquitetura de mosaico. Esses tabuleiros, base para todos os outros componentes, são chamados de **containers**, e consistem na base de toda a programação desktop em Java.

- Os principais *containers* existentes em Java são **Component**, **Container**, **Panel**, **Window**, **Dialog** e **Frame**.
- Por ser o mais completo, o objeto ***Frame*** é o mais comumente utilizado em aplicações AWT, podendo ser mesclado com objetos **Panel**.
- A classe **Graphics** é responsável por executar operações de desenho numa janela. Nessa classe, vários métodos nos auxiliam a executar operações de desenho como numa aplicação tipo *Paint*.
- Numa aplicação *desktop* você tem um tabuleiro (seu *container*) sobre o qual você irá arquitetar seu mosaico. As peças desse mosaico são compostas desses componentes internos ou microcomponentes de **AWT**. Os principais deles são **Label**, **TextField**, **TextArea**, **Button**, **CheckboxChoice**, **List** e **Scrollbar**.
- A técnica para organizar as peças no tabuleiro é feita por meio de estruturas chamadas Gerenciadores de *Layout*, que são os responsáveis por determinar a posição e o tamanho dos componentes no container.
- Os **Gerenciadores de *Layout*** em Java são **FlowLayout**, **BorderLayout**, **GridLayout**, **GridBagLayout**, **CardLayout** e **BoxLayout**.
- A biblioteca Swing está no pacote **javax.swing**, e nela os componentes básicos do **AWT** foram apenas reescritos, mas praticamente continuam os mesmos, inclusive com o mesmo nome, e nos nomes dos componentes da biblioteca Swing foi acrescentado o prefixo J, sendo que seus principais componentes são **JComponent**, **JButton**, **JCheckBox**, **JFileChooser**, **JTextField**, **JFrame**, **JPanel**, **JApplet**, **JOptionPane**, **JDialog** e **JColorChooser**.



Informação sobre a próxima aula

Na próxima aula, iremos continuar com programação gráfica em Java, aprendendo uma importante parte que é necessária para definirmos a ação que ocorrerá a partir da interação do usuário com o componente. Essa parte se chama tratamento de eventos. Não perca!

AULA 10

Programação gráfica em Java II

Meta da aula

Introduzir os conceitos básicos e ensinar o aluno a desenvolver aplicações gráficas desktop em Java por meio das bibliotecas AWT e Swing.

objetivos

Após o estudo do conteúdo desta aula, esperamos que você seja capaz de:

1. entender o funcionamento das bibliotecas AWT e Swing;
2. explicar o fluxo de informações e tratamento de eventos de aplicações gráficas desktop em Java;
3. criar aplicativos com recursos gráficos de janelas utilizando as bibliotecas gráficas do JDK.

Pré-requisitos

Para o correto e completo aprendizado desta aula, precisamos que você tenha um computador com o Windows XP ou Windows Vista instalado com pelo menos 512MB de memória RAM; tenha instalado e configurado o Java Development Kit (JDK) conforme as instruções do Apêndice A; tenha instalado o Ambiente de Desenvolvimento Integrado (IDE) NetBeans conforme as instruções do Apêndice B; tenha compreendido bem todos os conceitos de Orientação a Objeto aprendidos nas últimas aulas, importantes para continuarmos o aprendizado de Java; esteja animado e motivado para iniciar o aprendizado dessa fabulosa linguagem de programação que está dominando os mercados de desenvolvimento, programação e tecnologia na atualidade.

INTRODUÇÃO

Vamos dar continuidade ao aprendizado da programação gráfica em Java com as APIs AWT e Swing.

Na aula passada, vimos a primeira parte dessa programação, na qual você aprendeu a fazer a parte gráfica propriamente dita, ou seja, aprendeu como construir e desenhar as janelas de sua aplicação com seus componentes de botão, caixas de entrada de texto, combos, listas e outros que farão parte da interface gráfica com o usuário ou GUI (**Graphical User Interface**).

Graphical User Interface

Interface com o usuário que permite a interação com dispositivos digitais por meio de elementos gráficos como ícones, botões e outros indicadores visuais, em contraste com a interface de linha de comando.

O que acontece se o usuário interagir com algum componente da GUI? O que irá disparar? Como será tratado? É esse conceito que iremos ver nesta aula, o chamado **Tratamento de Eventos em Interfaces Gráficas**.

Tratamento de Eventos é, portanto, a técnica de capturar a ação que o usuário realizou por meio da GUI, interpretá-la identificando qual componente foi acionado e por último dar a resposta desejada à ação daquele componente analisando, se for o caso, o conteúdo inserido pelo usuário através deste componente.

Vamos nas próximas sessões ensinar mais alguns conceitos sobre essa técnica e como realizar sua programação para que consigamos de forma correta interpretar o que o usuário está querendo realizar. No final, também vamos ensinar algumas facilidades que a IDE NetBeans nos proporciona ao trabalharmos com a programação de GUIs.

MODELO DE TRATAMENTO DE EVENTOS

O Modelo de Tratamento de Eventos descreve o modo como sua classe pode responder a uma interação do usuário.

Esse modelo é dividido em três componentes, que são definidos e utilizados da seguinte forma:

1. **Gerador de Evento (Event Source):** é o componente da interface que origina o evento. Por exemplo, caso o usuário pressione o botão para solicitar uma pesquisa, nesse caso o Gerador de Evento é o botão que foi clicado.
2. **Monitor de Eventos/Manipulador (Event Listener/Handler):** recebe as informações enviadas pelo Gerador de Evento processando a interação. Por exemplo, no caso do usuário que apertou

o botão, o Monitor de Eventos pode receber uma informação que foi enviada e retornar uma mensagem ao usuário relativa a essa recepção e tratamento.

3. **Objeto Evento (Event Object):** quando um evento ocorre, o pressionar de um botão por exemplo, um Objeto Evento é criado. Ele contém todas as informações necessárias sobre o evento gerado. Um exemplo seria o clicar do botão direito do mouse sobre um botão.

Quando programamos um evento que pode ocorrer, um ***listener*** (**Monitor de Eventos**) deve ser registrado pelo **Gerador de Evento**, pois assim esse ***listener*** irá receber as informações do evento que ocorrer nesse **Gerador de Evento**. Uma vez registrado esse ***listener*** fica aguardando a geração de evento sobre esse componente gerador do evento.

Quando algum evento ocorre no gerador do evento, um **Objeto Evento** é criado, ele é o descritor do evento. Após isso, o evento é disparado pelo gerador para os ***listeners*** registrados.

Quando um ***listener*** recebe uma notificação de um gerador de evento, ele executa sua função, interpretando e processando o evento ocorrido.

Registrando os *listeners*

Como vimos, um dos passos do programador no processo de tratamento de eventos é fazer o registro dos ***listeners***. Esse registro é feito da seguinte forma no código:

void add<TipoDoListener>**Listener**(<TipoDoListener>**Listener listenerObj**)

Onde a parte <TipoDoListener> deve ser substituído pelo **Gerador de Evento** associado ao evento escolhido pelo programador, podendo ser **Key**, **Mouse**, **Focus**, **Component**, **Action** etc.

Você pode ter mais de um ***listener*** associado a um **Gerador de Evento**.

Quem são meus Objetos de Evento?

Um **Objeto de Evento** tem uma classe de evento que indica seu tipo. Vamos ver as classes dos principais eventos associados a um processo de evento programado em AWT e Swing, todos subclasses de **AWTEvent**:

Classes de Evento	***Descrição***
ComponentEvent	Estende a classe *AWTEvent*. Instanciada quando um componente é movido, redimensionado, tornado visível ou invisível.
InputEvent	Estende a classe *ComponentEvent*. Classe abstrata de evento, raiz a partir da qual são implementadas todas as classes de componentes de entrada de dados.
ActionEvent	Estende a classe *AWTEvent*. Instanciada quando um botão é pressionado, um item de uma lista recebe duplo-clique ou quando um item de menu é selecionado.
ItemEvent	Estende a classe *AWTEvent*. Instanciada quando um item é selecionado ou desmarcado pelo usuário, seja numa lista ou caixa de seleção (checkbox).
KeyEvent	Estende a classe *InputEvent*. Instanciado quando uma tecla é pressionada, liberada ou digitada (pressionada e liberada).
MouseEvent	Estende a classe *InputEvent*. Instanciada quando um botão do mouse é pressionado, liberado, clicado (pressionado e liberado) ou quando o cursor do mouse entra ou sai de uma parte visível de um componente.
TextEvent	Estende a classe *AWTEvent*. Instanciada quando o valor de um campo texto ou área de texto sofre alteração.
WindowEvent	Estende a classe *ComponentEvent*. Instanciada quando um objeto *Window* é aberto, fechado, ativado, desativado, minimizado, restaurado ou quando o foco é transferido para dentro ou para fora da janela.

Quem são meus Monitores de Evento ou *listeners*?

Monitores ou listeners de Evento são classes que implementam as interfaces de ***listener*** e possuem métodos que devem ser implementados para tratar a ação que ocorreu por ocasião do evento.

Primeiramente, vamos ver as classes que implementam as interfaces ***listeners***:

***Listeners* de Evento**	***Descrição***
ActionListener	Recebe eventos de ação. Estes podem ser um pressionamento do mouse ou da barra de espaço sobre o objeto.
MouseListener	Recebe eventos do mouse.
MouseMotionListener	Recebe eventos de movimento do mouse, que incluem arrastar e mover o mouse.
WindowListener	Recebe eventos de janela (abrir, fechar, minimizar, entre outros).

Vamos agora ver os métodos que essas classes implementam:

ActionListener:

Médoto de *ActionListener*
`public void actionPerformed(ActionEvent e)`
Chamado quando o mouse é pressionado ou quando foi utilizada a barra de espaço sobre um botão por exemplo.

MouseListener:

Métodos de *MouseListener*
`public void mouseClicked(MouseEvent e)`
Chamado quando o mouse é pressionado (pressionar e soltar).
`public void mouseEntered(MouseEvent e)`
Chamado quando o cursor do mouse entra em um componente.
`public void mouseExited(MouseEvent e)`
Chamado quando o cursor do mouse sai de um componente.
`public void mousePressed(MouseEvent e)`
Chamado quando o botão do mouse é pressionado sobre um componente (pressionar).
`public void mouseReleased(MouseEvent e)`
Chamado quando o botão do mouse é solto sobre um componente (soltar).

MouseMotionListener:

Métodos MouseListener
`public void mouseDragged(MouseEvent e)`
Chamado quando o mouse é pressionado sobre um componente e então arrastado (*dragged*). É chamado tantas vezes quantas o mouse for arrastado.
`public void mouseMoved(MouseEvent e)`
Chamado quando o cursor do mouse é movido sobre um componente, sem que o mouse esteja pressionado. Será chamado múltiplas vezes, tantas quantas o mouse for movido.

WindowListener:

Métodos WindowListener
`public void windowOpened(WindowEvent e)`
Chamado quando uma janela é aberta (quando o objeto *Window* torna-se visível pela primeira vez)
`public void windowClosing(WindowEvent e)`
Chamado quando uma janela é encerrada (objeto *Window*).
`public void windowClosed(WindowEvent e)`
Chamado quando a janela foi fechada após a liberação de recursos utilizados pelo objeto (*EventSource*).
`public void windowActivated(WindowEvent e)`
Chamado quando uma janela é ativada, ou seja, está em uso.
`public void windowDeactivated(WindowEvent e)`
Chamado quando uma janela deixa de ser a janela ativa.
`public void windowIconified(WindowEvent e)`
Chamado quando a janela é iconificada. (Este evento é utilizado especialmente para o ambiente Macintosh).
`public void windowDeiconified(WindowEvent e)`
Chamado quando a janela retorna do estado iconificado para o normal. (Este evento é utilizado especialmente para o ambiente Macintosh).

Resumindo o processo e aplicando o Tratamento de Eventos

Então vamos resumir os passos e ações necessárias para criar uma aplicação gráfica GUI com Tratamento de Eventos:

1. Crie a parte visual de sua aplicação, a **GUI** propriamente dita, com seu container e seus componentes de botões, textfields, combos, listas etc.;
2. Crie uma classe que implemente a interface ***listener*** apropriada. Normalmente, o código desse passo pode estar incluído na classe do passo 1;
3. Na classe criada no passo 2, sobrescrever todos os métodos da interface ***listener*** escolhida, sendo que caso exista algum método que não vá tratar o evento, basta deixá-lo vazio;
4. Registre o objeto listener no **Gerador de Evento** por meio do método **add<TipoDoListener>Listener**.

