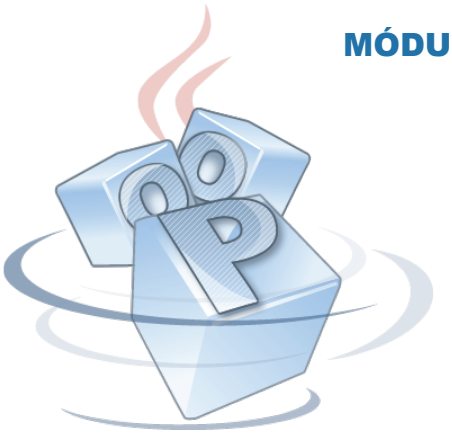
INSTITUTO FEDERAL SUL-RIO-GRANDENSE
UNIVERSIDADE ABERTA DO BRASIL
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET
Modalidade a Distância

MÓDULO 4



LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Júlio Mattos

Ministério da Educação
TSIaD
UAB UNIVERSIDADE ABERTA DO BRASIL SEED/MEC
INSTITUTO FEDERAL SUL-RIO-GRANDENSE Campus Pelotas
CAPES
GOVERNO FEDERAL BRASIL PAÍS RICO É PAÍS SEM POBREZA

PRESIDÊNCIA DA REPÚBLICA

Dilma Rousseff
PRESIDENTE DA REPÚBLICA FEDERATIVA DO BRASIL

MINISTÉRIO DA EDUCAÇÃO

Fernando Haddad
MINISTRO DO ESTADO DA EDUCAÇÃO

Luiz Cláudio Costa
SECRETÁRIO DE EDUCAÇÃO SUPERIOR - SESU

Eliezer Moreira Pacheco
SECRETÁRIO DA EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA

Luís Fernando Massonetto
SECRETÁRIO DA EDUCAÇÃO A DISTÂNCIA – SEED

Jorge Almeida Guimarães
PRESIDENTE DA COORDENAÇÃO DE APERFEIÇOAMENTO DE PESSOAL DE NÍVEL SUPERIOR - CAPES

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE [IFSUL]

Antônio Carlos Barum Brod
REITOR

Daniel Espírito Santo Garcia
PRÓ-REITOR DE ADMINISTRAÇÃO E DE PLANEJAMENTO

Janete Otte
PRÓ-REITORA DE DESENVOLVIMENTO INSTITUCIONAL

Odeli Zanchet
PRÓ-REITOR DE ENSINO

Lúcio Almeida Hecktheuer
PRÓ-REITOR DE PESQUISA, INOVAÇÃO E PÓS-GRADUAÇÃO

Renato Louzada Meireles
PRÓ-REITOR DE EXTENSÃO

IF SUL-RIO-GRANDENSE CAMPUS PELOTAS

José Carlos Pereira Nogueira
DIRETOR-GERAL DO CAMPUS PELOTAS

Clóris Maria Freire Dorow
DIRETORA DE ENSINO

João Róger de Souza Sastre
DIRETOR DE ADMINISTRAÇÃO E PLANEJAMENTO

Rafael Blank Leitzke
DIRETOR DE PESQUISA E EXTENSÃO

Roger Luiz Albernaz de Araújo
CHEFE DO DEPARTAMENTO DE ENSINO SUPERIOR

IF SUL-RIO-GRANDENSE DEPARTAMENTO DE EDUCAÇÃO A DISTÂNCIA

Luis Otoni Meireles Ribeiro
CHEFE DO DEPARTAMENTO DE EDUCAÇÃO A DISTÂNCIA

Daniel Grill Lacerda
COORDENADOR DA UNIVERSIDADE ABERTA DO BRASIL – UAB/IFSUL

Marla Cristina da Silva Sopeña
COORDENADORA ADJUNTA DA UNIVERSIDADE ABERTA DO BRASIL – UAB/IFSUL

Cinara Ourique do Nascimento
COORDENADORA DA ESCOLA TÉCNICA ABERTA DO BRASIL – E-TEC/IFSUL

Ricardo Lemos Sainz
COORDENADOR ADJUNTO DA ESCOLA TÉCNICA ABERTA DO BRASIL – E-TEC/IFSUL

IF SUL-RIO-GRANDENSE UNIVERSIDADE ABERTA DO BRASIL

Daniel Grill Lacerda
COORDENADOR DA UNIVERSIDADE ABERTA DO BRASIL – UAB/IFSUL

Marla Cristina da Silva Sopeña
COORDENADORA ADJUNTA DA UNIVERSIDADE ABERTA DO BRASIL – UAB/ IFSUL

Mauro Hallal dos Anjos
GESTOR DE PRODUÇÃO DE MATERIAL DIDÁTICO

CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET Modalidade a Distância

Daniel Grill Lacerda
COORDENADOR DO CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET – TSIAD

Beatriz Helena Zanotta Nunes
SUPERVISORA PEDAGÓGICA

Suzana Grala Tust
REVISORA LINGUÍSTICA

Vera Maria Machado Damé
COORDENADORA DE TUTORIA DO TSIAD

EQUIPE DE PRODUÇÃO DE MATERIAL DIDÁTICO – UAB/IFSUL

Lisiane Corrêa Gomes Silveira
GESTORA DA EQUIPE DE DESIGN

Alessandro Cruz Wrague
Aline de Almeida Tessmer
Andressa Silva Nasiloski
Denise Zarnottz Knaback
Felipe Rommel
Helena Guimarães de Faria
Helena Rodrigues dos Santos
Igor da Silva Amaral
Lucas Quaresma Lopes
Luisa Mendes Machado
Marco Lucas dos Anjos
Morgana Ávila dos Santos
Taís Lopes Barbosa
Talita Mesquita Barbosa
EQUIPE DE DESIGN

Catiúcia Klug Schneider
GESTORA DE PRODUÇÃO DE VÍDEO

Gladimir Pinto da Silva
PRODUTOR DE ÁUDIO E VÍDEO

Jeferson de Oliveira Oliveira
AUXILIAR DE EDIÇÃO DE VÍDEO E PROGRAMADOR EM FLASH

João Eliézer Ribeiro Schaun
GESTOR DO AMBIENTE VIRTUAL DE APRENDIZAGEM

Giovani Portelinha Maia
GESTOR DE MANUTENÇÃO E SISTEMA DA INFORMAÇÃO

Acauan Merseburger Picanço
Anderson Weige Dias
Bruna Gonçalves Ribeiro
Carlo Camani Schneider
Diego Barcellos Rocha
Efrain Becker Bartz
Francine Neuschrank
Gabriel Duarte
Mateus Lorenzato Braga
Neimar Mendes Lima
Paula Cruz Guttier
Piter Oliveira Vergara
Vinícius Maciel
EQUIPE DE PROGRAMAÇÃO PARA WEB

GUIA DIDÁTICO GD

APRESENTAÇÃO

Prezado (a) aluno (a),

Prezado(a) aluno(a),

Bem-vindo (a) ao espaço de estudo da Linguagem de Programação Orientada a Objetos.

Nesta disciplina, estudaremos os fundamentos da Programação Orientada a Objetos e aplicaremos estes princípios na linguagem Java.

Nas unidades, serão abordados os seguintes conteúdos: Introdução à Linguagem Java, Fundamentos de Orientação a Objetos com Java, Uso das Bibliotecas Java, Coleções de Objetos e Tratamento de Erros e Exceções.

Esperamos que, através dos conteúdos e das atividades propostas, você possa estabelecer subsídios para modelar e desenvolver programas orientados a objetos através da linguagem Java.

Desejamos um ótimo aprendizado !

Objetivos

Objetivo Geral

Compreender os conceitos básicos abstraindo e modelando soluções sob enfoque da Programação Orientada a Objetos.

Objetivos Específicos

- Compreender a filosofia e princípios da Programação Orientada a Objetos;
- Compreender e distinguir os principais componentes de uma Linguagem Orientada Objetos;
- Identificar os princípios básicos da Orientação a Objetos;
- Compreender os conceitos de Classes, Objetos, Atributos e Métodos;
- Compreender os conceitos Abstração, Encapsulamento, Herança, Polimorfismo;
- Armazenar e manipular dados através de uma Linguagem de Programação Orientada a Objetos;
- Diferenciar os métodos para o desenvolvimento de Sistemas Orientados a Objetos;
- Desenvolver programas utilizando uma Linguagem Orientada a Objetos.

Metodologia

O tema será desenvolvido através do Ambiente Virtual de Aprendizado Moodle, onde serão disponibilizados materiais a serem estudados para subsidiar a aprendizagem. Os recursos tecnológicos para interação serão os seguintes: Fórum e Chat de Dúvidas, E-mail, Textos, Exercícios on-line.

Avaliação

A avaliação dar-se-á mediante a participação nos fóruns e nas atividades propostas, tanto presenciais como a distância.

Programação

A disciplina será desenvolvida no decorrer de três semanas.

Primeira semana:

1. Leitura e estudo da Unidade A1: Introdução à Linguagem Java.
2. Leitura e estudo da Unidade A2: Estruturas Fundamentais da Linguagem Java.
3. Vídeo Conferência: revisão dos principais conceitos da Unidade A e prática (A2).

Segunda semana:

4. Atividade: Atividade 1 (Unidade A1).
5. Atividade: Atividade 2 (Unidade A2).
6. Leitura e estudo da Unidade B1: Conceitos Básicos de Orientação a Objetos em Java.
7. Fórum: dúvidas.
8. Video conferência: revisão dos principais conceitos e prática dos conteúdos da Unidade B.

Terceira Semana:

9. Leitura e estudo da Unidade B2: Conceitos Avançados de Orientação a Objetos em Java.
10. Atividade: Atividade 3 (Unidade B1).
11. Fórum: dúvidas.
12. Vídeo Conferência: revisão dos principais conceitos e prática dos conteúdos da Unidade B

Quarta Semana

13. Leitura e estudo da Unidade C: Uso de Pacotes e Bibliotecas.
14. Atividade: Atividade 4 (Unidade B2).
15. Fórum: dúvidas.
16. Vídeo conferência: revisão dos principais conceitos e prática dos conteúdos da Unidade C.

Quinta semana

17. Atividade: Atividade 5 (Unidade C).
18. Leitura e estudo da Unidade D: Coleções de Objetos, Tratamento de Exceções.
19. Atividade: Atividade 6 (Unidade D).
20. Fórum: dúvidas.
21. Vídeo Conferência: revisão dos principais conceitos e práticas dos conteúdos da Unidade D.
22. Encontro Presencial: Avaliação

Orientações quanto ao desenvolvimento do projeto

O projeto que será desenvolvido no 3º módulo será a criação de um portal para o polo, onde diferentes etapas serão desenvolvidas pelas demais disciplinas deste módulo.

Na primeira disciplina deste módulo, ou seja, na disciplina de Gerência de Projetos de Sistemas será desenvolvido um trabalho final com o intuito de fazer o planejamento desse portal a ser desenvolvido posteriormente, durante as demais disciplinas.

Instruções do Trabalho

Objetivo do Trabalho Final

- Planejar, em equipe, o sistema de informatização do Polo de Apoio Presencial da UAB/Instituto Federal Sul-rio-grandense (já estudado na disciplina de Análise e Projetos de Sistemas).

Organização dos alunos para o trabalho

- A turma deve ser dividida em 5 (cinco) grupos (Turmas com mais de 25 alunos devem ser dividida em 6 (seis) grupos).
- Deve ser definido um gerente de projeto para cada grupo que será responsável por postar, no Moodle, o planejamento do grupo.
- Deve ser definido um gerente de projeto geral para o projeto que será responsável por organizar e postar, no Moodle, o Plano Geral do projeto criado por toda a turma.