Vamos ver um primeiro exemplo que aplica todos esses passos numa única classe, para você entender todos esses conceitos:

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class ActionDemo extends JFrame implements ActionListener{
    private JTextField tf;
    private JButton bt;
    public ActionDemo(String title){
        super(title);
        tf = new JTextField();
        bt = new JButton("Clique aqui!!!");
        add(tf, BorderLayout.SOUTH);
        add(bt, BorderLayout.NORTH);
        setSize(240, 240);
        bt.addActionListener(this);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if(tf.getText().equals("Ação solicitada!"))
            tf.setText("");
        else
            tf.setText("Ação!");
    }
    public static void main(String[] args) {
        new ActionDemo("Minha primeira janela que trata eventos...");
    }
}
```

Atividade prática 1

No seu projeto do NetBeans em que você está programando as classes AWT e Swing, crie a classe anterior e procure observar todos os passos do processo de criação de aplicações gráficas explicado na sessão anterior. Por fim, execute e veja o resultado da sua primeira janela com tratamento de eventos.

CONCEITOS COMPLEMENTARES SOBRE CLASSES E ORIENTAÇÃO A OBJETO

Vamos explicar mais alguns conceitos complementares sobre classes e orientação a objeto aplicando-os a classes que implementam programação gráfica.

O primeiro conceito que vamos ver é o de ***anonymous inner class***, que é uma forma de simplificarmos uma classe já escrita por meio da criação de classes internas sem as mesmas estarem declaradas; funciona como se fosse uma chamada de método em que você coloca sua implementação em outra parte por questão de organização. Vamos ver a mesma classe anterior, mas agora com uma implementação de ***anonymous inner class***:

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class ActionDemo2 extends JFrame {
    private JTextField tf;
    private JButton bt;
    public ActionDemo2(String title){
        super(title);
        tf = new JTextField();
        bt = new JButton("Clique aqui!!!");
        add(tf, BorderLayout.SOUTH);
        add(bt, BorderLayout.NORTH);
        setSize(440, 240);
        bt.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent actionEvent) {
                method();
            }
        });
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    public void method() {
        if(tf.getText().equals("Ação solicitada!"))
            tf.setText("");
        else
            tf.setText("Ação!");
    }

    public static void main(String[] args) {
        new ActionDemo2("Minha segunda janela que trata eventos...");
    }
}
```

Como podemos observar, a classe ficou mais organizada a partir do momento em que separamos a parte gráfica do tratamento de eventos propriamente dito; este foi apenas chamado através de **method**( ). Procure sempre criar classes mais simples, claras e, principalmente, desacopladas entre as funcionalidades e camadas.

Vamos ver agora como realizar o tratamento de um evento de fechamento de janela por meio de um **WindowListener** (outro tipo de **listener**):

```
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.JFrame;
public class WindowDemo extends JFrame{
    public WindowDemo(String title){
        super(title);
        this.setSize(350, 350);

        this.addWindowListener(new WindowListener() {
            public void windowActivated(WindowEvent e) {
            }
            public void windowClosed(WindowEvent e) {
            }
            public void windowClosing(WindowEvent e) {
                fechar();
            }
            public void windowDeactivated(WindowEvent e) {
            }
            public void windowDeiconified(WindowEvent e) {
            }
            public void windowIconified(WindowEvent e) {
            }
            public void windowOpened(WindowEvent e) {
            }
        });

        this.setVisible(true);
    }
     public void fechar(){
         System.exit(0);
     }

     public static void main(String[] args) {
         new WindowDemo("Exemplo de fechamento de janela");
     }
}
```

Tudo bem, até aí é tudo que já aprendemos, utilizamos o método ***addWindowListener*** para registrar o ***listener*** e o passamos como parâmetro. Então sobrescrevemos todos os métodos da interface ***listener***, inclusive os que não iremos utilizar, deixando-os vazios simplesmente com o *{ ... }* (abre e fecha chaves). Funciona perfeitamente, mas ficou um pouco feio, não ficou? Apesar de conseguirmos fazer tudo que queríamos, será que não conseguimos fazer nosso código ficar um pouco mais elegante, legível, claro e simples? É o que aprenderemos a fazer no próximo exemplo:

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
public class WindowDemo2 extends JFrame{
    public WindowDemo2(String title){
        super(title);
        this.setSize(350, 350);

        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                fechar();
            }
        });

        this.setVisible(true);
    }

    public void fechar(){
        System.exit(0);
    }

    public static void main(String[] args) {
        new WindowDemo2("Exemplo de fechamento de janela 2, agora com ADAPTER");
    }
}
```

Vejam como ficou muito mais simples. O que fizemos foi substituir o parâmetro passado para o método ***add<Tipo>Listener***, em vez de passarmos ***<Tipo>Listener***, passamos uma classe ***Adapter*** no formato ***<Tipo>Adapter*** (no nosso caso, como o método ***add*** foi ***addWindowListener***, a classe passada como parâmetro foi ***WindowAdapter***). Com isso podemos implementar apenas o método desejado. Veja que o preço disso é termos de saber o método exato que queremos implementar, o que na verdade não é muito difícil e é muito mais vantajoso diante da elegância que essa técnica nos proporciona.

Uma última técnica que iremos ensinar também pode ser utilizada com as classes ***Adapter*** para organizar mais ainda o seu código. Vamos ver na classe a seguir uma implementação do conceito de ***Inner Class***:

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

public class WindowDemo3 extends JFrame{
    public WindowDemo3(String title){
        super(title);
        this.setSize(350, 350);

        this.addWindowListener(new MinhaIClass());
        this.setVisible(true);
    }

    private class MinhaIClass extends WindowAdapter{
        public void windowClosing(WindowEvent e){
            fechar();
        }
    }

     public void fechar(){
         System.exit(0);
     }

     public static void main(String[] args) {
         new WindowDemo3("Exemplo de fechamento de janela 3, " +
                 "agora com ADAPTER e InnerClass");
     }
}
```

Como podemos ver, em vez de utilizarmos um ***adapter*** como parâmetro do método ***add***, agora passamos uma classe que está definida mais abaixo na nossa classe e, essa sim, ***estendendo*** um ***adapter***.

Mas pode isso? Isso mesmo, em Java podemos ter uma classe dentro da outra. Aliás, podemos ter quantas classes quisermos, mas somente uma pode ser pública e tem de ser aquela que leva o nome do arquivo .java. Isso organiza mais ainda o nosso código em algumas ocasiões.

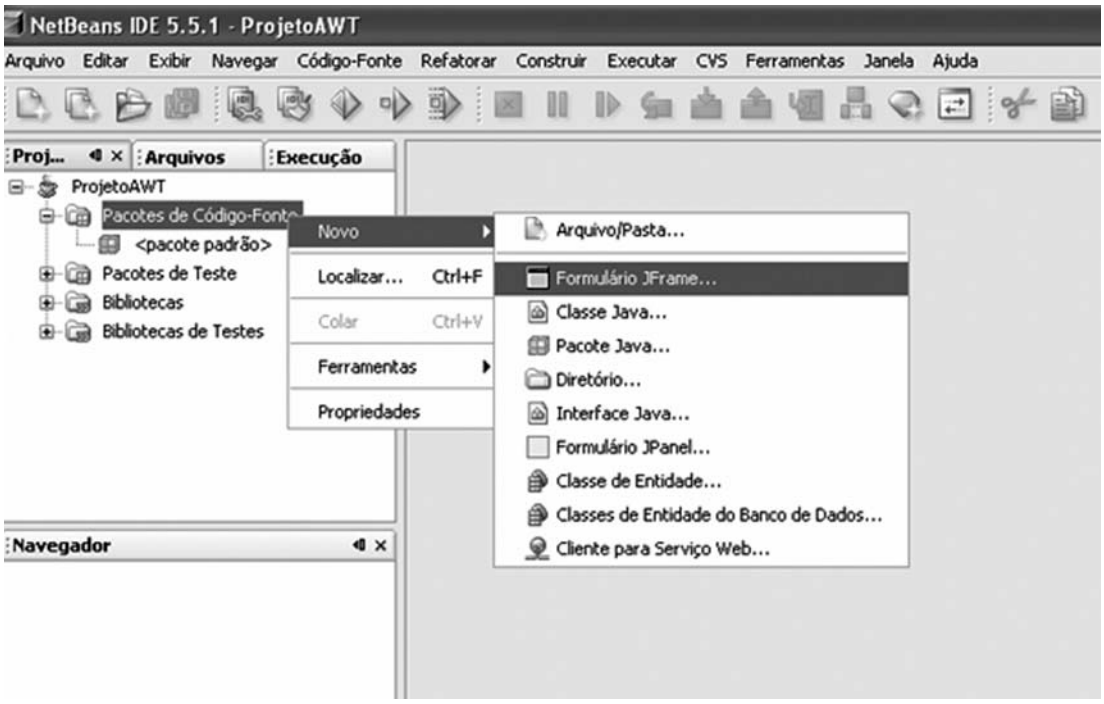
Atividade prática **2**

Implemente no seu projeto todas as classes que foram demonstradas anteriormente e execute-as prestando muita atenção nos conceitos que foram ensinados, tanto os de tratamento de eventos quanto os de organização de classes e código. Procure utilizá-los no seu dia a dia e mantenha seu código sempre limpo e elegante.

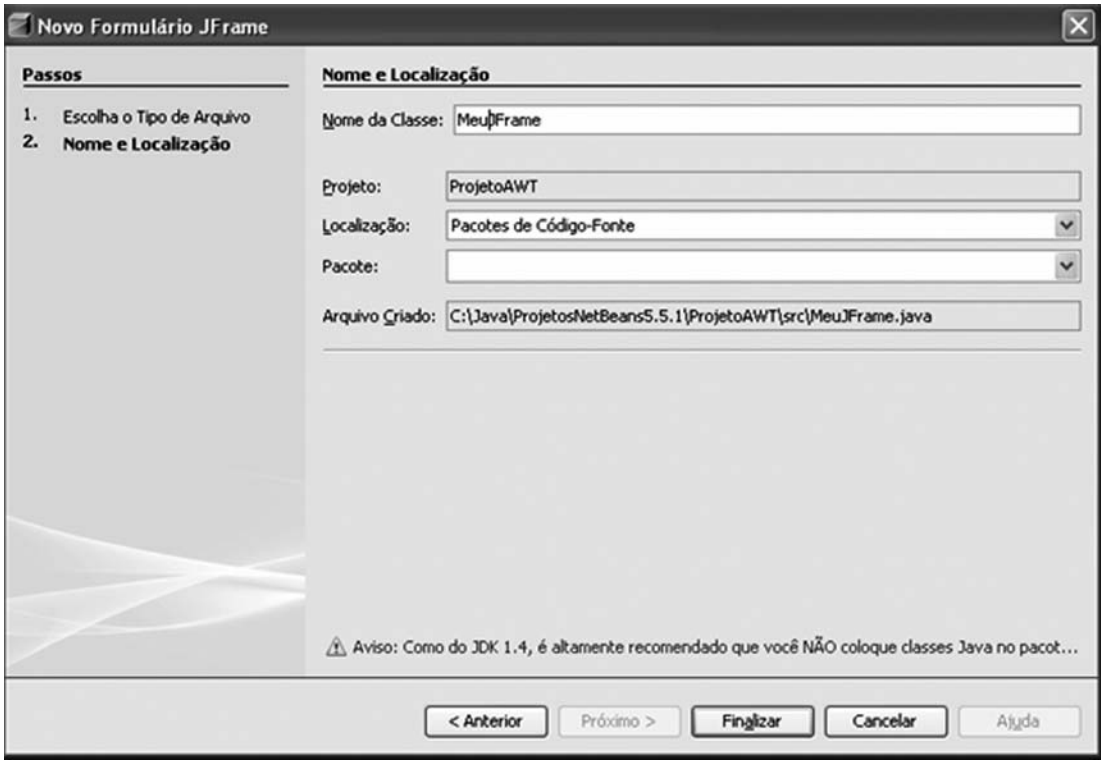
A IDE NETBEANS AUXILIANDO NA PROGRAMAÇÃO GRÁFICA

Vamos fechar nossa aula de programação gráfica ensinando um assunto bem legal e que pode auxiliar muito no desenvolvimento de janelas, principalmente no que se refere à construção dos **containers** e distribuição dos **componentes,** e faremos isso simplesmente arrastando-os de uma **paleta** lateral utilizando a IDE como **ferramenta RAD**.