Observações

- Os documentos a serem planejados encontram-se disponíveis no Moodle, através da ferramenta Wiki, para serem editados e completados. Todo o material gerado para o trabalho deve ser documentando através da Wiki.
- Cada grupo desenvolverá seu Plano Geral do Projeto.
- Durante o primeiro encontro presencial, haverá a divisão dos grupos, seguindo critérios definidos pelos professores da disciplina. No último encontro presencial, haverá uma apresentação em forma de banca, na qual os alunos de cada grupo irão apresentar o projeto e serão avaliados pelo grupo de professores das disciplinas envolvidas.
- Ao término da disciplina, deverá ser gerado um único Plano geral para toda a turma, o qual servirá de base para a futura implementação do Portal. Para isso, será necessário que a turma se reúna, discuta e escolha os melhores aspectos e itens de cada grupo.

Referências:

AUDY, Jorge Luis N.; ANDRADE, Gilberto; CIDRAL, Alexandre. Fundamentos de Sistema de Informação. Porto Alegre: Bookman, 2005

HELDMAN, Kim. Gerência de Projetos: fundamentos. 4ª Edição. Rio de Janeiro: Elsevier, 2005

HELDMAN, Kim. Gerência de Projetos: guia para o exame oficial da PMI. 3ª Edição. Rio de Janeiro: Elsevier, 2006

PRESSMAN, Roger S.. Engenharia de Software. 5ª Edição. Rio de Janeiro: McGraw-Hill, 2002

UNIDADE A

INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO JAVA

Nesta disciplina, estudaremos os fundamentos da Programação Orientada a Objetos e aplicaremos estes princípios na linguagem Java. A Unidade A apresenta uma introdução à linguagem Java, como o seu histórico, seus objetivos, características e por fim, você conhecerá as estruturas fundamentais desta linguagem e executará seu primeiro "programa Java".

Introdução

A primeira versão da linguagem Java foi lançada em 1996 pela Sun Microsystems (atualmente adquirida pela Oracle), porém Java nunca foi apenas uma linguagem de programação. Java é uma plataforma para desenvolvimento e execução de aplicações que possui uma biblioteca com uma grande quantidade de códigos reutilizáveis e um ambiente de execução que fornece serviços como segurança e portabilidade para diferentes sistemas operacionais. Atualmente, Java é utilizada para desenvolvimento de software desde aplicações coorporativas com acesso a grandes bancos de dados até aplicações embarcadas como celulares, impressoras, etc.

A plataforma Java tem revolucionado o desenvolvimento das aplicações Internet/Intranet e tem tornado-se uma linguagem atrativa para diversos programadores através de uma sintaxe agradável e uma semântica compreensível. Além disso, uma das principais vantagens em relação a outras linguagens é a ampla biblioteca disponível (em Java, existem códigos prontos para manipulação de elementos gráficos, acesso a banco de dados ou rede) tornando desnecessária a tarefa do programador de criar os seus próprios códigos para tarefas comuns.

Outra grande vantagem de Java é portabilidade. A Sun Microsystems criou o slogan *"write once, run everywhere"* que significa "escrever uma vez e executar em qualquer lugar", onde uma aplicação Java pode ser executada em diferentes sistemas operacionais e/ou computadores com diferentes arquiteturas sem modificação alguma.

Um dos objetos iniciais de Java foi permitir a troca de executáveis Java entre computadores pela Internet, sendo que o receptor deste programa não deveria realizar modificação alguma. A ideia era simples: os usuários farão o download dos bytecodes Java (executáveis Java) da Internet e irão executá-los nas suas próprias máquinas. Assim, nasceram os **applets**, programas Java que funcionam em páginas Web. Para utilizar um **applet**, somente é necessário um navegador compatível com Java que os bytecodes serão executados (independente da plataforma, ou seja, sistemas operacional e arquitetura da máquina – exemplo: x86 utilizando Linux).

A tecnologia Java possui um conjunto de características no qual destacamos as a seguir:

- **Simples**: Java foi inspirada na linguagem C++ para tornar a linguagem compreensível, porém a sintaxe foi simplificada (não existe necessidade de arquivos de cabeçalho, manipulação de ponteiros, etc.). Outra intenção dos criadores de Java foi permitir a construção de software pequenos que possam ser executados em máquinas de pequeno porte, por exemplo, atualmente todos os celulares executam Java;
- **Orientada a Objetos**: Java é uma linguagem orientada a objetos. O paradigma orientado a objetos é um padrão conceitual que orienta soluções de projeto e implementação de software e se constitui na principal metodologia utilizada nas últimas décadas (será estudado com profundidade na disciplina de Análise e Projeto de Sistemas de Informação Orientados a Objetos). Uma grande vantagem da orientação a objetos é o reuso de código;
- **Robusta e Segura**: Java possui diversas verificações para detectar possíveis problemas em tempo de compilação e tempo de execução. Uma grande vantagem de Java é que não é possível utilizar ponteiros explicitamente. Além disso, Java foi concebida para ser utilizada em ambientes de rede/distribuídos, por isso foi dada ênfase em segurança;
- **Arquitetura Neutra e Portável**: o compilador gera um formato de arquivos que é neutro em relação a arquitetura (o

código gerado pelo compilador, chamado de bytecodes, pode ser executado em qualquer arquitetura – processador - desde que o ambiente de execução do Java esteja presente). Este ambiente de execução é uma máquina virtual. A linguagem é altamente portável pois não possui nenhum aspecto dependente de implementação (como C e C++, onde o tamanho de um dado inteiro depende do compilador). Além disso, as bibliotecas são parte do sistema e definem interfaces portáveis, com diferentes implementações para diferentes sistemas operacionais e arquiteturas;

- **Interpretada**: Java possui um interpretador que pode executar o código compilado Java (bytecodes) diretamente em qualquer máquina em que o interpretador esteja portado;
- **Multithread**: a linguagem Java oferece programação em múltiplas threads, ou seja, em um programa pode possuir várias linhas de execução realizando diversas tarefas ao mesmo tempo. O uso de multithreading é extremamente útil, por exemplo, um navegador pode executar o download de diversas imagens simultaneamente;
- **Dinâmica**: Java permite um excelente gerenciamento dinâmico (em tempo de execução) das suas aplicações. Isto facilita na execução de aplicações em que o código é descarregado da Internet e executado em um navegador;
- **Desempenho**: Java utiliza interpretação, o que reduz o desempenho na execução de aplicações. Porém, Java utiliza compiladores Just-in-time que aumentam consideravelmente o desempenho das aplicações Java;

Tecnologias Java

Java possui com grande conjunto de tecnologias que são agrupadas em diferentes plataformas. Estas plataformas agrupam programas relacionados que permitem o desenvolvimento e execução de programas escritos na linguagem de programação Java. As principais plataformas são:

- **Java Platform Enterprise Edition (JEE)**: conjunto de tecnologias para desenvolvimento de aplicações coorporativas e internet (aplicações cliente servidor, entre outras);
- **Java Platform Standard Edition (JSE)**: é a plataforma base do Java, utilizada para desenvolvimento de aplicações de propósito geral, servidores e outros dispositivos;
- **Java Platform Micro Edition (JME)**: plataforma desenvolvida para dispositivos embarcados com limitações de desempenho e memória, como pagers, celulares, etc. Especifica um conjunto de bibliotecas (conhecidos como profiles) menor que o da plataforma Java SE;
- **JAVA Card**: plataforma utilizada no desenvolvimento de aplicações para smart cards.

Basicamente, cada plataforma Java consiste de um conjunto essencial de componentes como: o compilador que transforma o código fonte Java em uma linguagem intermediária (os bytecodes) que é fornecido pelo **Java Development Kit (JDK)**, as bibliotecas e o ambiente de execução que executa os bytecodes Java são fornecidos pelo **Java Runtime Environment (JRE)**. O JRE possui os elementos necessários para a execução de programas Java como a **Java Virtual Machine (JVM)** que possui a tarefa de converter os bytecodes em código de máquina em tempo de execução.

O **Java Development Kit (JDK)**, antigamente tratado como **SDK (Software Development Kit)**, contém o compilador Java e diversas outras ferramentas de desenvolvimento como também uma cópia completa do Java **Runtime Environment (JRE)**.

Você pode notar que o JDK somente será necessário para desenvolvedores Java (o nosso caso), enquanto que o JRE é necessário em usuários de computadores em geral, para que estes possam executar aplicativos Java (muitas vezes os usuários nem notam que executam aplicações Java).

A máquina virtual Java (**Java Virtual Machine – JVM**) é responsável pela execução dos bytecodes Java (código intermediário) gerados pelo compilador. Esta máquina virtual realiza a interpretação dos bytecodes (tradução em tempo de execução para o código de máquina). A JVM é auxiliada por um compilador **Just-in time (JIT) na tradução dos bytecodes**.

O uso dos bytecodes como linguagem intermediária **permite a execução de programas Java em qualquer plataforma** (diferentes arquiteturas e sistemas operacionais) desde que possua a JVM disponível, tornando assim, a linguagem Java **neutra e portável** em relação a arquitetura. Apesar dos programas Java serem independentes

de plataforma, o código da JVM que executa estes programas não é independente e necessita de uma JVM para cada plataforma.

A Figura 1 apresenta diversas plataformas Java, suas usos e tecnologias envolvidas. A plataforma Java, Micro Edition (Java ME) possui diversas tecnologia envolvidas como CDC, CLDC, MIDP que necessitariam de outra disciplina para abordar todos os conceitos envolvidos.

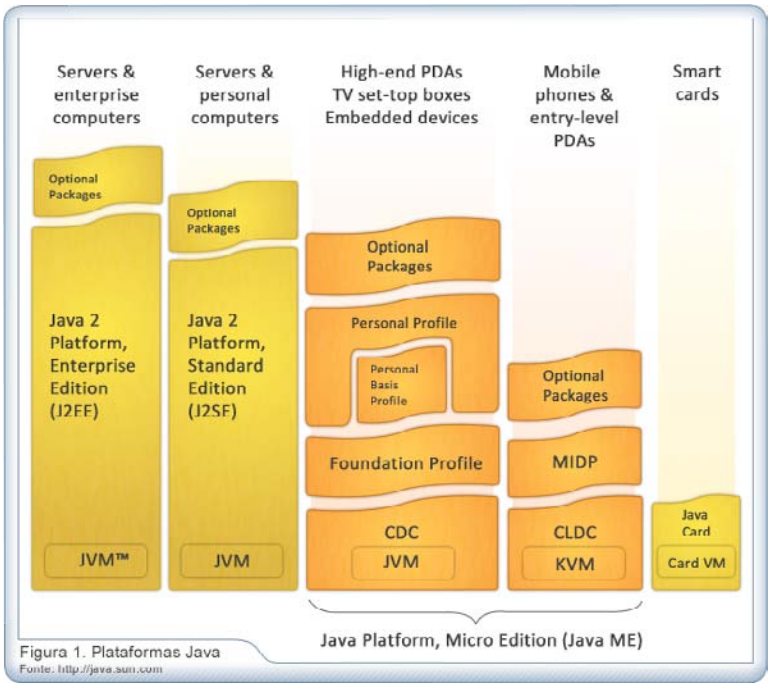


Figura 1. Plataformas Java
Fonte: http://java.sun.com

Durante a nossa disciplina nos utilizaremos a plataforma **Java Standart Edition (Java SE)**. Esta plataforma possui dois principais produtos: Java Development Kit (JDK) e Java Runtime Environment (JRE). O JDK é um superconjunto do JRE, contendo tudo do JRE mais ferramentas de desenvolvimento como compiladores, depuradores, ferramentas para documentação de código, etc.