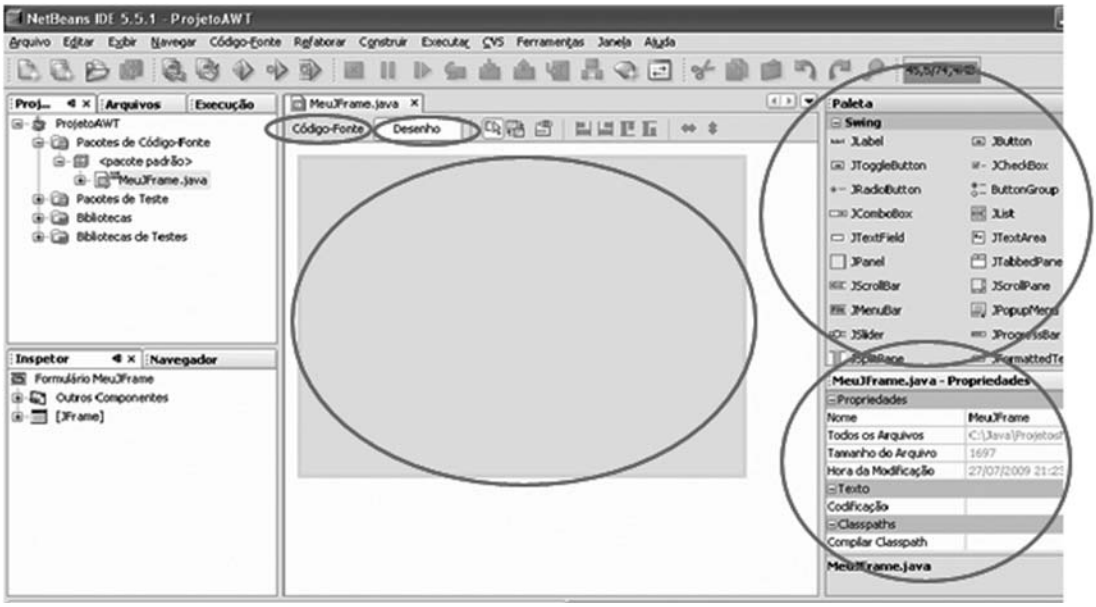
Num projeto comum Java, clique com o botão direito do mouse sobre **Pacotes de Código-Fonte** na área **Projetos** da janela do lado esquerdo da IDE e escolha **Novo => Formulário JFrame**, conforme a figura a seguir:



Depois dê um nome a sua classe e clique no botão Finalizar:



Veja que foi criada uma estrutura diferente na sua área de edição de código, composta agora de uma área de desenho já com o seu **JFrame** desenhado e ao lado direito uma **paleta** com vários componentes de **AWT** e **Swing** para serem arrastados para o **JFrame**:



Perceba nas cinco elipses destacadas as áreas mais importantes desta estrutura: o botão que apresenta a área de desenho, a área de desenho propriamente dita, o botão para mudar para o modo Código-Fonte, onde este é apresentado, a paleta com os componentes e a área das propriedades do componente selecionado.

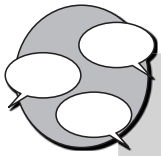
Veja a estrutura simples, mas rica em interfaces, que podemos montar em segundos com a IDE NetBeans como **RAD** AWT/Swing:

FERRAMENTA RAD (RAPID APPLICATION DEVELOPMENT)

ou Desenvolvimento Rápido de Aplicação (em português), é um modelo de processo de desenvolvimento de software interativo e incremental que enfatiza um ciclo de desenvolvimento extremamente curto (entre 60 e 90 dias). O termo foi registrado por James Martin em 1991 e tem substituído gradativamente o termo prototipação rápida, que já foi muito utilizado no passado.

Atividade prática 3

Utilizando a IDE NetBeans como ferramenta RAD, crie uma estrutura semelhante à da figura anterior, de um JFrame com componentes de envio de mensagem de e-mail. Além das áreas que foram explicadas, preste muita atenção na área chamada **Inspetor** no lado esquerdo inferior, onde é mostrada a árvore de todos os componentes utilizados na aplicação. Utilize a janela de propriedades para modificar os atributos dos elementos visuais.



INFORMAÇÕES SOBRE FÓRUM

Vamos entrar no fórum para discutir as dúvidas de vocês. Participem, troquem informações. Também não percam o ***chat*** da semana. O horário e o dia serão colocados na página do fórum.

RESUMO

Vamos rever os principais conceitos vistos nesta aula:

- **Tratamento de Eventos** é a técnica de capturar a ação que o usuário realizou por meio da GUI, interpretá-la identificando qual componente foi acionado e por último dar a resposta desejada à ação daquele componente, analisando, se for o caso, o conteúdo inserido pelo usuário por meio desse componente.
- **Graphical User Interface** é uma interface com o usuário que permite a interação com dispositivos digitais por meio de elementos gráficos como ícones, botões e outros indicadores visuais, em contraste com a interface de linha de comando.

- O **Modelo de Tratamento de Eventos** descreve o modo como sua classe pode responder a uma interação do usuário. Esse modelo é dividido em três componentes: **Gerador de Evento**, **Monitor de Evento / Manipulador** (***listener***) e ***Objeto Evento***.
- **Gerador de Evento** (***Event Source***): é o componente da interface que origina o evento. Por exemplo, caso o usuário pressione o botão para solicitar uma pesquisa, nesse caso o Gerador de Evento é o botão que foi clicado.
- **Monitor de Eventos/Manipulador** (***Event Listener/Handler***): recebe as informações enviadas pelo Gerador de Evento processando a interação. Por exemplo, no caso do usuário que apertou o botão, o Monitor de Eventos pode receber uma informação que foi enviada e retornar uma mensagem ao usuário relativa a essa recepção e tratamento.
- **Objeto Evento** (***Event Object***): quando um evento ocorre, o pressionar de um botão, por exemplo, um Objeto Evento é criado. Ele contém todas as informações necessárias sobre o evento gerado. Um exemplo seria o clicar do botão direito do mouse sobre um botão.
- Quando programamos um evento que pode ocorrer, um ***listener*** (**Monitor de Eventos**) deve ser registrado pelo **Gerador de Evento**, pois assim esse ***listener*** irá receber as informações do evento que ocorrer nesse **Gerador de Evento**. Uma vez registrado, esse ***listener*** fica aguardando a geração de evento sobre esse componente gerador do evento.
- Quando algum evento ocorre no gerador do evento, um **Objeto Evento** é criado, ele é o descritor do evento. Após isso, o evento é disparado pelo gerador para os ***listeners*** registrados.
- Quando um ***listener*** recebe uma notificação de um gerador de evento, ele executa sua função, interpretando e processando o evento ocorrido.
- Resumidamente, os passos e ações necessárias para criar uma aplicação gráfica GUI com tratamento de eventos são:
    1. Crie a parte visual de sua aplicação, a **GUI** propriamente dita com seu container e seus componentes de botões, textfields, combos, listas etc.;

2. Crie uma classe que implemente a interface ***listener*** apropriada. Normalmente o código deste passo pode estar incluído na classe do passo 1.
3. Na classe criada no passo 2, sobrescrever todos os métodos da interface ***listener*** escolhida, sendo que, caso exista algum método que não vá tratar o evento, basta deixá-lo vazio.
4. Registre o objeto listener no **Gerador de Evento** através do método **add<TipoDoListener>Listener.**

Apêndice A

Meta da aula

Criar e configurar um ambiente de desenvolvimento de aplicações Java.

objetivos

Após o estudo do conteúdo deste apêndice, esperamos que você seja capaz de:

1. efetuar o *download* do Java Development Kit (JDK);
2. realizar a instalação do Java Development Kit (JDK);
3. realizar as configurações das variáveis de ambiente do sistema operacional, necessárias para o desenvolvimento dos programas em Java.

INTRODUÇÃO

Neste apêndice, você será acompanhado nas atividades iniciais necessárias para a instalação e configuração do seu ambiente de desenvolvimento Java.

Isso é muito importante, pois somente com muita prática você vai ficar seguro para desenvolver aplicações cada vez mais profissionais, até se tornar um grande desenvolvedor Java, e para isso você precisa ter um ambiente de desenvolvimento adequado e corretamente configurado.

Vamos antes explicar alguns conceitos importantes.

A linguagem Java tem como uma das principais características não realizar execução de programas diretamente na máquina do computador, como ocorre com C e Pascal, mas sim numa máquina intermediária chamada **MÁQUINA VIRTUAL JAVA (JVM – JAVA VIRTUAL MACHINE)**.

MÁQUINA VIRTUAL JAVA (JVM – JAVA VIRTUAL MACHINE)

É um programa que carrega e executa os aplicativos Java, convertendo os **bytecodes** em código executável de máquina. A **JVM** é responsável pelo gerenciamento dos aplicativos à medida que são executados. Graças à Máquina Virtual Java, os programas escritos em Java podem funcionar em qualquer plataforma de *hardware* e *software* que possua uma versão da JVM, tornando essas aplicações independentes da plataforma em que funcionam.

Sendo assim, tanto para você que está desenvolvendo o aplicativo em Java, quanto para aquele que está apenas utilizando-o, será necessária a instalação da JVM, além de mais algumas tarefas, para que o aplicativo funcione corretamente.

Para o usuário final (aquele que apenas utilizará o programa escrito em Java), será necessário a JVM, além de um conjunto de aplicativos e bibliotecas para interpretar corretamente uma aplicação. A esse conjunto mínimo que deve ser baixado e instalado no computador de quem está utilizando um aplicativo Java damos o nome de **Java Runtime Environment** (**JRE**). Uma importante conclusão é de que não faz sentido (nem existe fisicamente para *download*) a instalação unicamente da **JVM**; como foi informado, o ambiente mínimo para a execução de um aplicativo Java deve ter instalado o **JRE**.

Além do **JRE**, você deverá instalar uma série de programas que irão auxiliar no desenvolvimento, como o compilador Java (**javap**), o compactador (**jar**) e o gerador de documentação (**javadoc**), entre outros, além das bibliotecas de desenvolvimento. A esse conjunto necessário para você trabalhar damos o nome de *Java Development Kit (JDK)*. Quando fazemos o *download* e instalamos o **JDK**, já baixamos automaticamente e realizamos a instalação do **JRE** (não é necessário ser feito separadamente). Para efeito de comparação, na versão mais atual dos pacotes o **JRE** tem por volta de **16Mb**, enquanto o **JDK** tem em torno de **75Mb**.

DOWNLOAD DO JAVA DEVELOPMENT KIT (JDK)

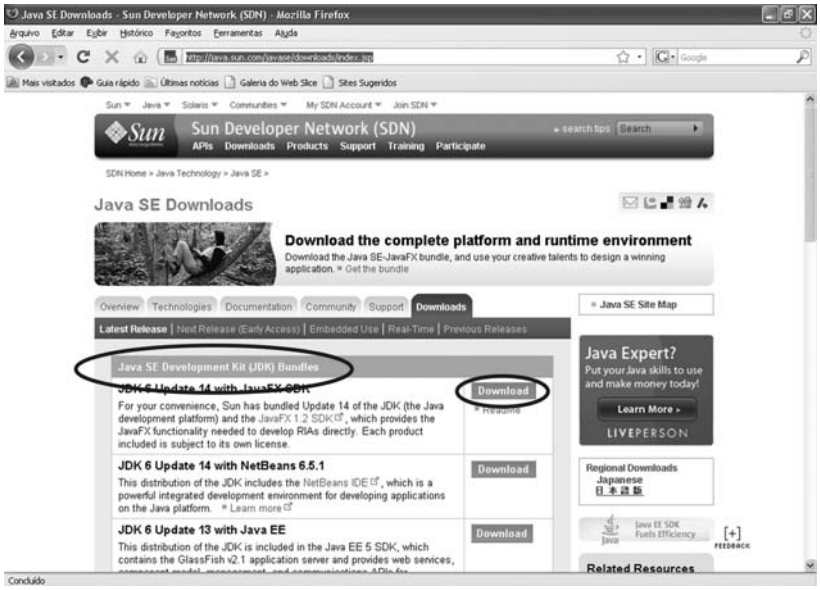
Comece então a preparar o ambiente para o desenvolvimento de sistemas em Java. A primeira coisa é realizar o *download* do JDK, preferencialmente na sua versão mais atual; assim, você terá instalado a JVM além dos aplicativos e bibliotecas necessários ao desenvolvimento.

Então, vamos lá! No seu computador, faça o *download* do JDK diretamente do *site* da Sun Microsystems em:

http://java.sun.com/javase/downloads/index.jsp

Passo 1

Nesta página, clique no botão **Download** no item Java SE Development Kit (JDK):



Passo 2

Selecione então a plataforma e aceite as condições de licença:

Passo 3

Selecione o arquivo e clique no *link* azul do executável:

Passo 4

Na janela que irá se abrir, clique para salvar o arquivo:

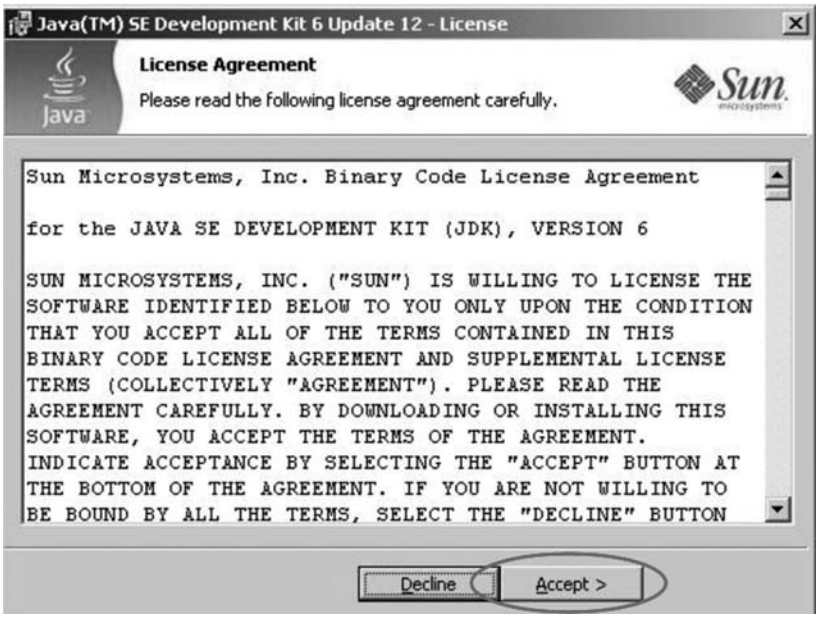
INSTALAÇÃO DO JAVA DEVELOPMENT KIT (JDK)

Passo 1

Vá ao local onde o **JDK** foi baixado e dê um duplo clique no ícone para iniciar o executável da instalação:

Passo 2

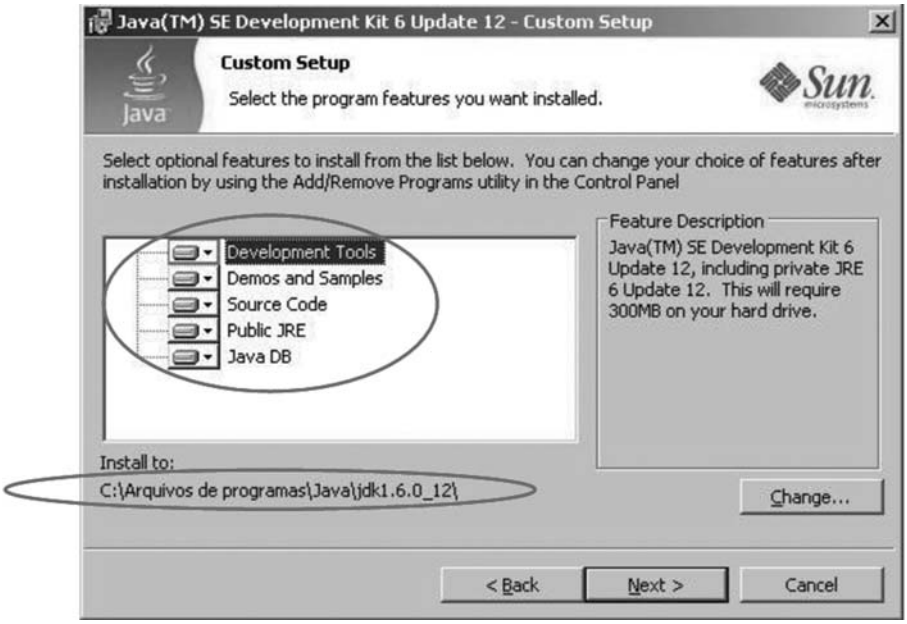
Será aberta uma tela que faz a verificação do sistema. Depois, passa automaticamente para a tela que apresenta as condições de licença do **JDK**. Clique no botão **Accept**:



Passo 3

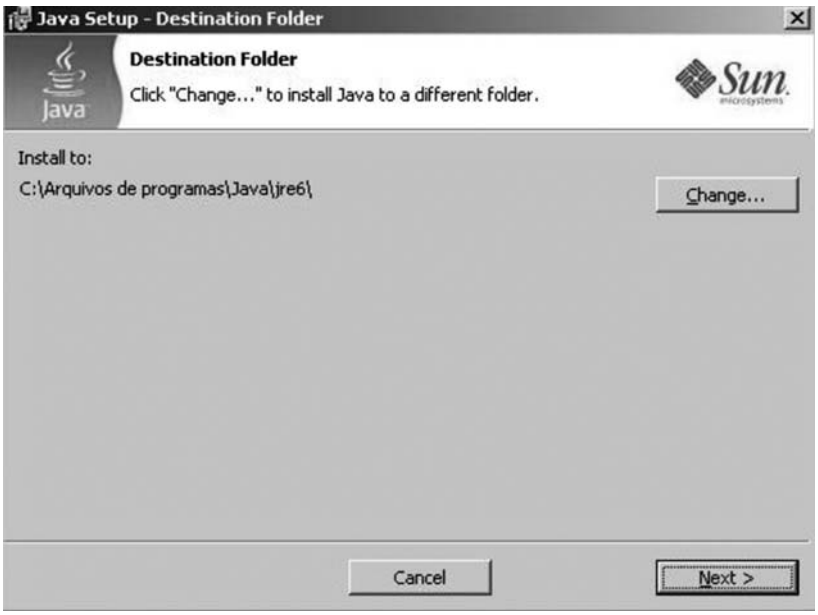
Na tela seguinte, você irá definir sua configuração do sistema; ali poderá selecionar os pacotes que serão instalados (aconselhamos deixar a configuração padrão, que instala todos os pacotes) e o local de instalação do Java.

Memorize o local de instalação do Java, pois esse endereço será utilizado na configuração das variáveis de ambiente.



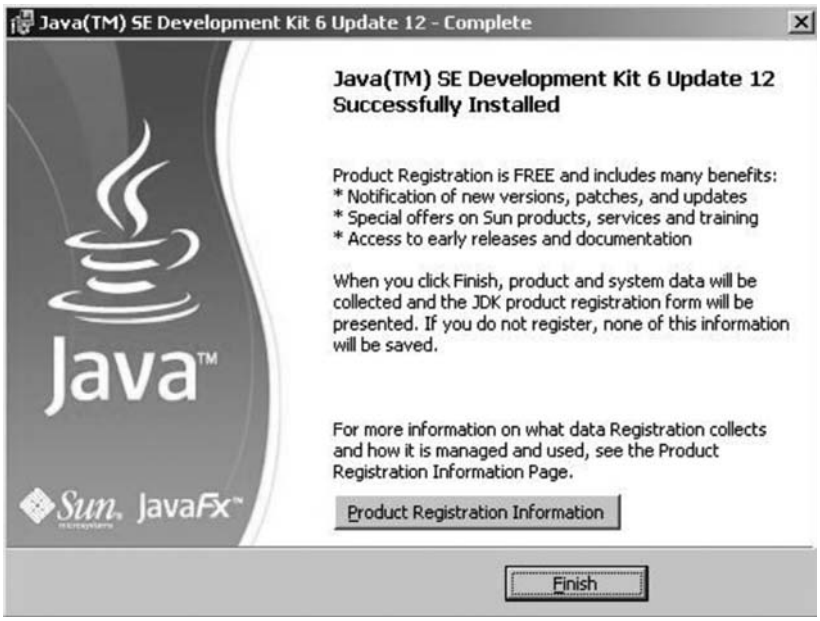
Após isso, clicando no botão **Next** a instalação propriamente dita será iniciada; aguarde alguns minutos.

Como foi explicado na Introdução, a instalação do **JRE** é feita dentro da instalação do **JDK**. Nesse momento você verá uma tela como a que está a seguir (caso queira, modifique o local de instalação):



Passo 4

Aguarde mais alguns minutos. Ao clicar novamente em **Next** e ao ser finalizada a instalação, você verá uma tela como a que está a seguir, o que significa que deu tudo certo:



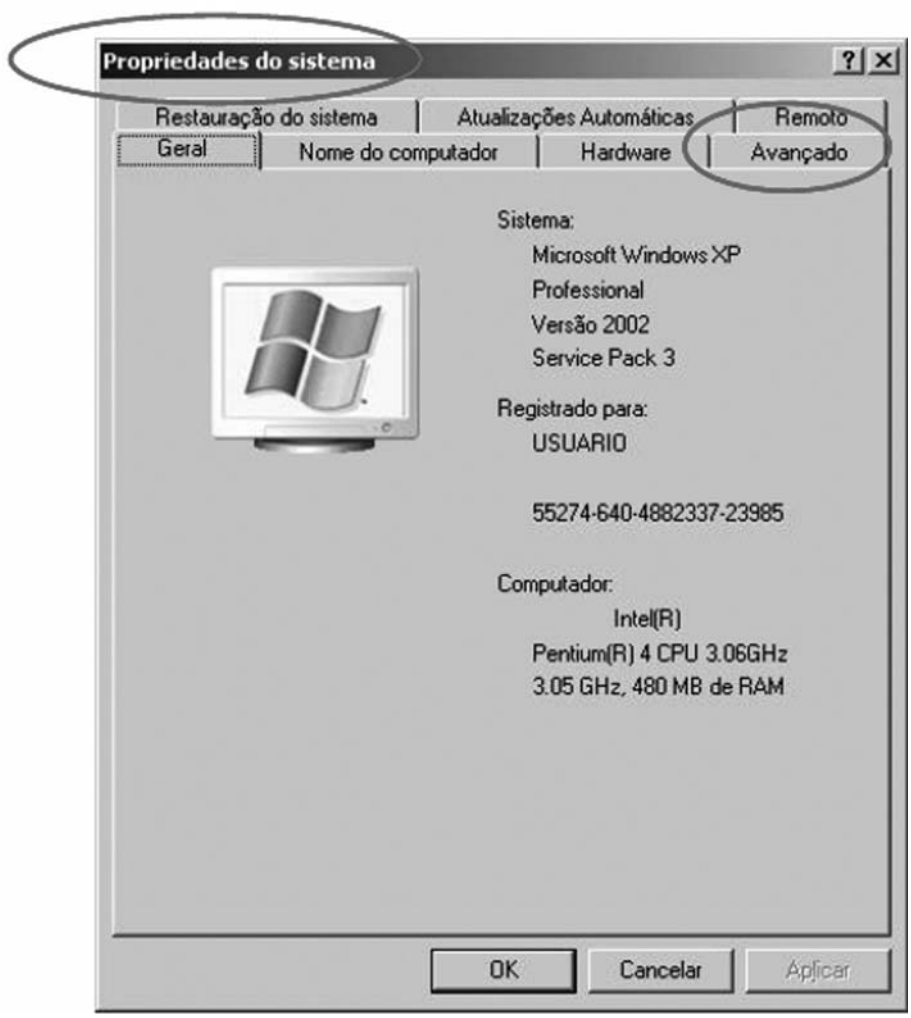
CONFIGURAÇÃO DAS VARIÁVEIS DE AMBIENTE

Agora que você já baixou e instalou o JDK, ainda falta uma última tarefa. Configure as variáveis de ambiente do sistema operacional para que ele possa encontrar os caminhos dos aplicativos necessários para executar o programa em Java, além das classes que são carregadas em tempo de execução.

Vamos fazer a configuração considerando o sistema operacional Windows XP Professional SP2.

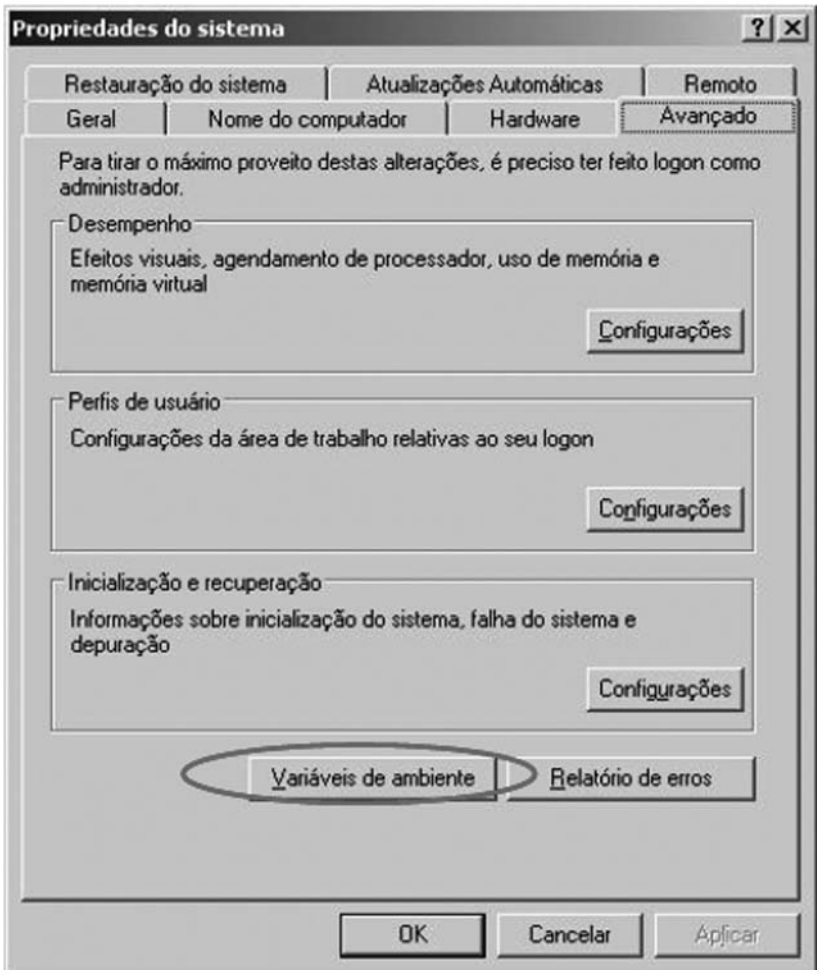
Passo 1

Clique nos botões *Iniciar* => *Configurações* => *Painel de Controle* e dê um duplo clique no ícone *Sistema* para chegar até a janela *Propriedades do sistema*.