A Figura 2 apresenta um diagrama conceitual de todos os componentes da plataforma Java SE e os seus relacionamentos.

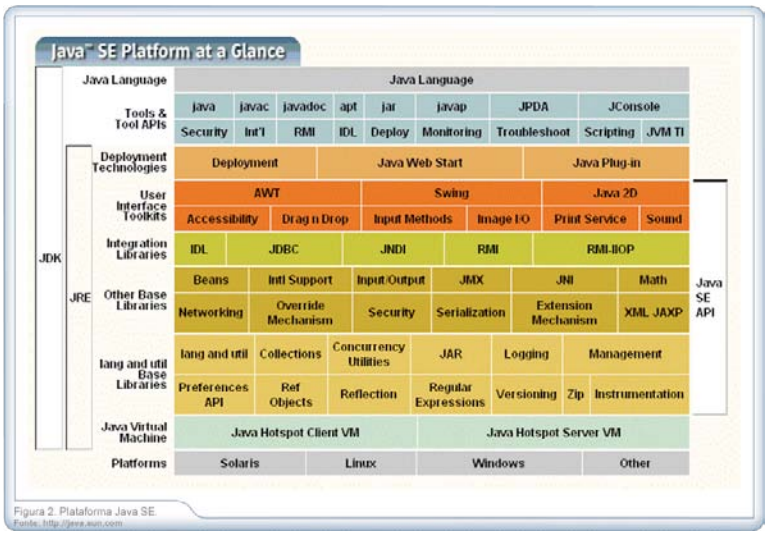


Figura 2. Plataforma Java SE
Fonte: http://java.sun.com

Fluxo de Desenvolvimento e Execução de um Programa Java

Vamos mostrar de maneira simplificada o fluxo de desenvolvimento e execução de um programa Java.

O desenvolvimento tradicional de um programa (por exemplo, na linguagem C) é apresentado na Figura 3: o programador utiliza um editor e escreve um programa que é compilado para um código objeto específico para uma plataforma. Ou seja, se o programar compilou para um sistema operacional Windows e necessitasse executar este mesmo código objeto em outro computador com o sistema operacional Linux será necessário recompilador o arquivo fonte.

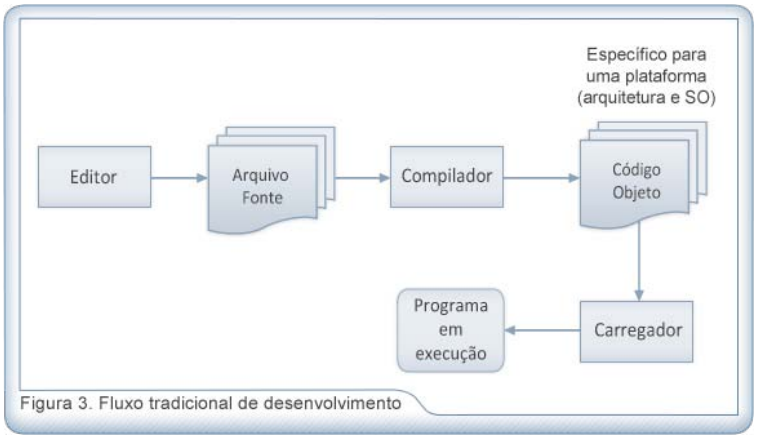


Figura 3. Fluxo tradicional de desenvolvimento

Já no desenvolvimento de software **baseado em Java** o código gerado pelo compilador é um código portável que pode ser executado em qualquer plataforma que possua o Java Runtime Environment (JRE) instalado. A Figura 4 apresenta este fluxo de desenvolvimento.

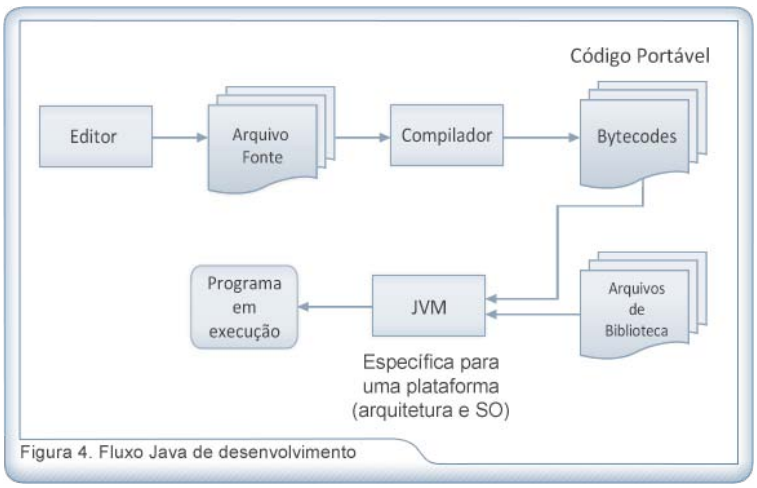


Figura 4. Fluxo Java de desenvolvimento

A portabilidade é fornecida pelo bytecodes Java (código intermediário) e pela máquina virtual Java (JVM). A Figura 5 ilustra o mesmo código Java sendo executado em diversas plataformas.

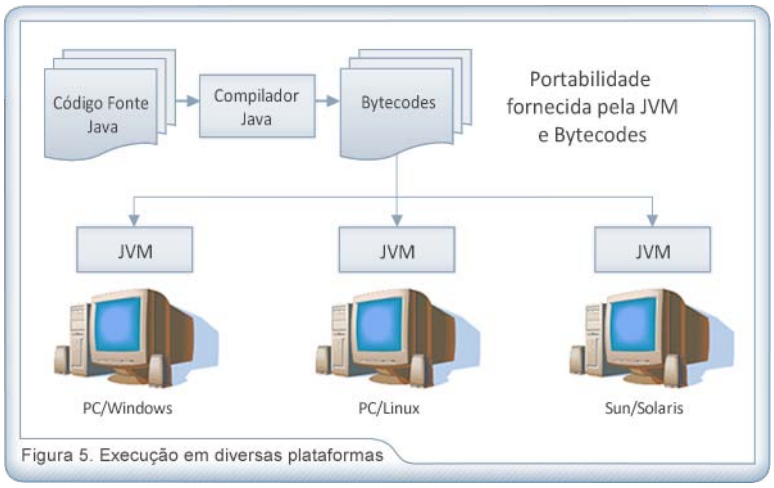


Figura 5. Execução em diversas plataformas

Ambiente de Programação Java

Para desenvolver os nossos programas Java durante a disciplina será necessário instalar o **Java Development Kit (JDK)** e um ambiente de desenvolvimento gráfico.

Nós utilizaremos o Java SE Development Kit 6 (JDK 6) que o download pode ser realizado em (existem versões para Windows, Solaris e Linux):

- http://java.sun.com/javase/downloads/widget/jdk6.jsp

Também iremos utilizar um ambiente gráfico de desenvolvimento (uma IDE - *Integrated Development Environment*). Os ambientes integrados para desenvolvimento de software facilitam a programação e depuração de programas. Existem diversos ambientes que podem ser utilizados com Java, contudo dois ambientes se sobre saem: Eclipse (http://eclipse.org/) e Netbeans (http://netbeans.org/).

Nós utilizaremos o Netbeans IDE, que o download pode ser realizado em (existem versões para Windows, Solaris, Linux, Mac OS):

- http://netbeans.org/downloads/index.html

Instalando o JDK no Windows

Para instalar o JDK no Windows apenas execute (duplo click) no arquivo executável resultante do download. Você pode instalar no diretório padrão (C:\Program Files\Java\jdk1.6.0_20) ou em outro diretório de sua preferência. Aceite a licença (Figura 6) e continue a instalação.

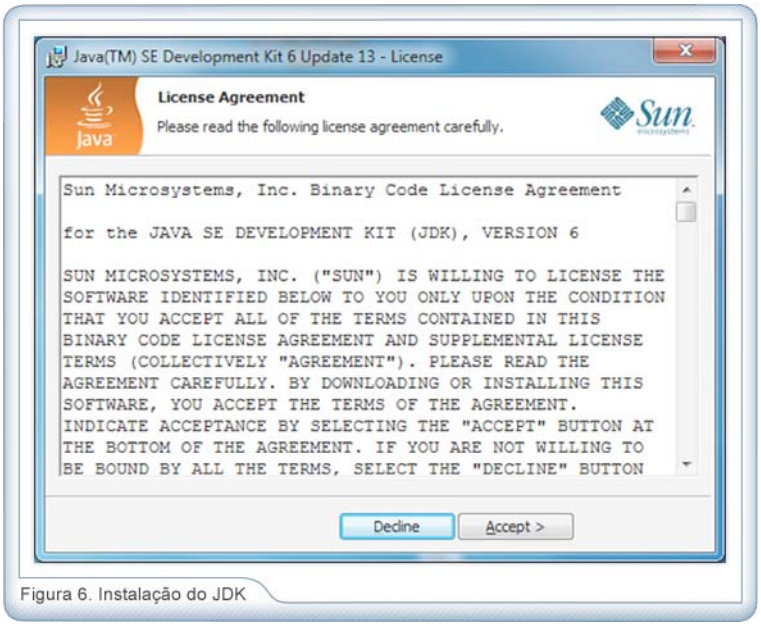


Figura 6. Instalação do JDK

Após a instalação do JDK, você deve configurar as variáveis de ambiente. Esta configuração facilita a uso do Java por linha de comando (utilizando um prompt de comando). Existem três variáveis de ambiente importantes para o correto funcionamento do Java:

- **path**: variável que indica a lista de diretórios que o SO analisa para localizar arquivos executáveis;
- **JAVA_HOME**: variável que indica o diretório de instalação do Java;
- **CLASSPATH**: variável que especifica onde estão armazenados os arquivos e bibliotecas necessários, tanto para a compilação, quanto para a execução.

Os valores para as variáveis de ambiente serão:

- JAVA_HOME = C:\Program Files\Java\jdk1.6.0_20 (diretório que o Java foi instalado em sua máquina)
- path = %JAVA_HOME%\bin (indica o diretório bin dentro do diretório de instalação do Java)
- CLASSPATH = %JAVA_HOME%;. (indica o diretório de instalação do Java e o diretório corrente – observe o ponto)

Para configurar no Windows você deve ir em Computador, Propriedades, Configurações Avançadas do Sistema, Avançadas e por fim Variáveis de Ambiente e criar as três variáveis descritas acima (as Figuras 7 e 8 apresentam a tela da Propriedades do Sistema e Variáveis de Ambiente).