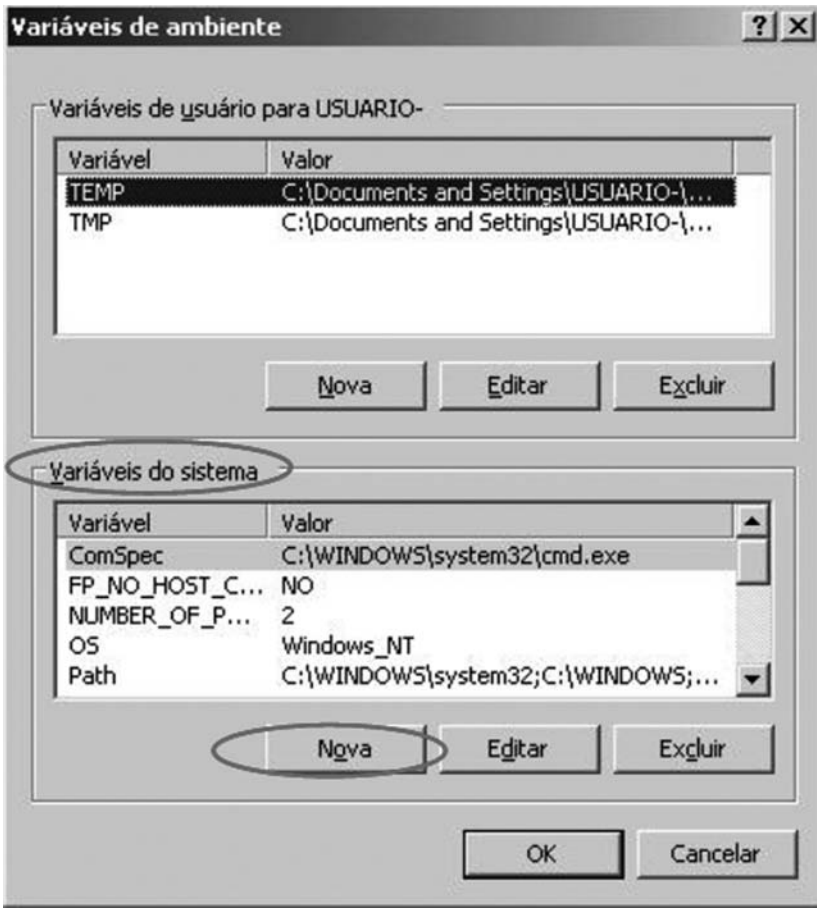

Propriedades do sistema
Restauração do sistema
Atualizações Automáticas
Remoto
Geral
Nome do computador
Hardware
Avançado
Sistema:
Microsoft Windows XP
Professional
Versão 2002
Service Pack 3
Registrado para:
USUARIO
55274-640-4882337-23985
Computador:
Intel(R)
Pentium(R) 4 CPU 3.06GHz
3.05 GHz, 480 MB de RAM
OK
Cancelar
Aplicar

Passo 2

Nessa janela, selecione a aba *Avançado* e, na parte inferior dela, selecione *Variáveis de ambiente*

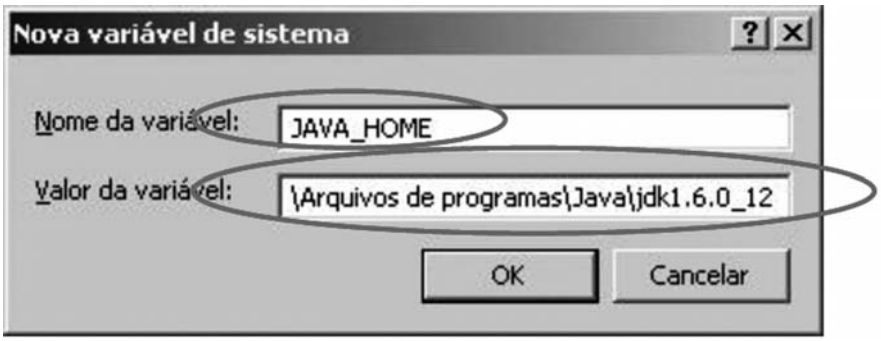


Veja que nessa janela temos as variáveis configuradas para o usuário corrente e para o sistema, que serão aplicadas a todos os usuários. Prefira configurar as variáveis do sistema, a menos que tenha alguma regra de segurança na sua rede que impeça.

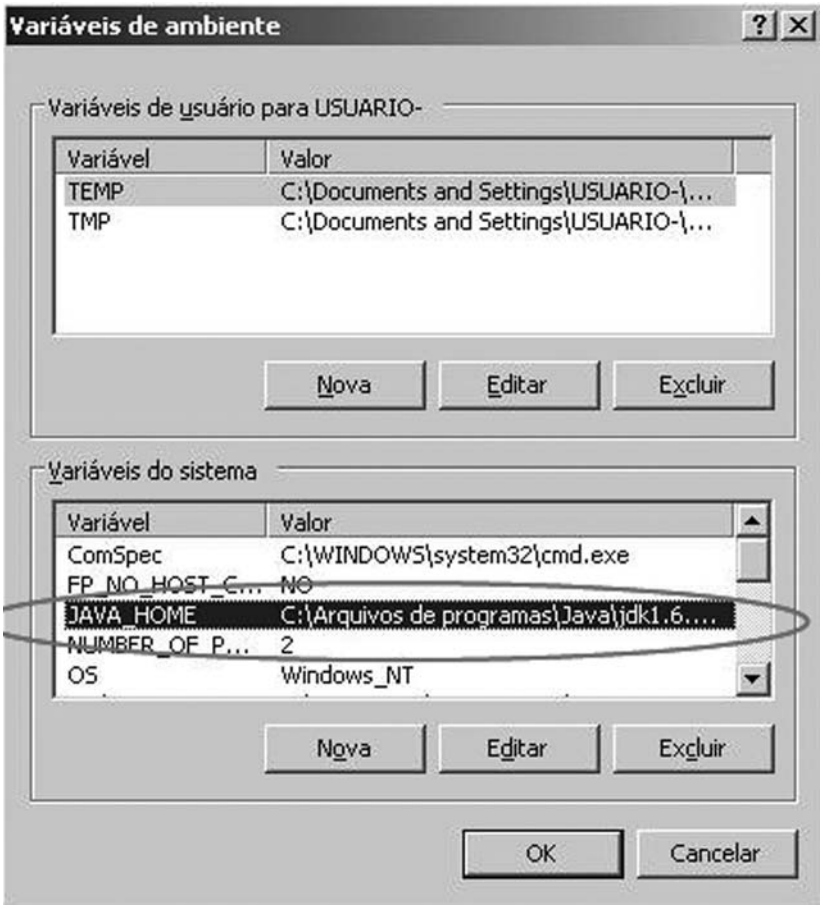


Passo 3

O primeiro passo, nessa janela, é a criação de uma variável do sistema que, por convenção, recebe o nome de **JAVA_HOME**, que, como o próprio nome diz, é o local de instalação do **JDK**. Caso tenha aceitado as configurações padrão do sistema, essa variável terá o valor: **C:\Arquivos de programas\Java\jdk1.6.0_12** (onde 1.6.0_12 é a versão do JDK que foi instalada). Para criá-la, clique no botão "Nova" e insira o nome da variável e seu respectivo valor na janela que irá se abrir.



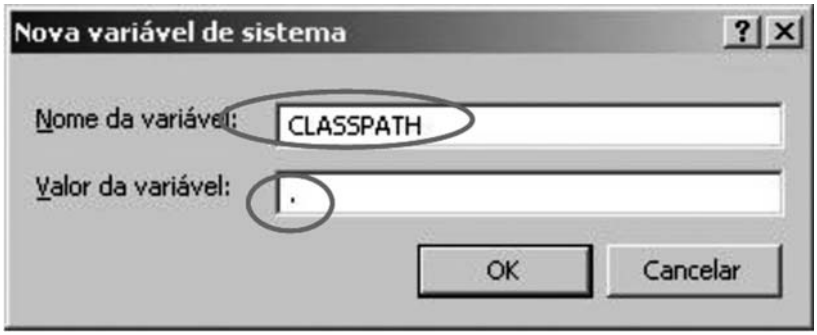
Após preencher os campos, clique no botão **OK** e veja se a variável foi criada como variável do sistema.



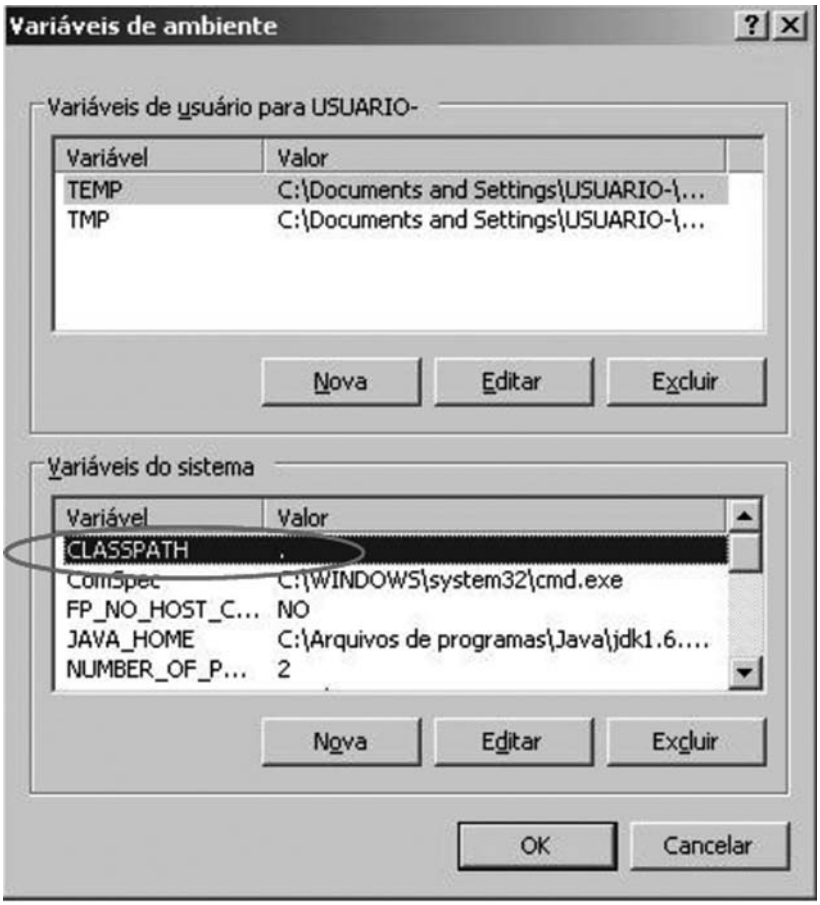
Passo 4

Após isso, você criará outra variável do sistema, chamada **CLASSPATH**, cujo valor é unicamente um ponto: **"."**

Para isso, clique mais uma vez no botão "Nova" e insira os valores dos campos conforme a figura a seguir:

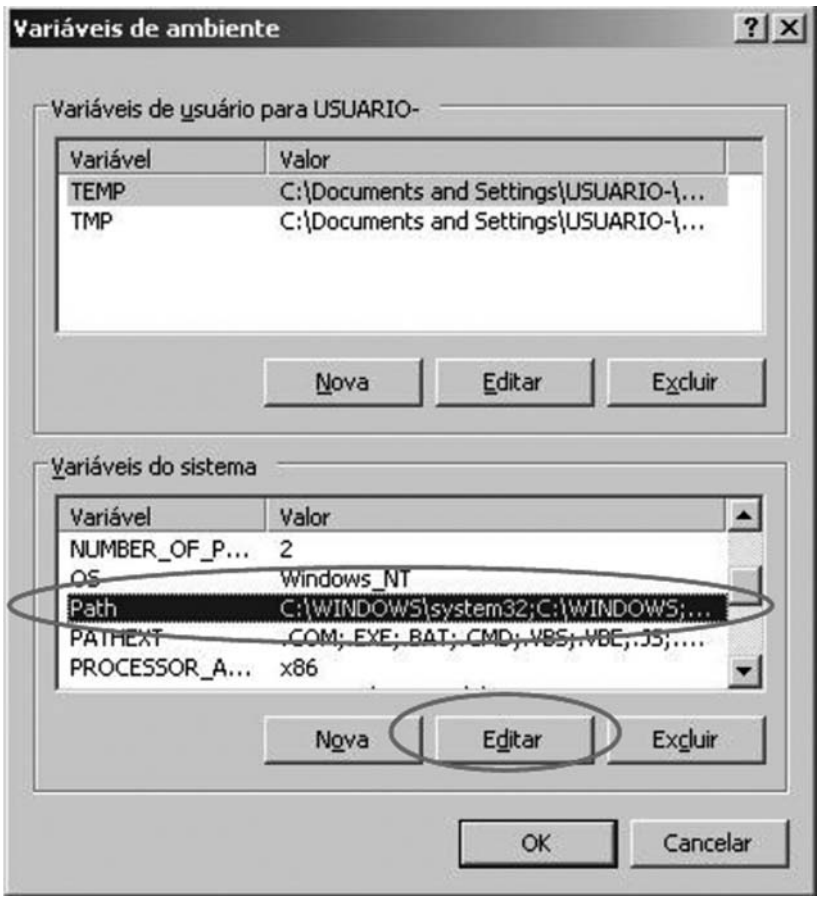


Clique no botão **OK** e veja se a variável criada aparece corretamente nas variáveis do sistema:



Passo 5

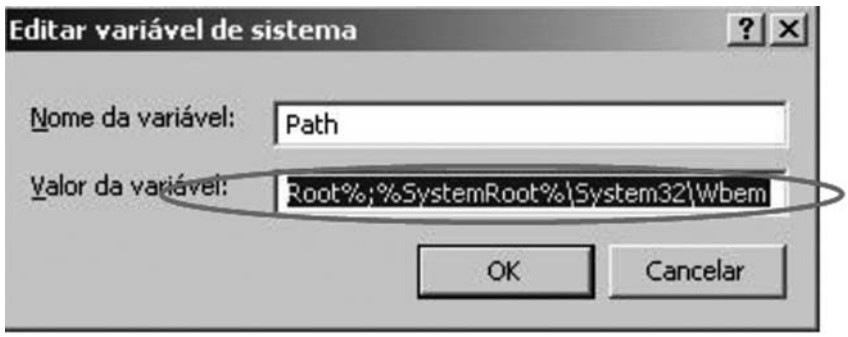
A última variável não será criada, mas sim editada, uma vez que já foi criada por ocasião da instalação do Windows. Essa variável é a *Path*, e para isso você deve selecioná-la e em seguida clicar no botão "Editar".

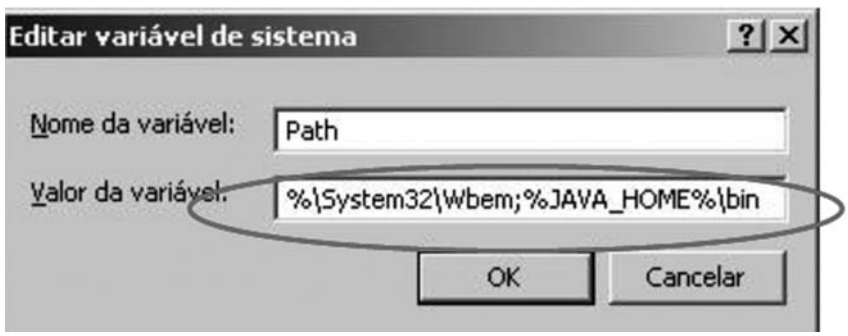


Aparecerá a variável já existente e com vários valores já preenchidos.

Passo 6

Cuidadosamente, vá com o cursor até o fim do campo "Valor da variável", insira um **";"** seguido de **%JAVA_HOME%\bin** e clique em **OK**.





Clique em **OK**. Está finalizada a configuração das variáveis de ambiente no Windows XP Professional SP2. Se tiver dúvidas, vá ao fórum da disciplina e solicite esclarecimento ao tutor.

TESTANDO A INSTALAÇÃO E AS CONFIGURAÇÕES

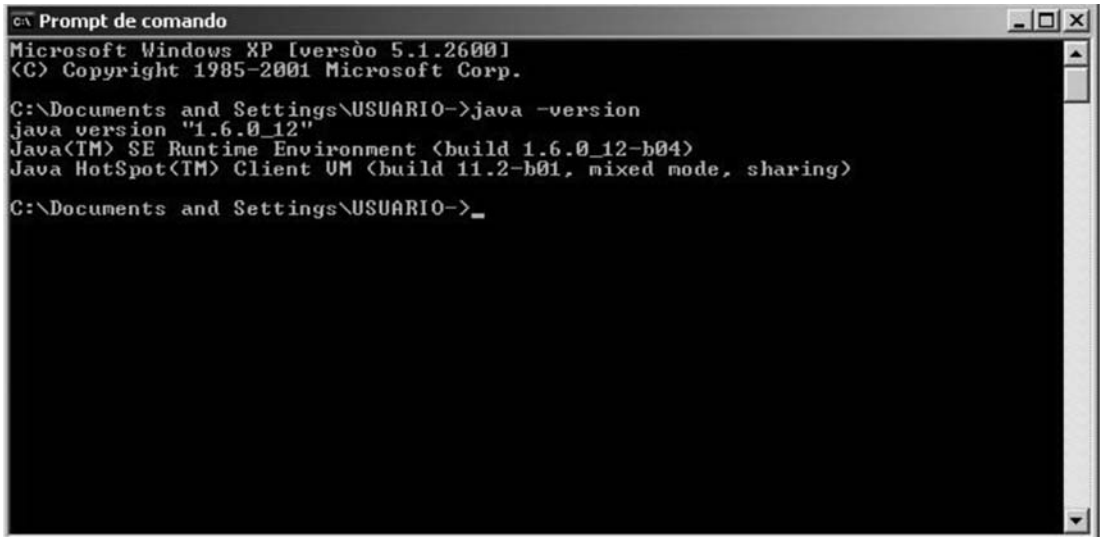
Vamos ver se deu realmente tudo certo por meio de dois testes bem simples.

Passo 1

Abra o Prompt de Comando por meio da sequência *Iniciar* => *Programas* => *Acessórios* => *Prompt de comando*.

Passo 2

Digite o comando *java –version* e tecle *Enter*. Aparecerá a versão do Java como na figura a seguir:



Passo 3

Digite agora o comando javac –version e tecle *Enter* novamente. Aparecerá a versão do compilador Java como na figura que se segue:

Se os dois testes funcionaram no seu computador, isso significa que você já está com o ambiente instalado e configurado para iniciar o desenvolvimento de sistemas em Java. Parabéns!

Caso tenha ocorrido algum problema, reveja cuidadosamente os passos dos itens anteriores para verificar se deixou de efetuar alguma configuração corretamente. Se alguma dúvida persistir, vá ao fórum da disciplina para esclarecê-la.

Agora que você já tem os *softwares* do Java instalados e configurados, você pode começar a programar, seja da forma mais simples, por meio de um editor de texto, ou de um ambiente mais sofisticado, como uma IDE. Aliás, você deve estar se perguntando o que é uma IDE.

Todos esses detalhes você verá no **APÊNDICE B**. Por enquanto, retorne ao ponto onde havíamos parado na Aula 1 e continue seus estudos.

Apêndice B

Metas da aula

Criar e configurar um ambiente de desenvolvimento de aplicações Java.

objetivos

Após o estudo do conteúdo deste apêndice, esperamos que você seja capaz de:

1. efetuar o *download* do Ambiente de Desenvolvimento Integrado (IDE) NetBeans;
2. realizar a instalação do NetBeans;
3. criar um projeto e testá-lo por meio da IDE.

INTRODUÇÃO

Ao ter desempenhado as atividades do Apêndice A, você instalou e configurou o JDK. Agora, vamos terminar os trabalhos de montagem do nosso ambiente para desenvolvimento Java. Está pronto?

Ao escrever programas em Java, você pode utilizar qualquer editor de texto, como Bloco de Notas, Wordpad, Vi, Emacs etc. Entretanto, se você deseja ter mais facilidade e produtividade, pode utilizar ferramentas mais apropriadas para a programação, as chamadas **IDEs (Integrated Development Environment)** ou **Ambiente de Desenvolvimento Integrado**. Tais ferramentas possuem várias funcionalidades para auxiliá-lo no preenchimento automático de código, editor de código, *wizards* para criação automática de classes, controle de versões, ferramentas de modelagem, conexão com banco de dados, gerenciamento de servidores e muitas outras funcionalidades que auxiliam na programação.

Se você for um programador iniciante, deve ter cuidado para que a utilização de IDEs não prejudique o aprendizado da tecnologia. Assim, utilize bem as facilidades das ferramentas para que isso possa auxiliá-lo a ganhar produtividade na correção e debug dos códigos, mas nunca como uma máquina de produzir programas.

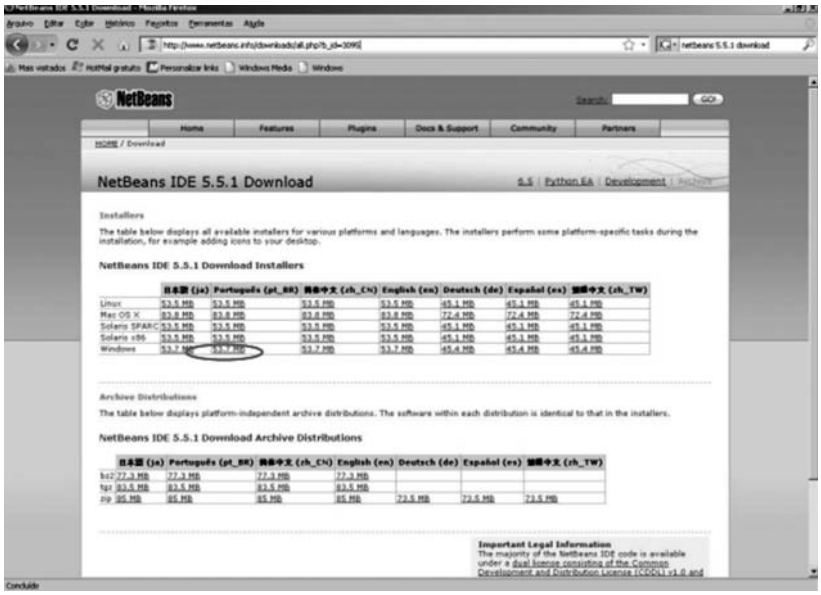
Existem muitas opções de escolha de IDEs, algumas pagas, outras gratuitas. Os mais famosos e utilizados são **Eclipse** e **NetBeans**, ambos gratuitos. Apesar de serem de empresas diferentes, seguem a mesma linha no desenvolvimento de suas funcionalidades, e a curva de aprendizado quando se sai de uma IDE para outra não é tão grande. Neste curso, iremos utilizar a IDE oficial da Sun, o famoso NetBeans, que iremos estudar melhor nas próximas sessões.

DOWNLOAD DO NETBEANS

Primeiramente, você precisa fazer o *download* do NetBeans diretamente do *site*. Existem diversas versões; a mais nova é a 6.5. As versões atuais estão muito grandes e pesadas devido à quantidade de pacotes que já vêm na plataforma, mas que muitas vezes não são importantes para o iniciante e mesmo para o desenvolvedor intermediário; sendo assim, aconselhamos fazer o *download* de uma versão mais leve e estável, suficiente para realizar todos os trabalhos sem comprometer a performance do computador utilizado. Vamos, então, utilizar o **NetBeans 5.5.1**.

Passo 1

No seu computador, acesse http://www.netbeans.info/downloads/all.php?b_id=3095 e escolha a versão em português, de acordo com seu sistema operacional. No exemplo, optamos pelo Windows. Aguarde a janela de *download* do arquivo e escolha **Salvar**.