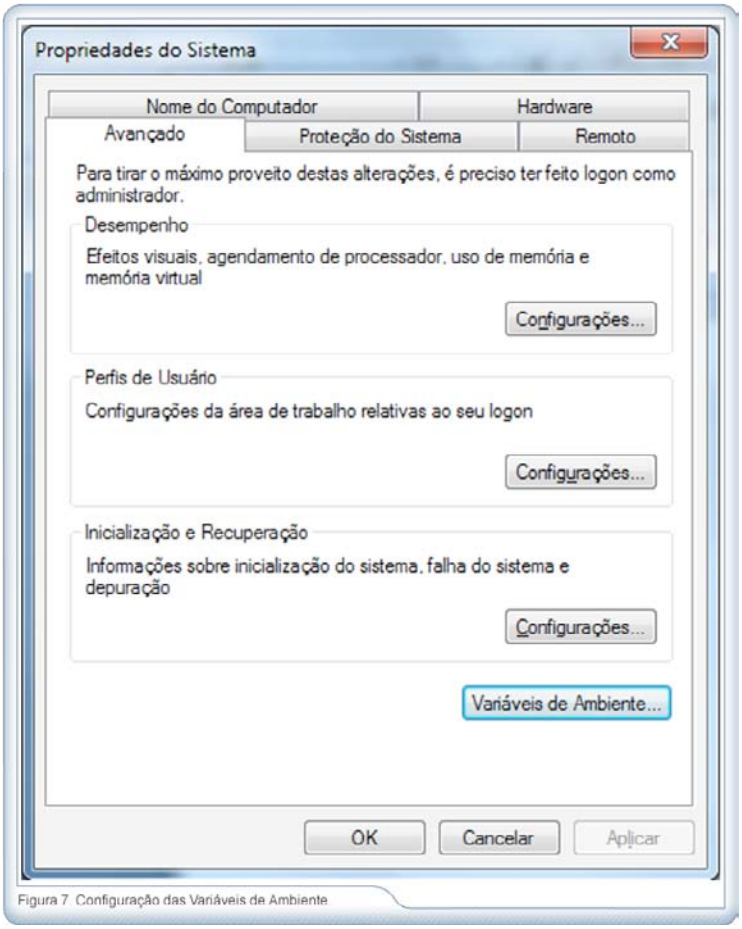


Figura 7 Configuração das Variáveis de Ambiente

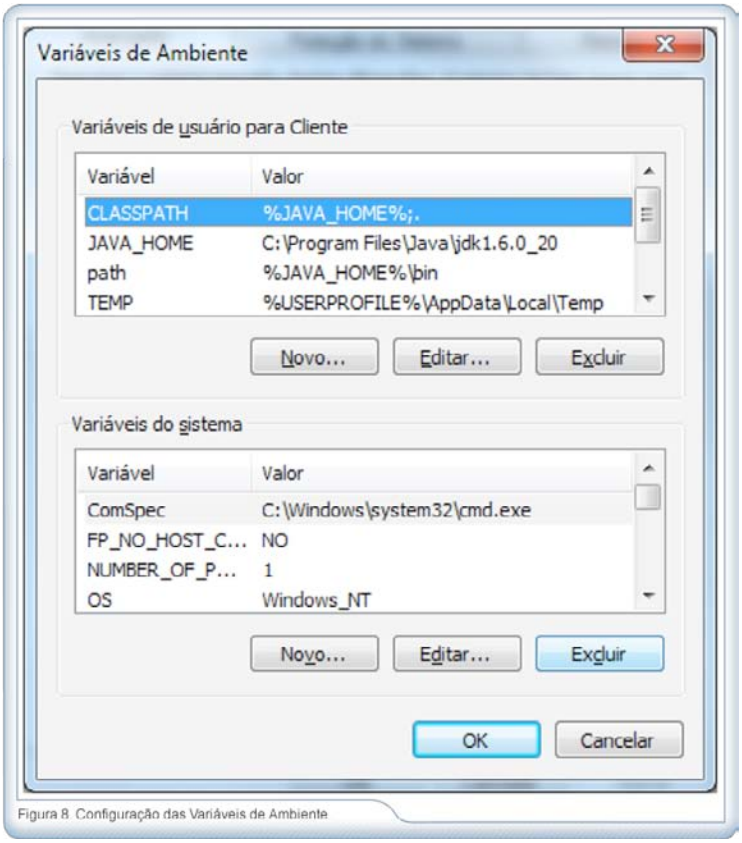


Figura 8 Configuração das Variáveis de Ambiente

O JDK possui a seguinte estrutura de diretórios (entre no diretório onde o Java foi instalado em sua máquina) onde destacamos os seguintes:

\ jdk1.6.0_20	
\bin	Compilador e demais ferramentas
\demo	Exemplos e demostrações
\docs	Documentação da biblioteca em HTML (é necessário realizar o download de um arquivo compactado no site da Sun)
\jre	Arquivos do ambiente de execução
\lib	Arquivos de biblioteca

Instalando o NetBeans no Windows:

Para instalar o NetBeans no Windows, basta executar (duplo click) o arquivo executável resultante do download. A única configuração necessária é indicar o diretório de instalação do NetBeans e o diretório do JDK (conforme apresentado na Figura 9).

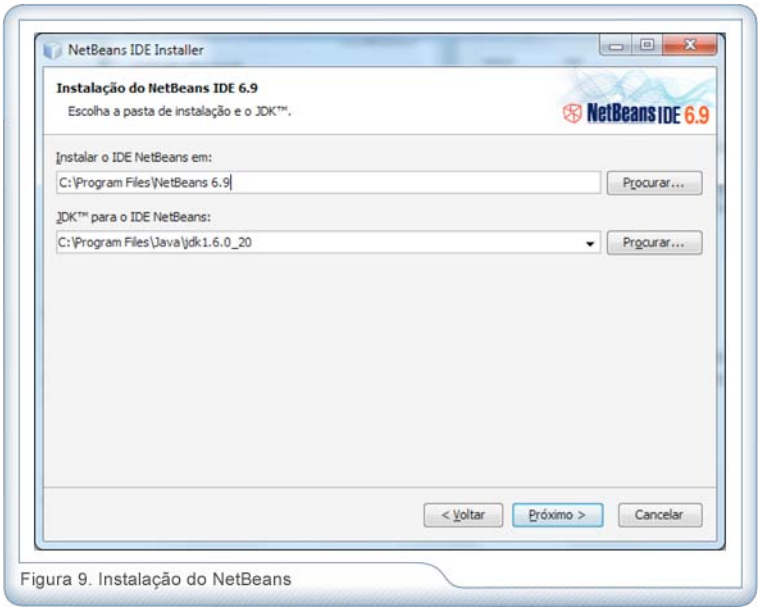


Figura 9. Instalação do NetBeans

Onde aprender mais ?

Além da bibliografia da disciplina existe muito material na Internet sobre Java. Porém uma ótima fonte de consulta é o próprio site da Sun (**http://java.sun.com/**). Destaco um conjunto de tutoriais sobre Java que pode ser encontrado em:

- http://java.sun.com/docs/books/tutorial/index.html

Esta seção apresentou os conceitos introdutórios e as tecnologias de Java. Na próxima seção, você irá conhecer as estruturas fundamentais da linguagem Java e executará o seu primeiro programa Java.

O JDK possui a seguinte estrutura de diretórios (entre no diretório onde o Java foi instalado em sua máquina) onde destacamos os seguintes:

- \ jdk1.6.0_20
- \bin Compilador e demais ferramentas
- \demo Exemplos e demostrações
- \docs Documentação da biblioteca em HTML (é necessário realizar o download de um arquivo compactado no site da Sun)
- \jre Arquivos do ambiente de execução
- \lib Arquivos de biblioteca

Instalando o NetBeans no Windows:

Para instalar o NetBeans no Windows, basta executar (duplo click) o arquivo executável resultante do download. A única configuração necessária é indicar o diretório de instalação do NetBeans e o diretório do JDK (conforme apresentado na Figura 8).

Onde aprender mais ?

Além da bibliografia da disciplina existe muito material na Internet sobre Java. Porém uma ótima fonte de consulta é o próprio site da Sun (http://java.sun.com/). Destaco um conjunto de tutoriais sobre Java que pode ser encontrado em:

http://java.sun.com/docs/books/tutorial/index.html

Esta seção apresentou os conceitos introdutórios e as tecnologias de Java. Na próxima seção, você irá conhecer as estruturas fundamentais da linguagem Java e executará o seu primeiro programa Java.

UNIDADE A

ESTRUTURAS FUNDAMENTAIS DA LINGUAGEM JAVA

Neste ponto, consideramos que você já tenha entendido os conceitos introdutórios e as tecnologias de Java apresentados na seção anterior, como também tenha conseguido instalar corretamente os JDK e a IDE NetBeans.

Nesta seção, você deverá:

- Conhecer as estruturas básicas da linguagem Java como tipos de dados, comandos de repetição, comandos de seleção entre outras. Estas estruturas são muito semelhantes às utilizadas por outras linguagens como C e PHP;
- Executar programas Java na IDE NetBeans.

Primeiro Programa em Java

Antes de apresentarmos as estruturas de programação fundamentais do Java, vamos conhecer um pequeno programa Java, sua estrutura básica e executar este programa no NetBeans.

Um código em Java pode ser estruturado da seguinte maneira: uma classe é colocada dentro de um arquivo fonte; métodos são colocados dentro da classe e por fim, comandos são colocados dentro dos métodos (como apresentado na Figura 1).

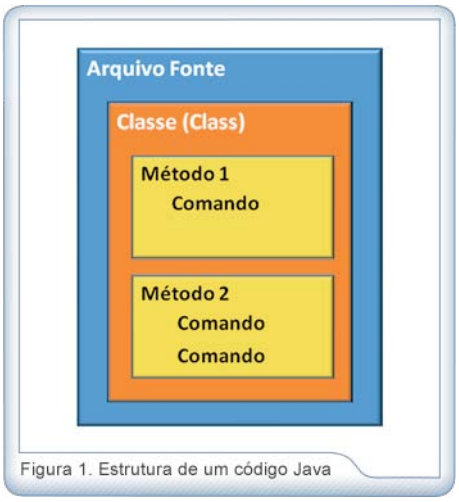


Figura 1. Estrutura de um código Java

Desta forma algumas perguntas devem ser respondidas:

1. **O que é colocado em um arquivo fonte?** Um arquivo fonte possui a extensão .java e armazena uma definição de classe. A classe é um pedaço de um programa (porém pode ser a única parte se for um programa muito pequeno);
2. **O que é colocado em uma classe (class)?** Uma classe contém um ou mais métodos. Os métodos agrupam sequências de comandos para realizar uma determinada ação;
3. **O que é colocado em um método?** Um método nada mais é que um conjunto de comandos, por enquanto, você pode imaginar que um método é uma função ou um procedimento.

O objetivo do primeiro programa em Java é bem simples: imprimir uma mensagem no console padrão. O programa possui 5 linhas de código que são explicadas a seguir (diversos detalhes serão omitidos, porém não se preocupe pois iremos estudar estes detalhes durante a disciplina):

- Linha 1: define uma classe de nome “PrimeiroPrograma” que é acessível (visível) a qualquer outro código;

- Linha 2: define um método chamado "*main*" também acessível a outros códigos e não retorna nenhum valor (a palavra static veremos mais adiante);
- Linha 3: Imprime uma string (na verdade, chama um método – print – da biblioteca e passa como parâmetro para impressão uma string);
- Linha 4: fecha o parênteses do método aberto na linha 2;
- Linha 5: fecha o parênteses da classe aberto na linha 1.

```
1.  public class PrimeiroPrograma {
2.      public static void main(String[] args){
3.          System.out.print("Hello World !");
4.      }
5.  }
```

A Figura 2 apresenta as explicações do primeiro exemplo de forma gráfica.