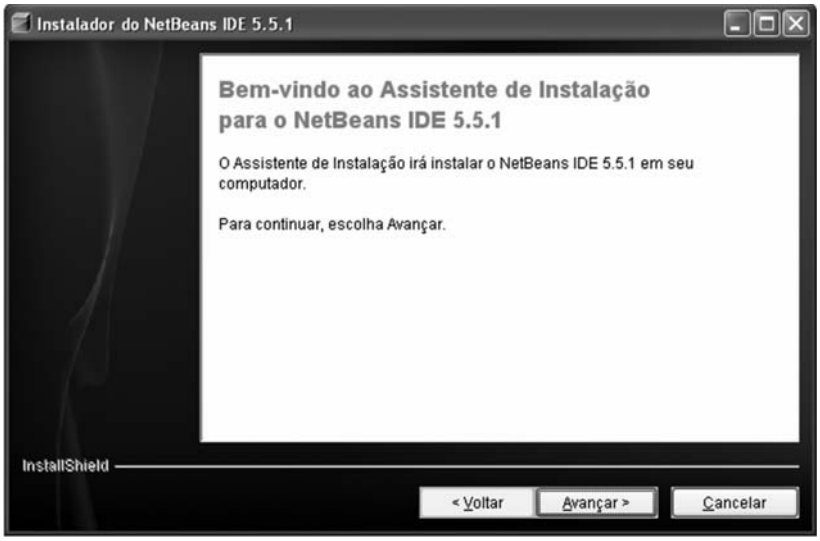


INSTALAÇÃO DO NETBEANS

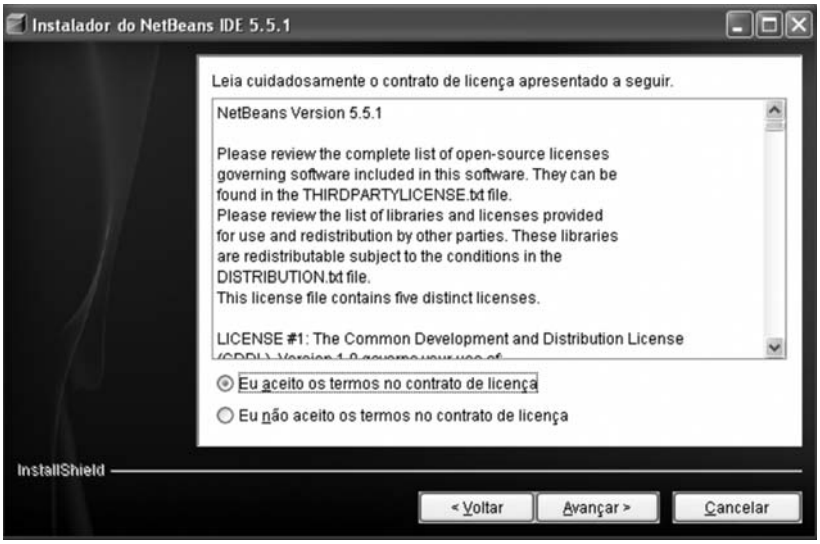
Passo 2

Uma vez feito o *download*, vá ao local onde está o executável e dê um duplo clique no ícone do programa:

Irá aparecer a tela inicial do instalador do NetBeans. Clique em **Avançar:**

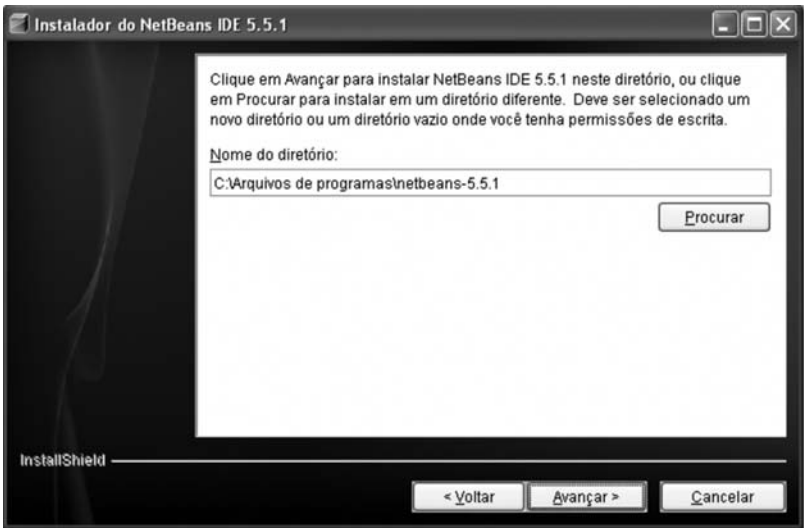


Aceite os termos de contrato da licença:



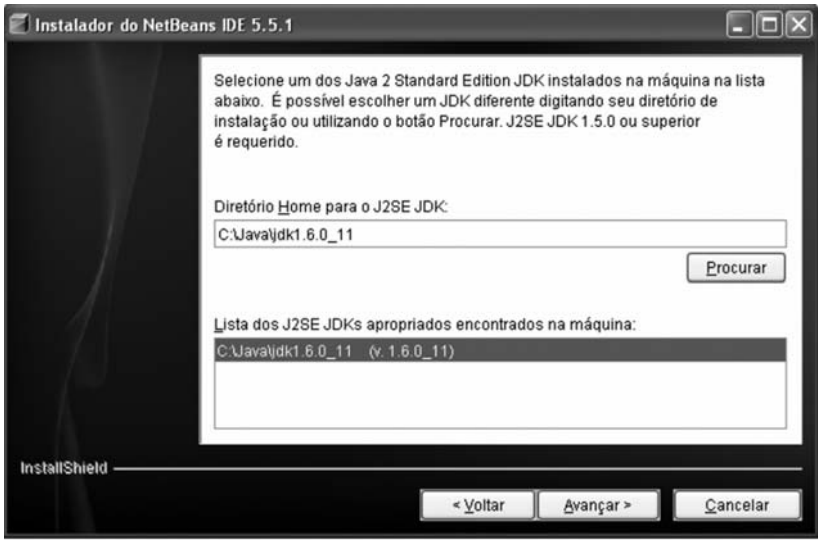
Passo 3

A próxima tela apresenta o local de instalação da ferramenta. Caso queira alterar o local, clique em **Procurar**; quando tiver escolhido o local, clique em **Avançar**:



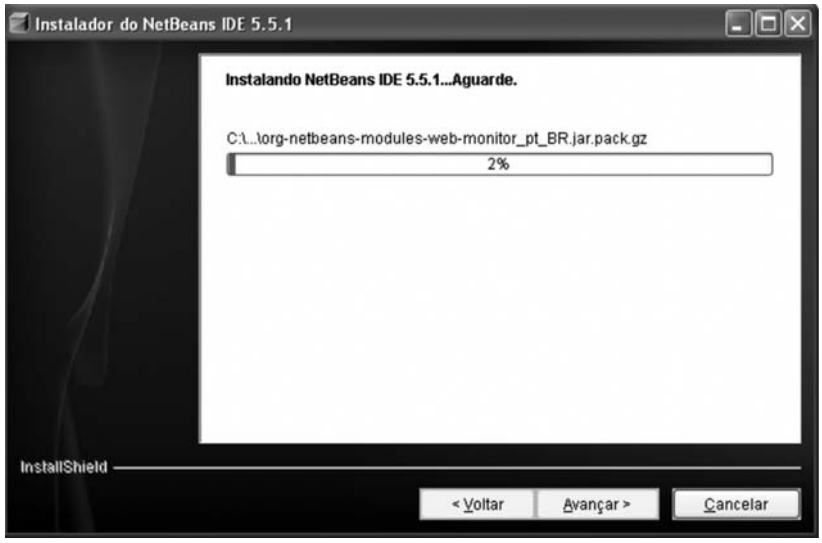
Passo 4

Na próxima tela, escolha o JDK mais adequado ao seu desenvolvimento, caso tenha mais de um instalado no seu computador. Clique então em **Avançar**:

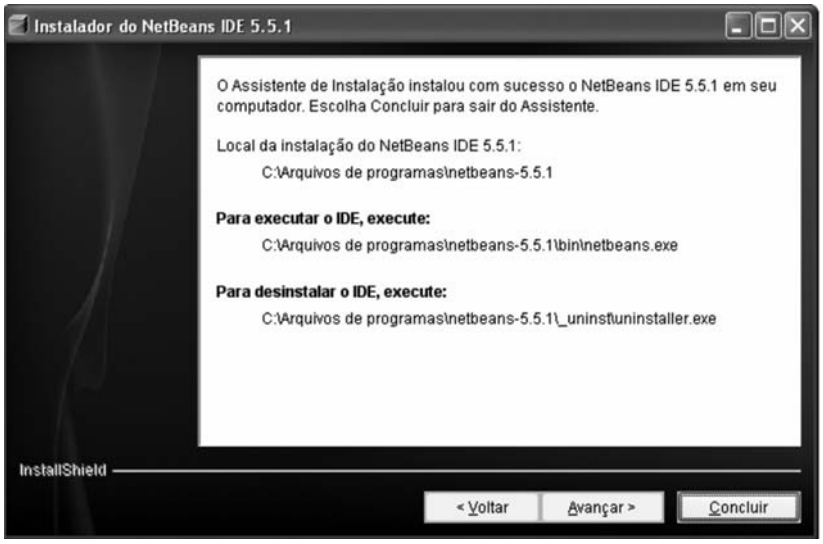


Aparecerá então a última tela, com a confirmação de alguns dados antes de iniciar a instalação propriamente dita:

Aguarde a instalação.



Se tudo der certo, você verá a tela final da instalação. Basta agora clicar em **Concluir**:

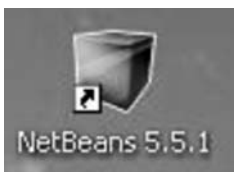


CRIAÇÃO E TESTE DE PROJETO

Agora que você tem seu ambiente configurado e com a IDE instalada, vamos testá-la criando um primeiro projeto e executando a primeira classe.

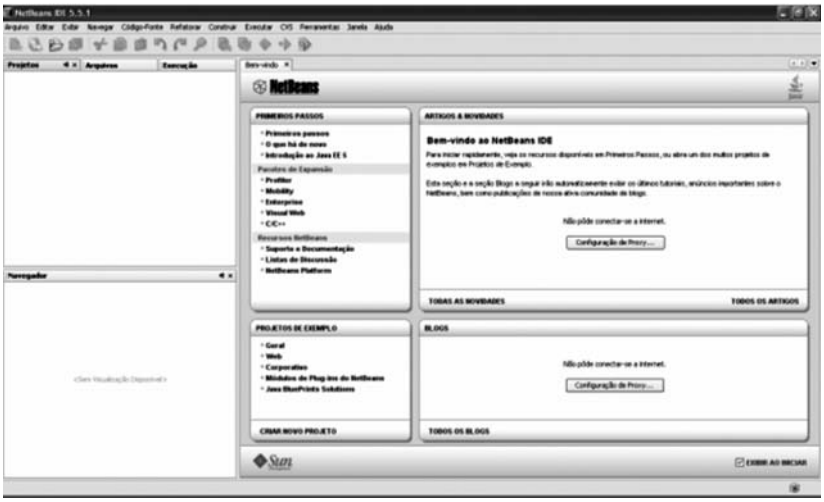
Passo 1

Para inicializar o NetBeans, dê um duplo clique no ícone que foi criado na área de trabalho após a instalação ou inicialize a ferramenta por meio do caminho *Iniciar* => *Programas* => *NetBeans 5.5.1* => *NetBeans IDE*:



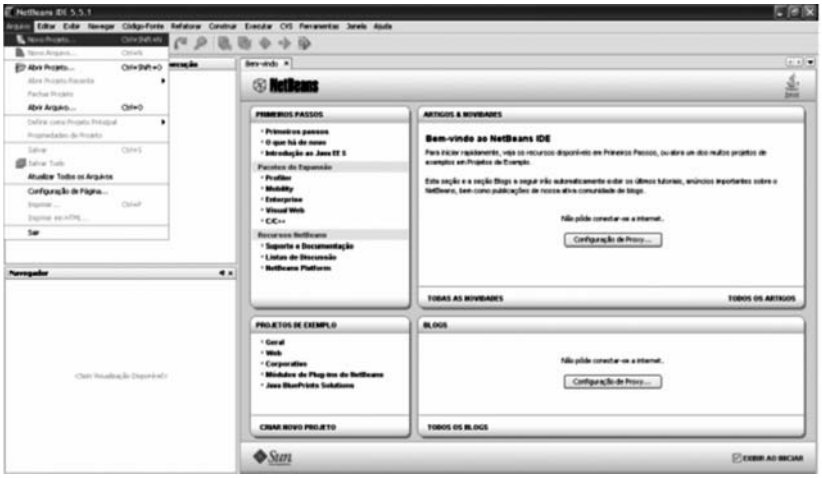
Aguarde a abertura da IDE (enquanto é carregada, aparece a tela a seguir):

Será então mostrado o ambiente gráfico da ferramenta com suas diversas áreas, como mostrado na figura a seguir.

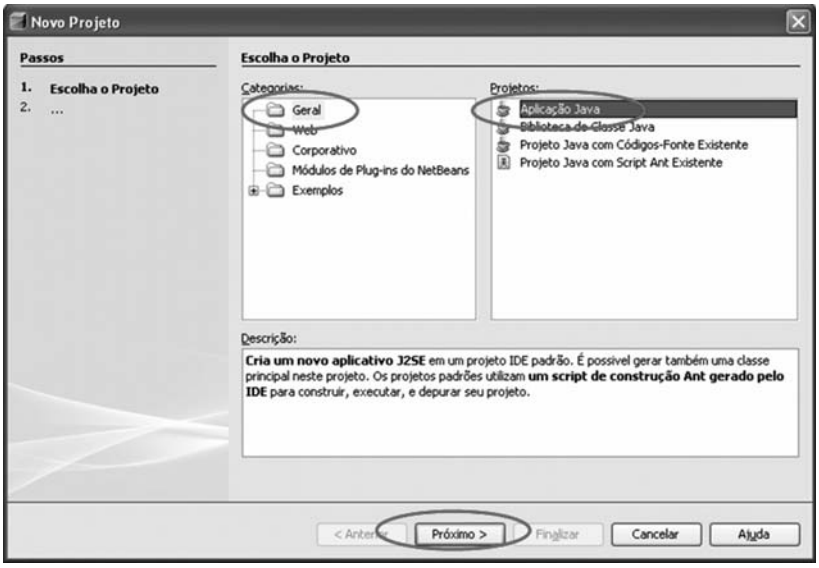


Passo 2

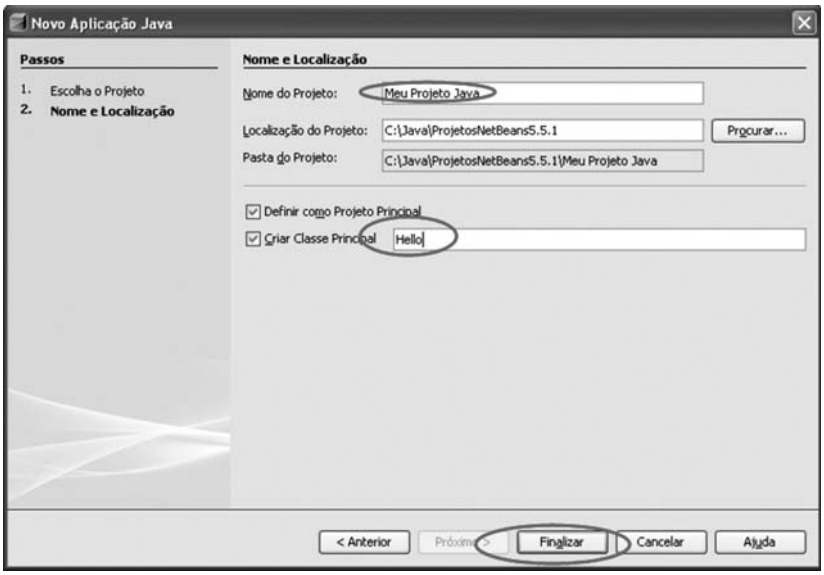
Em toda IDE, qualquer programa em Java deve estar associado a um projeto. Crie então o primeiro projeto, selecionando *Arquivo* => *Novo Projeto*.



Na caixa de diálogo que abrir, escolha na janela *Categoria* a opção *Geral* e na janela *Projetos* a opção *Aplicação Java*.

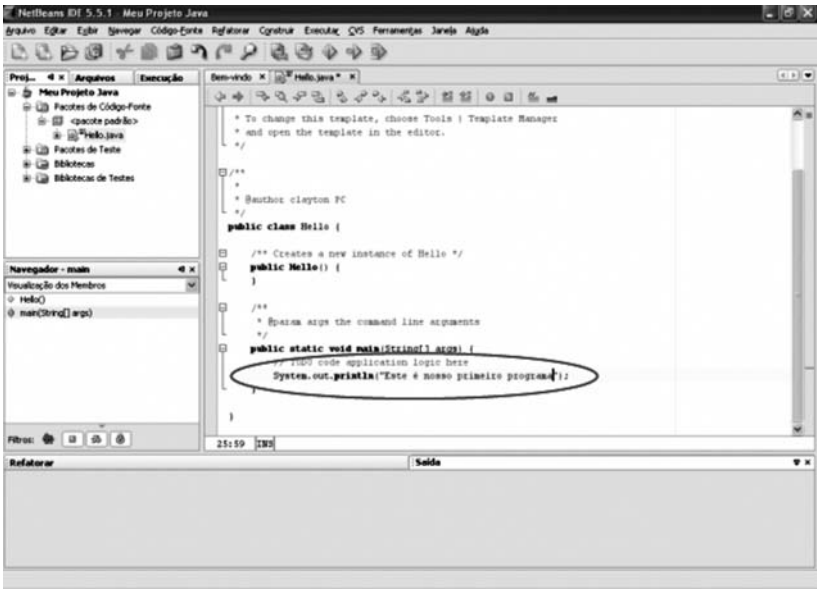


Clique no botão **Próximo** e na caixa de diálogo seguinte preencha o *Nome do Projeto* com **Meu Projeto Java** e, ao lado da *checkbox Criar Classe Principal*, defina seu nome como ***Hello*** e clique no botão **Finalizar**.



Na janela que se abrir, a IDE insere um código básico de todo programa Java, completado com algumas informações inseridas durante a criação do projeto. Complete esse código com a linha a seguir, conforme mostrado na figura:

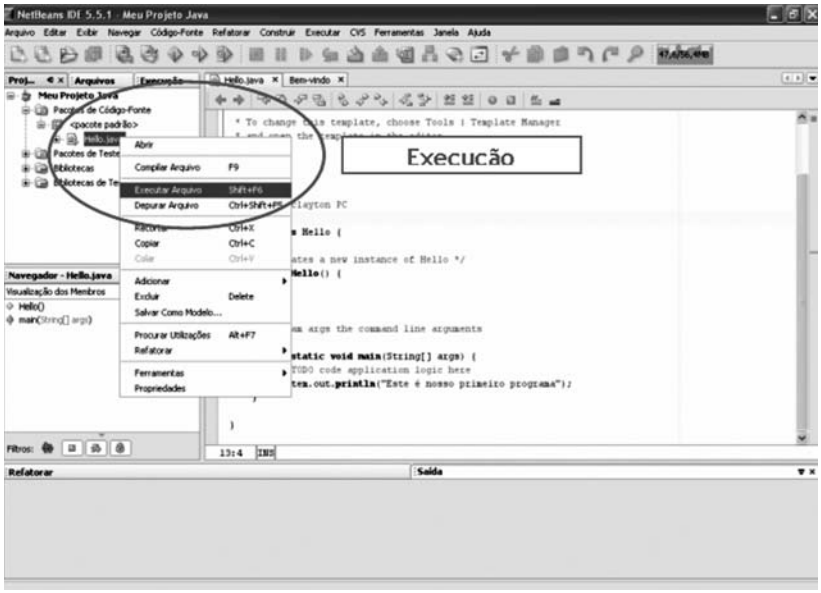
System.out.println("Este é nosso primeiro programa");

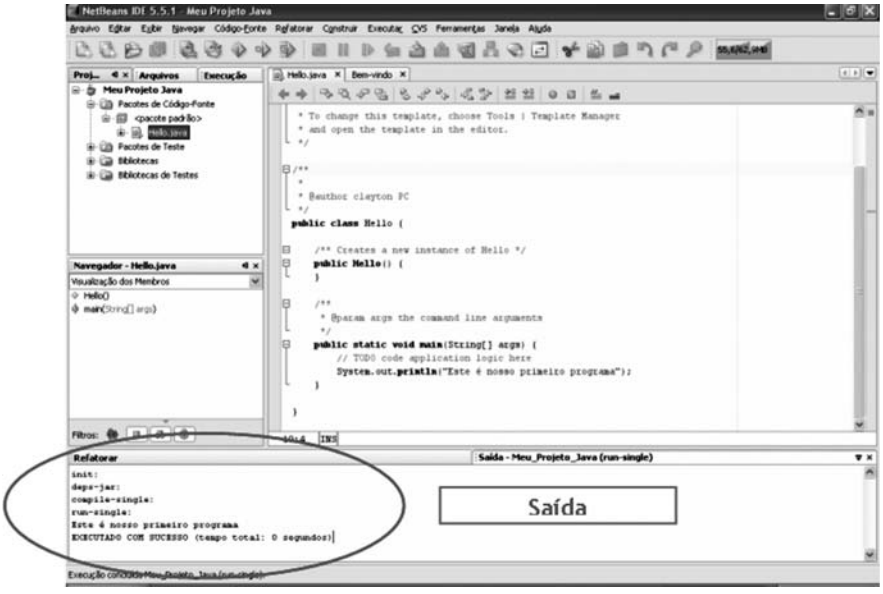


Passo 3

Ao mandar executar um programa, a IDE automaticamente compila e interpreta o código, mostrando o resultado dos comandos, caso seja uma aplicação via console (que será o nosso caso, por enquanto), na janela abaixo do código.

Para executar o código, você deve selecionar, na janela do lado esquerdo superior, a janela Projetos e depois o programa que deseja executar, e clicar com o botão direito do *mouse* selecionando a opção *Executar Arquivo*.





Com isso, você já tem tudo instalado e configurado para continuar seu aprendizado. Além disso, você já sabe como manipular as ferramentas necessárias e pode voltar para o ponto em que paramos na Aula 1 para dar continuidade aos estudos.

Quanto à IDE NetBeans, fique tranquilo. Se estiver parecendo muita informação, com o tempo e com os exercícios você vai ficar mais seguro, e os procedimentos de manipulação dos projetos vão se tornar mais naturais.

Não se esqueça: se tiver alguma dúvida ou dificuldade, mande uma mensagem para o fórum. Sempre estaremos prontos para debater suas dúvidas com seus colegas de aula.

Java Básico e Orientação a Objeto

Referências

BARUQUE, L. *Elgorm*: um modelo de referência para governança de e-learning. 2004. Tese (Doutorado) – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de janeiro, 2004.

MARTINS, F. M. *Java:* tecnologia e introdução à linguagem, 2006. Disponível em <http://sim.di.uminho.pt/disciplinas/poo/0708/CAP2_TECNOLOGIA_JAVA5.pdf>. Acesso em: jun. 2009.

OLIVEIRA JÚNIOR, D. A. *Comunidades de prática*: um estudo dos grupos de usuários Java brasileiros. 2005. Dissertação (Mestrado) – Programa de Pós-Graduação em Gestão do Conhecimento e da Tecnologia da Informação, Universidade Católica de Brasília, Brasília, 2005. Disponível em: <http://www.dfjug.org/DFJUG/DissertacaoDaniel%20v6.1.pdf>. Acesso em: jun. 2009.

Este curso é baseado no Java Education and Development Initiative

AOLI. Tecnologia Web. Disponível em: <http://www.aoli.com.br/dicionarios.aspx?palavra=Cache>. Acesso em: 22 abr. 2009.

GOOGLE. Disponível em: <http://www.google.com.br>. Acesso em: 22 abr. 2009.

JAVA NET. *The Source for Java Technology Collaboration*. Disponível em: <https://jedi.dev.java.net/>. Acesso em: 22 abr. 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 22 abr. 2009.

SUN. Sum Developer Network. *Online Courses*. Tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 22 abr. 2009.

SUN. Sum Developer Network. *The Source for java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 22 abr. 2009.

Sun Microsystems. In: *Wikipedia*: a enciclopédia livre. Disponível em: <http://pt.wikipedia.org/wiki/Sun_Microsystems>. Acesso em 22 abr. 2009.

THREAD (desambiguação). In: *Wikipedia*: a enciclopédia livre. Disponível em: <http://pt.wikipedia.org/wiki/Thread>. Acesso em: 22 abr. 2009.

WIKIPEDIA. Disponível em: <http://www.pt.wikipedia.org>. Acesso em: 22 abr. 2009.

Aula 2

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java Technology Collaboration. *Jedi Project*. Disponível em: <https://jedi.dev.java.net/>. Acesso em: 18 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org>. Acesso em: 18 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining>. Acesso em: 18 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com>. Acesso em: 18 maio 2009.

Aula 3

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. *Jedi Project*. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 4

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. *Jedi Project*. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 5

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. *Jedi Project*. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 6

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. *Jedi Project*. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 7

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. Jedi Project. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 8

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. Jedi Project. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 9

Este curso é baseado no Java Education and Development Initiative

JAVA Net: the source for Java technology collaboration. *Jedi Project*. Disponível em: <https://jedi.dev.java.net/.>. Acesso em: 07 maio 2009.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *Online courses*: tutorials and online training. Disponível em: <http://java.sun.com/developer/onlineTraining/>. Acesso em: 07 maio 2009.

SUN Developer Networker. *The spource for Java developers*. Disponível em: <http://java.sun.com/>. Acesso em: 07 maio 2009.

Aula 10

Este curso é baseado na Java Education and Development Initiative

JAVA. NET: The Source Of Java Techonology Collaboration. Disponível em: <https://jedi.dev.java.net/>. Acesso em: 05 ago. 2009.

NET BEANS IDE 6.7. Disponível em: <http://www.netbeans.org/>. Acesso em: 05 ago. 2009.

SUN DEVELOPER NETWORK (SDN). Disponível em: <http://java.sun.com/>. Acesso em: 05 ago. 2009.

SUN DEVELOPER NETWORK (SDN). Java.sun.com: the source for Java developers. Disponível em: <http://java.sun.com/developer/onlineTraining>. Acesso em: 05 ago. 2009.

ISBN 978-85-7648-648-0

FUNDAÇÃO SANTA CABRINI
Provedora de acesso à Cidadania

FAPERJ
Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro

GOVERNO DO Rio de Janeiro
SECRETARIA DE CIÊNCIA E TECNOLOGIA