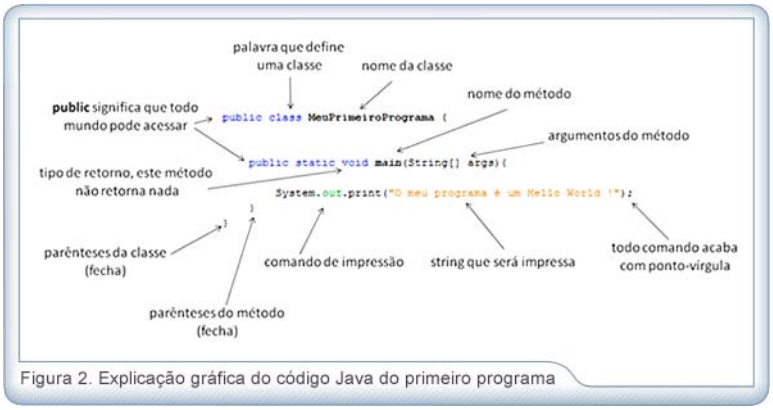


Figura 2. Explicação gráfica do código Java do primeiro programa

Observações:

- A linguagem Java é "case sensitive", ou seja, ela diferencia letras maiúsculas de minúsculas;
- As palavras azuis no código são palavras reservadas da linguagem;
- As regras para nomes de classe são bem fáceis: devem começar sempre por um letra e, depois dela, podem ter qualquer combinação de letras ou números (porém você não pode utilizar uma palavra reservada do Java, como public ou class);
- A convenção-padrão (*Code Conventions*) define que nomes de classes devem ser substantivos e sempre iniciam com a primeira letra em maiúscula. Se um nome consistir em múltiplas palavras, utilize uma letra maiúscula em cada uma das palavras (exemplo: MeuPrimeiroPrograma). Isso é chamado de notação camelo (camel case);
- O nome do arquivo fonte deve ser o mesmo nome da classe, com a extensão Java anexada. Desta forma, o nome do arquivo do primeiro exemplo deve ser "**PrimeiroExemplo.java**";
- As chaves { } delimitam **blocos** de código. No exemplo acima as chaves delimitam o código pertenente a classe PrimeiroPrograma e o código pertencente a método main;
- A JVM sempre inicia a execução com o código no método main na classe que você indica. Assim, sempre você precisa ter um método main no seu arquivo fonte para a sua classe para que o seu código seja executado.

Dicas:

O Java possui uma convenção padrão () que específica a melhor forma de organizar o código. Este estilo de codificação apresenta "regras" para nomes e formas adequadas para as diversas estruturas de programação e Java. Este padrão de código deve ser seguido pois permite um código mais legível e de mais fácil manutenção.

O "**Code Conventions for the Java Programming Language**" pode ser encontrado no link":

http://java.sun.com/docs/codeconv/

Executando o primeiro programa no NetBeans

Para executar o seu primeiro programa Java você deve:

- **Criar um projeto no NetBeans:** quando você cria um projeto na IDE, você cria um ambiente para desenvolver e executar as suas aplicações. O uso da IDE elimina necessidades de configurações manuais e facilita o desenvolvimento e teste das suas aplicações. Um projeto consiste de diversos arquivos fonte e configurações (não é necessário criar um projeto para cada programa Java);
- **Escrever o código Java da aplicação:** você deve criar um arquivo fonte Java e escrever o seu código;
- **Compilar o código fonte em um arquivo .class:** o NetBeans irá invocar o compilador Java e gerar o arquivo .class que contém os bytecodes Java;
- **Executar a aplicação Java:** o NetBeans invocará o aplicação que utiliza a máquina virtual Java (Java) e executa a sua aplicação.

Criar um projeto no NetBeans

- *Abra o NetBeans;*
- *No NetBeans, selecione o menu "Arquivo" | "Novo Projeto" (Figura 3);*

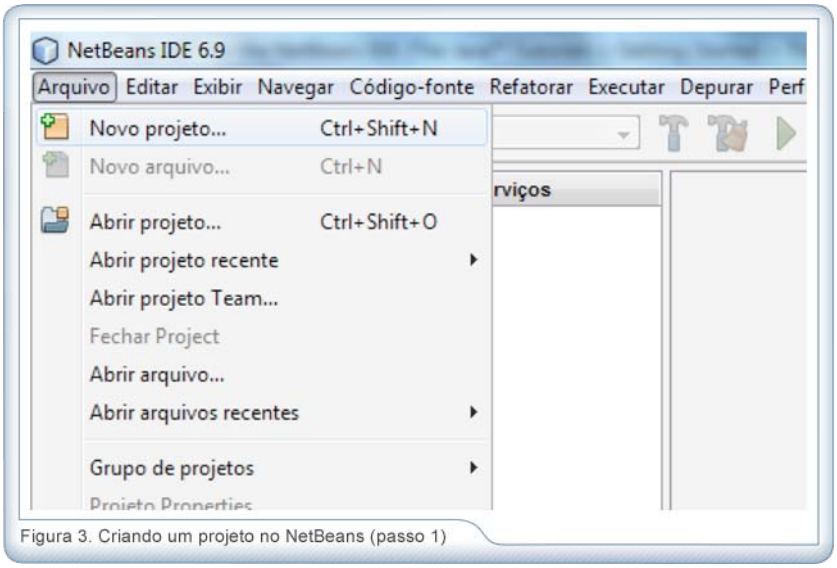


Figura 3. Criando um projeto no NetBeans (passo 1)

- No wizard do “Novo projeto” escolha nas Categorias “Java” e no item Projetos escolha “Aplicativo Java” e pressione próximo (Figura 4);

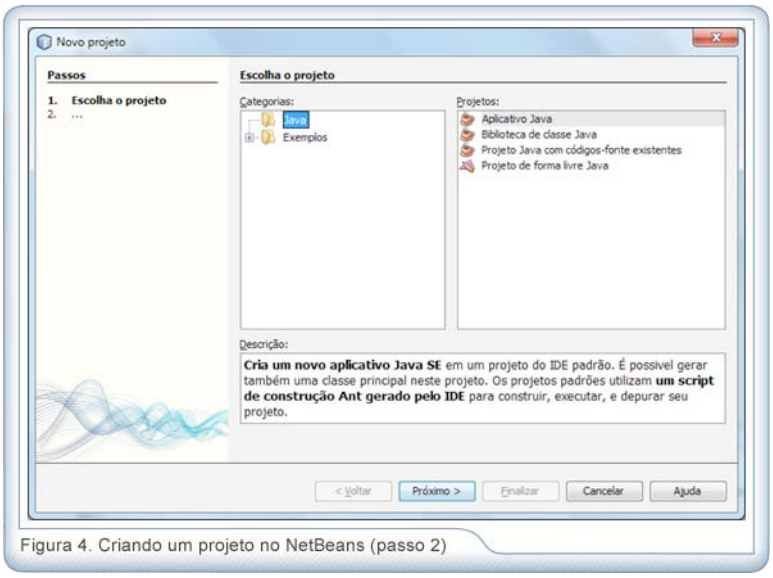


Figura 4. Criando um projeto no NetBeans (passo 2)

- Coloque o nome do Novo Projeto “PrimeiroProjeto” e desmarque a opção “Criar classe principal” e pressione Finalizar (Figura 5);

Escrever o código Java da aplicação

- Para criar um arquivo fonte Java selecione o menu “Arquivo” | “Novo arquivo” (Figura 6);

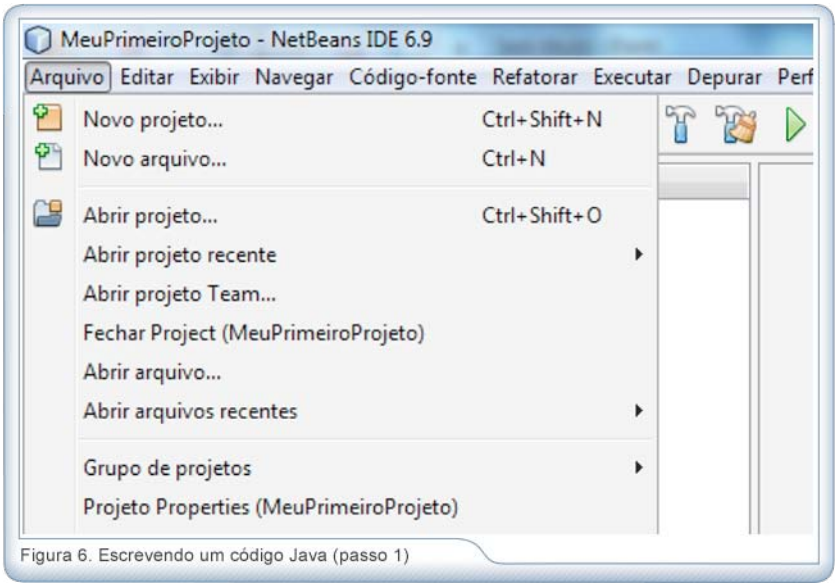


Figura 6. Escrevendo um código Java (passo 1)

- Escolha o tipo de arquivo na Categoria “Java” | Tipo de arquivos “Classe Java” e pressione próximo (Figura 7);

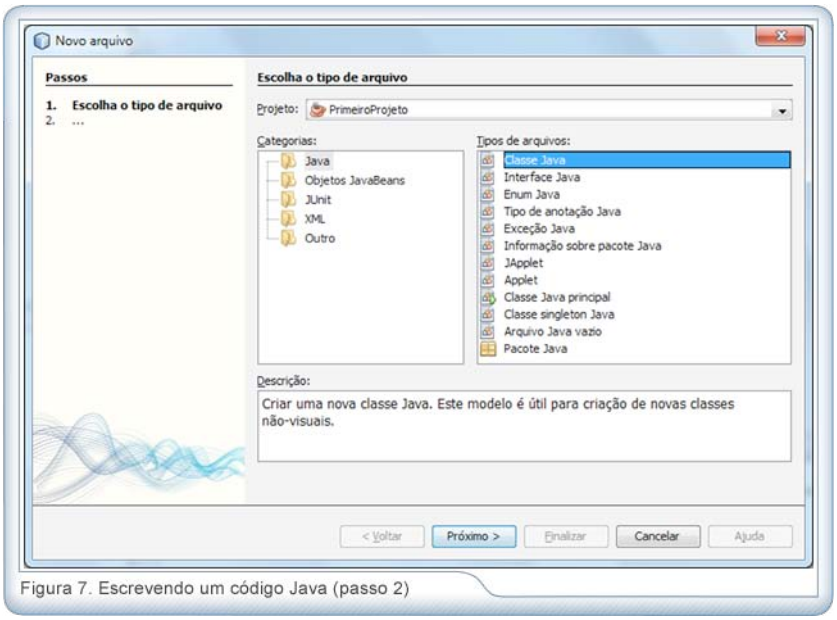


Figura 7. Escrevendo um código Java (passo 2)

- Coloque o nome da classe “PrimeiroPrograma” e pressione finalizar (Figura 8). O NetBeans acusa uma mensagem de Aviso informando que não é recomendado colocar Classes no pacote padrão, porém não se preocupe com isso pois veremos pacotes (um mecanismo de organização do código Java) em seguida;

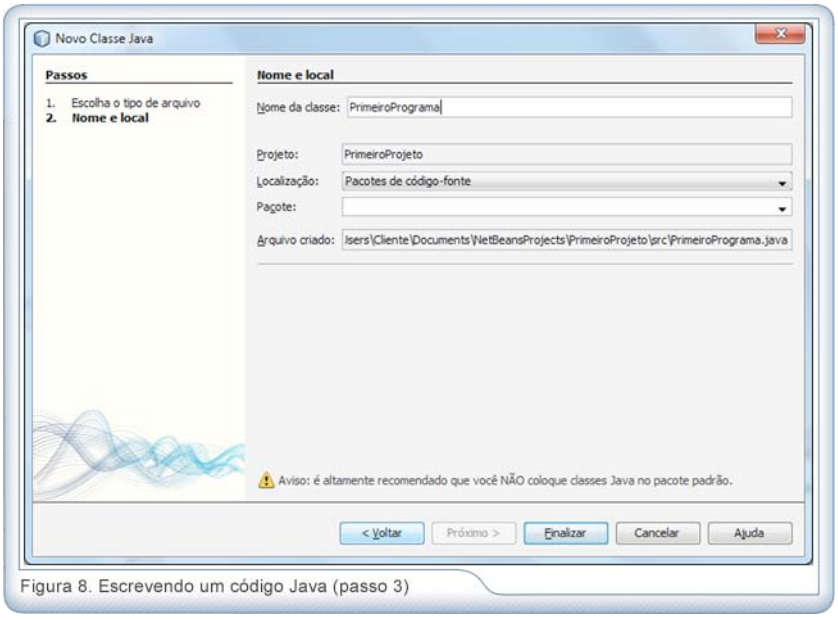


Figura 8. Escrevendo um código Java (passo 3)

- Após criado e aberto o projeto e o arquivo fonte na IDE, você pode visualizar os seguintes componentes (ver Figura 9):
  - A janela “Projetos” (1), que contém uma visão de árvore dos componentes do projeto, incluído os arquivos fonte, bibliotecas que o seu código depende, etc;
  - A janela de “Edição”(2) com o arquivo “PrimeiroPrograma.java” aberto;
  - A janela de “Navegação” (3), que você pode utilizar para acessar rapidamente entre os elementos de da classe selecionada;
- Escreva o seu programa na área de edição do programa como apresentado na Figura 9. O NetBeans já coloca a definição da classe para você (após salve o arquivo);

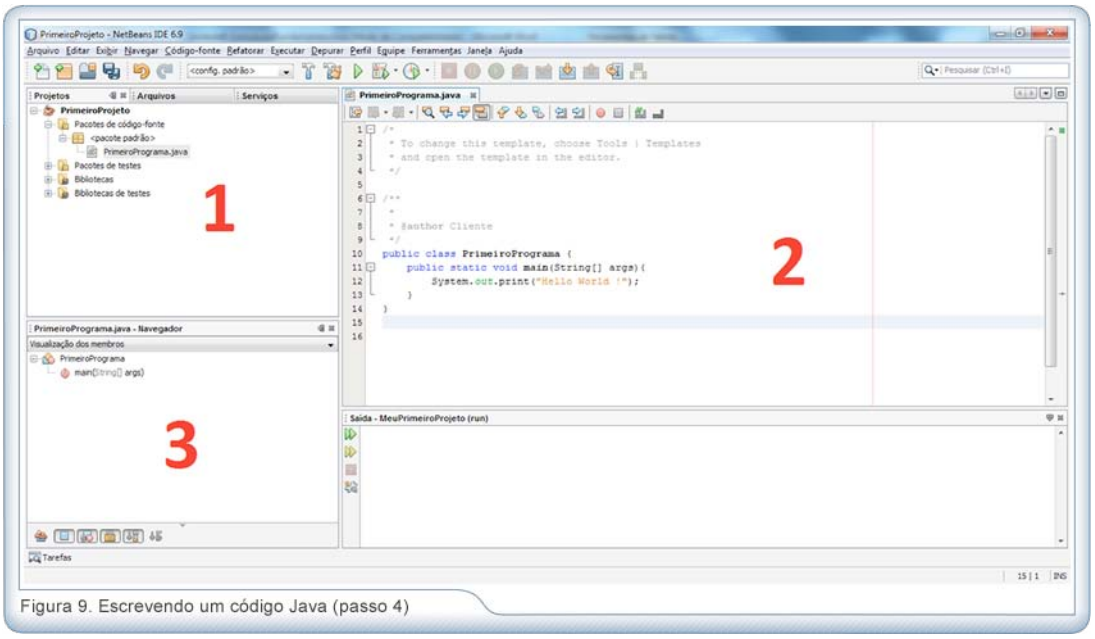


Figura 9. Escrevendo um código Java (passo 4)

Compilar e executar o programa

- Você pode executar o programa diretamente no NetBeans ou compilar o programa (gerar o arquivo PrimeiroPrograma.class) e depois executá-lo;
- Para executar o programa diretamente escolha no menu “Executar” | “Executar Main Project”. Neste momento, você deverá selecionar a classe principal “PrimeiroPrograma” e pressionar “OK”;
- O programa imprimirá no console de saída (Figura 10) a mensagem “Hello World”;
- **Parabéns, o seu primeiro programa funcionou!**

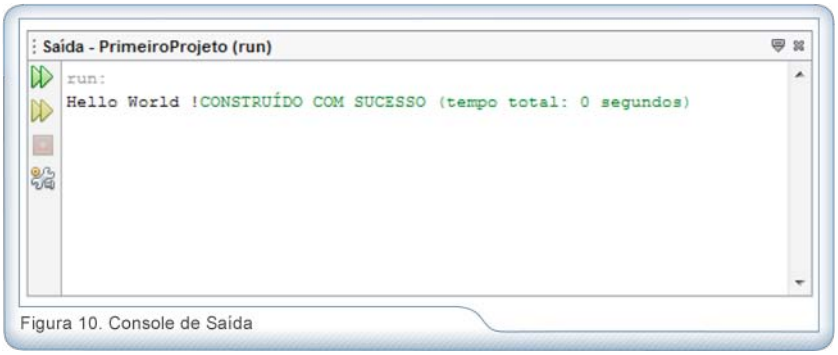


Figura 10. Console de Saída

Dicas:

- Você pode compilar o programa e depois executá-lo, para isso você deve ir no menu “Executar” | “Compilar File” (passo 1) e depois ir no menu “Executar” | “Executar Main Project”;
- Uma forma mais rápida de acessar o “Executar” | “Executar Main Project” é clicar em ;
- O NetBeans possui um mecanismo de checagem da sintaxe do código durante a edição. Por exemplo, na Figura 11 a palavra public foi digitada sem o “p” e a ferramenta indica um erro na linha 11.

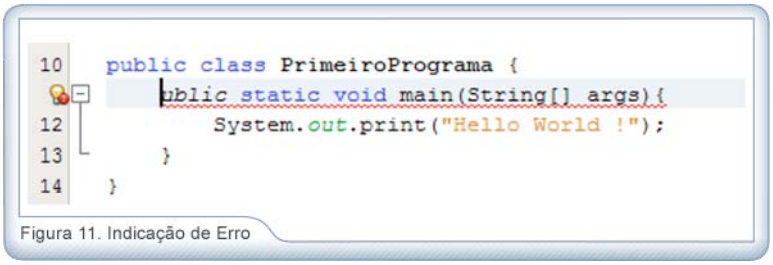


Figura 11. Indicação de Erro

Após a programação de uma pequena aplicação em Java, vamos conhecer as estruturas básicas da linguagem Java nas seções seguintes.

Comentários

A linguagem Java possui três diferentes tipos de comentários (sendo os dois primeiros tipos iguais ao da linguagem C):

1. **Comentário de Linha:** utiliza duas barras (//) e comenta até o final da linha;
2. **Comentário de Múltiplas Linhas:** os delimitadores /* */ permitem um comentário mais longo composto por várias linhas;
3. **Comentário de Documentação:** a linguagem Java possui um formato padrão para comentário de documentação. Através dos delimitadores /** */ é possível gerar (a partir do código fonte) um conjunto de páginas HTML que os descreve a classe e seus métodos (o nome da ferramenta é Javadoc).

Exemplos de comentários:

```
1.  /**
2.      Este é um comentário para gerar documentação
3.  */
4.  public class PrimeiroPrograma {
5.      public static void main(String[] args){
6.          System.out.print("Hello World !");
7.          // Este é um comentário de apenas uma linha
8.          /*
9.           Este é um comentário que pode ser utilizado em várias linhas
10.         */
11.     }
12. }
```

Tipos de Dados

A linguagem Java é uma linguagem fortemente tipada, isso significa que cada variável necessita de um tipo declarado. Existem oito tipos de primitivos (nativos da linguagem) para dados inteiros, em ponto flutuante, caractere e booleano.

Tipo Inteiros	Tamanho	Intervalo
Int	4 bytes	-2.147.483.648 a 2.147.483
Short	2 bytes	-32.768 a 32.767
Long	8 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
Byte	1 byte	-128 a 127
Tipo Ponto Flutuante	**Tamanho**	**Intervalo**
Float	4 bytes	aprox. +/- 3.402822347E+38
Doublé	8 bytes	aprox. +/- 1.79769313486231570E+308
Tipo Caractere	**Tamanho**	**Intervalo**
Char	2 bytes	Representa caracteres na codificação UNICODE
Tipo Booleano	**Tamanho**	**Intervalo**
Boolean	especificado pela JVM	Possui dois valores true e false

Observações:

- Você sempre deve escolher o tipo de dados mais adequado para cada tipo de variável;
- Números inteiros longos (long) possuem o sufixo L (por exemplo: 40000000L);
- Números hexadecimais possuem o prefixo 0x (por exemplo: 0x1FACA);

- Número octais possuem o prefixo 0 (por exemplo: 010 é 8 em decimal). Não recomendamos utilizar números octais pois podem causar confusão;
- Números do tipo float possuem sufixo F (minúsculo ou maiúsculo) (por exemplo: 3.54F);
- Números em ponto flutuante sem sufixo são considerados do tipo double (opcionalmente podem possuir a letra D – minúscula ou maiúscula – como sufixo).

Dicas:

- Procure saber mais da representação UNICODE na Internet ou em livros. A codificação UNICODE surgiu para as limitações dos esquemas tradicionais de codificação como o ASCII.

Variáveis e Operador de Atribuição

Em Java cada variável possui um tipo (a linguagem é fortemente tipada). Para declarar uma variável, você deve inserir o tipo de depois o nome da variável. Podem existir múltiplas declarações de variáveis na mesma linha (porém não é recomendado pelo padrão de codificação Java).

```
1.          int A;
2.          long populacao;
3.          float salarioMinimoRegional;
4.          boolean flag;
5.          int i,j;
```

Para inicializar uma variável Java, você deve explicitamente atribuir um valor através de uma instrução de atribuição. A instrução de atribuição em Java em é um sinal de igual (=).

```
1.          long populacaoBrasileira = 192304735; // declara e inicializa
2.          float salarioMinimoRegional; // declara
3.          salarioMinimoRegional = 546.57F; // inicializa
4.          flag = true; // inicializa uma variável booleana
```

Observações:

- NUNCA esqueça, comandos acabam com ponto-vírgula (;);
- De acordo com o padrão de codificação Java, uma variável sempre deve começar sempre por uma letra minúscula (porém é o Java permite declarações que iniciam em maiúsculas);
- Não é permitido o uso de variáveis não inicializadas (o compilador acusa erro);
- No Java você pode declarar variáveis em qualquer lugar do código.

Conversão entre Tipos Numéricos

Muitas vezes é necessário converter variáveis entre diferentes tipos numéricos. Existem dois tipos de conversão entre tipos: implícitas e explícitas. As conversões implícitas são as apresentadas pelas setas na Figura 12, as setas cheias indicam conversões em perda de precisão e as setas pontilhadas indicam que pode ocorrer perda de precisão.

```
1.          int i = 10;
2.          double j = i;
3.          long x = 100000;
4.          float y = x;
```

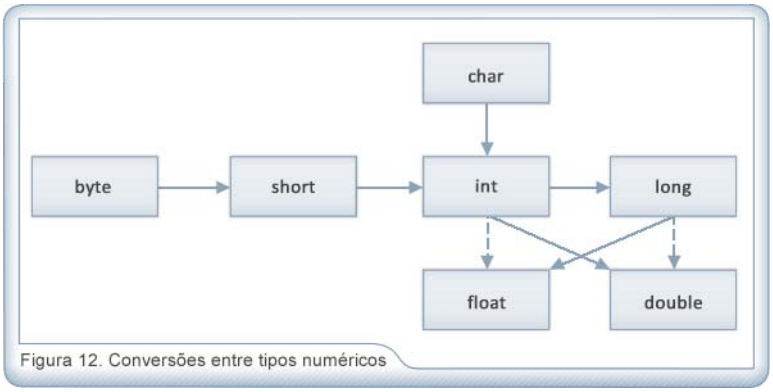


Figura 12. Conversões entre tipos numéricos

Às vezes é necessário converter algum número mesmo sabendo que poderá ser perdida alguma informação. Conversões numéricas onde a perda é possível são conhecidas como conversões explícitas e são realizadas através de casts. A sintaxe para realizar um cast é colocar o tipo entre parênteses seguido do nome da variável.

Exemplos de conversões numéricas explicítas (com cast):

```
1.          double x = 14.89;
2.           int y = (int) x;
3.          float z = 14.89f;
4.           long w = (long) z;
```

Operadores da Linguagem Java

Nesta seção, são apresentados os diferentes tipos de operadores em Java: aritméticos, incremento e decremento, relacionais, lógicos e bit a bit. Os operadores aritméticos são apresentados na Tabela 1.

Operador aritmético	Função	Sintaxe
+	Adição	op1 + op2
-	Subtração	op1 - op2
*	Multiplicação	op1 * op2
/	Divisão	op1 / op2
%	Resto da divisão inteira	op1 % op2

Tabela 1: Operadores aritméticos.

Exemplo com operadores aritméticos:

```
1.          int x = 5;
2.          int y = 4;
3.          resultadoSoma = x + y; // resulta em: 9
4.          resultadoSubtracao = x - y; // resulta em: 1
5.          resultadoMultiplicacao = x * y; // resulta em: 20
6.          resultadoDivisa = x / y; // resulta em: 1
7.          resultadoResto = x % y; // resulta em: 1
```

Os operadores aritméticos podem ser utilizados em conjunto com os operadores de atribuição. A Tabela 2 apre-

senta os operadores com atribuição, um exemplo e a respectiva equivalência.

Operador com atribuição	Exemplo	Equivalência
+=	x+=y;	x = x + y;
-=	x-=y;	x = x - y;
*=	x*=y;	x = x * y;
/=	x/=y;	x = x / y;
%=	x%=y;	x = x % y;

Tabela 2: Uso de operadores aritméticos com atribuição.

É conhecida dos programadores que uma das operações mais comuns é incrementar uma variável. Da mesma forma que o C e C++ utilizam operadores de incremento e decremento, o Java também define estes operadores. A Tabela 3 apresenta esses operadores e sua sintaxe.

Operadores		Sintaxe
Incremento	Pré	++op
	Pós	op++
Decremento	Pré	--op
	Pós	op--

Tabela 3: Operadores de incremento e decremento.

Exemplo com operadores de incremento e decremento:

```
1.          int m = 7;
2.          int n = 6;
3.          int a = 2 * ++m; // resulta em: a = 16, m = 8
4.          int b = 2 * --n; // resulta em: b = 10, n = 5
5.          m = 7;
6.          n = 6;
7.          int c = 2 * m++; // resulta em: d = 14, m = 8
8.          int d = 2 * n--; // resulta em: d = 12, n = 5
```

Os operadores relacionais e booleanos (apresentados nas Tabelas 4 e 5, respectivamente) são muito utilizados em estruturas de decisão e repetição.

Operador relacional	Função	Sintaxe
>	Maior	op1 > op2
<	Menor	op1 < op2
==	Igual	op1 == op2
>=	Maior Igual	op1 >= op2
<=	Menor Igual	op1 <= op2
!=	Diferente	op1 != op2

Tabela 4: Operadores relacionais.

Operador Booleano	Função	Sintaxe
&&	E lógico	op1 && op2
||	OU lógico	op1 || op2
!	Negação	! op

Tabela 5: Operadores Booleanos.

Os operadores de Bits são utilizados para manipular diretamente os bits de um número inteiro (apresentados na Tabela 6). Estes operadores são utilizados em operações específicas de manipulação de bits e durante a nossa disciplina não iremos utilizar estes operadores, porém é útil conheçamos.

Operador Booleano	Função	Sintaxe
&	AND	op1 & op2
|	OR	op1 | op2
^	XOR	op1 ^ op2
~	NOT	~op
>>	Desloca a direita	op1 >> op2
<<	Desloca a esquerda	op1 << op2
>>>	Desloca a direita em extensão de sinal	op1 >>> op2

Tabela 6: Operadores Bits.

Observação:

- Podem ser utilizados parênteses para indicar a precedência dos operadores.

Estruturas de Decisão

A linguagem Java possui estruturas de decisão como if-else e switch-case. A instrução condicional if-else possui a seguinte forma:

```
if (condição) {
    // código para condição verdadeira
    comandos;
}
```

Se a condição dentro dos parênteses (a condição sempre deve ser colocada dentro dos parênteses) for verdadeira a seqüência de comandos dentro do bloco (entre {}) deve ser executada. Um forma mais geral do if-else é a seguinte:

```
if (condição) {
    // código para condição verdadeira
    comandos;
} else {
    // código para condição falsa
    comandos;
}
```

Exemplo do if-else:

```
1.          if(x>y){
2.              System.out.println(x);
3.          } else {
4.              System.out.println(y);
5.          }
```

O exemplo apresenta uma condição simples, onde se x é maior que y o valor de x será impresso, caso contrário, o valor de y será impresso.

A estrutura condicional switch-case fornece uma construção para múltiplas seleções. Nos casos que existem muitas alternativas, o uso do if-else pode ser trabalhoso e neste caso o switch-case é uma opção mais fácil. O switch-case possui a seguinte forma:

```
switch (expressão) {
case valor1:
    // código caso expressão seja igual valor1
    comandos;
    break;

case valor2:
    // código caso expressão seja igual valor2
    comandos;
    break;

case valor3:
    // código caso expressão seja igual valor3
    comandos;
    break;

default:
    // código caso a expressão seja diferente de
    // todos os outros valores anteriores
    comandos;
    break;
}
```

Exemplo do switch-case:

```
1.      switch (mes) {
2.          case 1:
3.              System.out.println("Janeiro");
4.              break;
5.          case 2:
6.              System.out.println("Fevereiro");
7.              break;
8.          case 3:
9.              System.out.println("Março");
10.             break;
11.         default:
12.             System.out.println("Não é o mês de Jan, Fev ou Mar");
13.             break;
14.     }
```

No exemplo acima, a execução inicia no rótulo *case* que corresponde ao valor em que a seleção é realizada e continua até que o próximo comando *break* seja encontrado. Caso a variável *mes* não tenha correspondência em nenhum dos *cases* a cláusula default será executada.

Estruturas de Repetição

A linguagem Java possui três estruturas básicas de repetição como o for, while e do-while. As estruturas de repetição, como o seu próprio nome já fala, permitem a repetição (execução repetida) de um comando ou bloco de comandos. A instrução while possui a seguinte forma:

```
while (condição) {

    comandos;

}
```

O loop (laço) será executado enquanto a condição dentro dos parênteses for verdadeira. No exemplo a seguir, o laço será executado enquanto o valor de *k* for maior do que 10 (imprimindo os número de 10 a 1).

Exemplo do while:

```
1.      int k = 10;
2.      while(k>0){
3.          System.out.println(k);
4.          k--;
5.      }
```

A construção do-while é muito similar a estrutura while, porém o while sempre testa a condição na parte superior. Algumas vezes é interessante que o bloco seja executado pelo menos uma vez, assim você necessitará mover o teste para a parte inferior. Isso é feito com o laço do-while que possui a seguinte forma:

```
do {

    comandos;

} while (condição);
```

No exemplo a seguir, o laço com o do-while não faz muito sentido, pois a variável teste possui o valor zero. Contudo, este exemplo mostra que a condição somente será testada no final, ou seja, os comandos do bloco (impressão) serão executados antes do teste.

Exemplo do do-while:

```
1.      int teste = 0;
2.      do {
3.          System.out.println("Pelo menos entrou uma vez no laço !");
4.      } while (teste!=0);
```

A construção for é a construção geral para suportar a iteração controlada por uma variável de contador ou algo semelhante que é atualizada depois de cada iteração. O for possui a seguinte forma:

```
for (inicialização; condição; atualização) {

    comandos;

}
```

Normalmente o primeiro *slot* armazena a inicialização do contador. O segundo *slot* fornece a condição que será testada a cada passagem no laço e o terceiro *slot* apresenta a forma como será atualizado o contador. O exemplo a seguir, mostra o código que faz dez iterações imprimindo do valor de 0 a 9.

Exemplo do for:

```
1.      for(int i=0;i<10;i++){
2.          System.out.println(i);
3.      }
```

Observações:

- Existem duas instruções de controle de fluxo, a instrução break (que já utilizamos no switch-case) e a instrução continue;
- A instrução break interrompe estruturas while, for, do/while ou switch e a execução continua com a primeira instrução depois da estrutura;
- A instrução continue quando executada em uma estrutura while, for ou do/while, pula as instruções restantes no corpo dessa estrutura e prossegue com a próxima iteração do laço.

Arrays

Um array é uma estrutura de dados homogênea que permitem agrupar coleções de valores do mesmo tipo. Os arrays possuem algumas propriedades que podemos destacar:

- Um array inicia na posição 0 (igual a linguagem C) e podem ser unidimensionais ou multidimensionais;
- Apresentam limitações em função da estrutura estática: não se muda o seu tamanho facilmente e uma inserção ou remoção no meio do array exige uma rotina para reorganização da estrutura;
- Arrays também são objetos e por isso necessitam ser instanciados (estudaremos o que são objetos na próxima unidade);
- Arrays podem ser utilizados como argumentos e valores de retornos em métodos.

A criação de um array possui duas sintaxes possíveis:

```
<tipo> <nome>[] = new <tipo>[<tamanho>]

<tipo>[] <nome> = new <tipo>[<tamanho>]
```

Normalmente é utilizada a segunda notação, onde os colchetes aparecem depois do tipo. A criação de um array é dividida em duas partes: declaração e instanciação. A declaração de uma variável array é realizada especificando o tipo e o nome. Porém, é necessário também inicializar a variável com um array real, para isto é feito o uso do operador *new* (explicaremos este operador com mais detalhes na próxima unidade).

Exemplos de criações de arrays:

```
1.    int valores[]; // apenas declaração
2.    valores = new int[5]; // instanciação (inicialização com um array real)
3.    int valores2[] = new int[5]; // Declaração  e instanciação
4.    char vogais[] = new char[5];
5.    Char[] vogais2 = new char[5];
```

Nas linhas 1 e 2 é criado uma variável de nome valores com um array de 5 números inteiros (o mesmo é feito na linha 3 para a variável valores2). Já as linhas 4 e 5 definem arrays de cinco caracteres.

Também é possível inicializar um array com valores determinados. O exemplo a seguir mostra a criação juntamente com a inicialização de dois arrays.

Exemplos de inicializações de arrays:

```
1.     char vogais[] = {'a', 'e', 'i', 'o', 'u'};
2.     Int valores[] = {0, 1, 2, 3, 4};
```

O acesso aos elementos do array é feito com [<índice do array>]. Além disso, os arrays possuem um campo *length* que determina o tamanho do array. O exemplo a seguir mostra um exemplo completo (dentro do método *main*) de uma criação de um array de 6 valores, impressão do seu tamanho (o tamanho é impresso através do <nome da variável>.lenght) e após a alteração de dois valores do array (linhas 5 e 6) e por fim, a impressão dos valores na tela (linhas 8 a 10).

Exemplo manipulação de arrays:

```
1.  public class TesteArray {
2.          public static void main(String[] args){
3.                  int valores[] = {10, 20, 30, 40, 50, 60};
4.                  System.out.println("O tamanho: " + valores.length);
5.                  valores[1]= 25;
6.                  valores[3]= 45;
7.                  // imprimindo valores
8.                  for(int i=0; i<valores.length; i++){
9.                     System.out.println(valores[i]);
10.                 }
11.         }
12. }
```

Os arrays multidimensionais (com mais de uma dimensão) são criados da seguinte forma:

```
<tipo> <nome>[][] = new <tipo>[<tamanho>] [<tamanho>]

<tipo>[][] <nome> = new <tipo>[<tamanho>] [<tamanho>]
```

Exemplos de criações de arrays multidimencionais:

```
1.      // matriz de 5 linhas e 5 colunas
2.      int matriz1[][] = new int[5][5];
3.      // matriz de 2 linhas e 3 colunas já inicializada
4.      Int matriz2[][] = {{1,2,3}, {4,5,6}};
```

Para acessar um array de duas dimensões (exemplo a seguir), são necessários dois laços aninhados.

Exemplo de manipulação de arrays multidimencionais:

```
1.          // matriz de 4 linhas e 3 colunas
2.          int matriz[][] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
3.          // imprimindo matriz
4.          for(int i=0; i<matriz.length; i++){
5.              System.out.println("-----------");
6.              for(int j=0; j<matriz[i].length; j++){
7.                  System.out.println(matriz[i][j]);
8.              }
9.          }
```

Na verdade, a matriz é um vetor que aponta para outros vetores. A declaração a seguir é representada pela Figura 13.

```
int matriz[][] = new int[3][2];
```

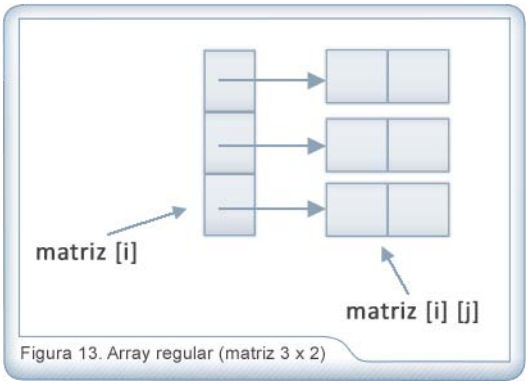


Figura 13. Array regular (matriz 3 x 2)

A linguagem Java permite a criação de arrays irregulares. A declaração a seguir conduz a um array irregular representado pela Figura 14.

```
int matriz[][] = new int[3][];

matriz[0] = new int[1];

matriz[1] = new int[2];

matriz[2] = new int[3];
```

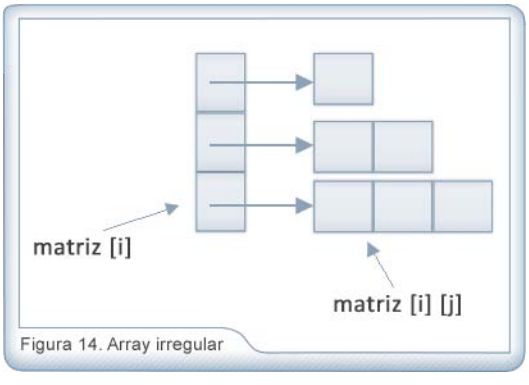


Figura 14. Array irregular

Você pode copiar uma variável array para outra, mas neste caso ambas as variáveis irão se referir ao mesmo array. No exemplo: Exemplo a seguir, na atribuição na linha 6 as variáveis referenciam o mesmo array. Desta maneira, se o vetor1 for alterado o vetor2 também sentirá esta mudança, pois as duas variáveis se referem ao mesmo vetor conforme apresentado na Figura 15.

Exemplo de variáveis referenciando o mesmo array:

```
1.  public class ArrayTest {
2.          public static void main(String[] args) {
3.              int vetor1[] = {1, 3, 5, 7, 9, 11};
4.              int vetor2[] = {21, 23, 25, 27, 29, 31};
5.              // apenas atribuição
6.              vetor2 = vetor1;
7.              // imprimindo vetores
8.              for(int i=0; i<vetor1.length; i++){
9.                 System.out.println(vetor1[i]);
10.             }
11.             for(int i=0; i<vetor2.length; i++){
12.                System.out.println(vetor2[i]);
13.             }
14.         }
15. }
```

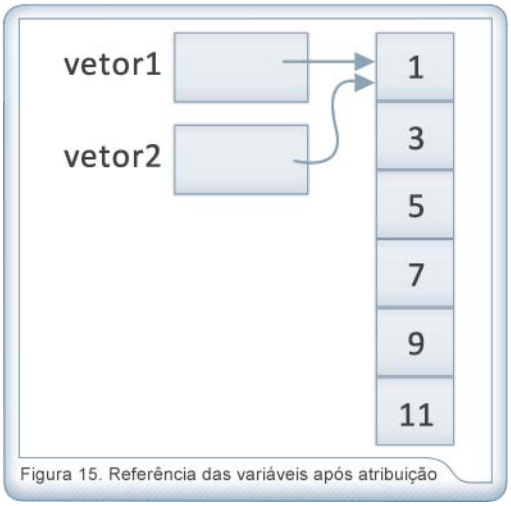
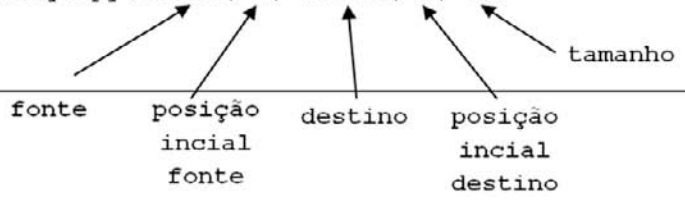


Figura 15. Referência das variáveis após atribuição

Para realizar uma cópia entre arrays é necessário utilizar um método da classe Arrays (o arraycopy). O exemplo a seguir mostra o código que realiza a cópia dos arrays.

Exemplo que realiza a cópia de arrays:

```
1.  public class ArrayTest {
2.          public static void main(String[] args) {
3.              int vetor1[] = {1, 3, 5, 7, 9, 11};
4.              int vetor2[] = {21, 23, 25, 27, 29, 31};
5.              // copiando com arraycopy
6.              System.arraycopy(vetor1, 0, vetor2, 0, 6);
7.              //...
8.          }
9.  }
```



Executando Programas Java no NetBeans

Você pode possuir vários arquivos fonte (classes Java) no mesmo projeto no NetBeans. Para testar a classe ArrayTest (apresentada a seguir) você deve:

- Criar um arquivo com o mesmo nome da classe Java (ArrayTest.java), para isso vá no menu “Arquivo” | “Novo Arquivo” e selecione Categoria “Java” | Tipo de Arquivo “Java” e coloque o mesmo nome da classe (ArrayTest);
- Após copie e cole o programa abaixo, sobrescrevendo o código anterior;
- Para executar este novo programa, você deve selecionar o arquivo na área de Projetos (clicar em cima do arquivo) e ir no menu “Arquivo” | “Executar arquivo” (conforme apresentado na Figura 16);
- O arquivo será executado com sucesso.

Exemplo para execução no NetBeans:

```
1.  public class ArrayTest {
2.          public static void main(String[] args) {
3.              int vetor1[] = {1, 3, 5, 7, 9, 11};
4.              int vetor2[] = {21, 23, 25, 27, 29, 31};
5.              // apenas atribuição
6.              vetor2 = vetor1;
7.              // imprimindo vetores
8.              for(int i=0; i<vetor1.length; i++){
9.                  System.out.println(vetor1[i]);
10.             }
11.             for(int i=0; i<vetor2.length; i++){
12.                 System.out.println(vetor2[i]);
13.             }
14.         }
15. }
```

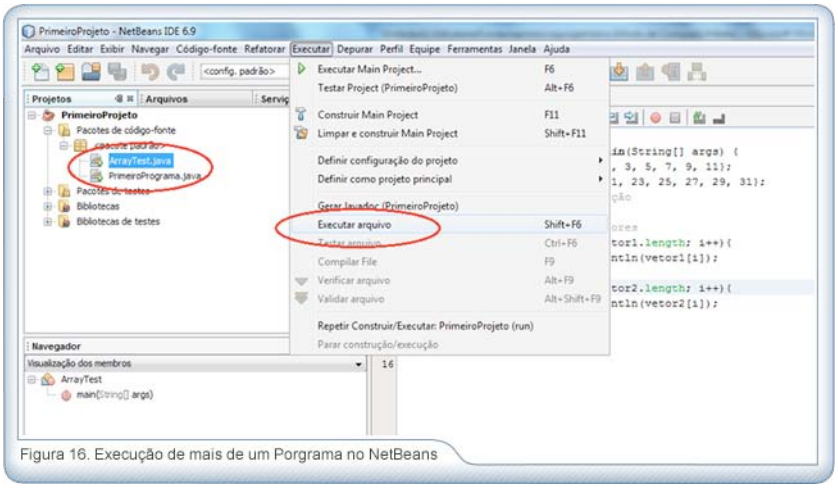


Figura 16. Execução de mais de um Porgrama no NetBeans

Conclusão

Nesta unidade, foram apresentadas a linguagem Java e a ferramenta NetBeans. Você viu as estrtuturas fundamentais da linguagem Java e a mecânica básica de compilação e execução de programas no NetBeans. Na próxima unidade, iniciaremos a estudar os conceitos de orientação a objetos e seu uso em Java.

UNIDADE B

CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS EM JAVA

Nesta unidade, estudaremos os principais conceitos envolvidos na Programação Orientada a Objetos e aplicaremos estes conceitos na linguagem de Programação Java.

Conceitos de Orientação a Objetos

A ideia é apresentar os principais conceitos envolvidos no paradigma orientado a objetos. Esta seção mostra apenas os aspectos teóricos da orientação a objetos e a partir da próxima seção estes conceitos são aplicados na linguagem Java.

A programação orientada a objetos (POO) é o paradigma de programação dominante nos dias de hoje, tendo substituído as técnicas de programação "estruturadas" que foram desenvolvidas na década de 1970. Java é uma linguagem orientada a objetos e para você ser um bom programador Java deve conhecer e entender bem o paradigma orientado a objetos.

Um programa orientado a objetos é composto de **objetos** que colaboram entre si para a realização de tarefas. Entender os objetos é a chave para entender a tecnologia orientada a objetos. Olhe a sua volta e você encontrará muitos exemplos de objetos do mundo real: o seu gato, a sua mesa, a sua TV e o seu carro. Estes objetos reais compartilham duas características: eles possuem um **estado** e um **comportamento**. O seu gato possui estado (nome, pelagem, raça) e comportamento (miam, andam, se lambem). O carro também possui estado (marca, tipo, cor, ano, velocidade atual) e comportamento (aumentar a velocidade, diminuir a velocidade, trocar de marcha).

Identificar o estado e o comportamento para objetos do mundo real é uma boa maneira de iniciar a pensar em termos de programação orientada a objetos. Uma lâmpada pode conter dois estados (ligada e desligada) e dois comportamentos (ligar e desligar), já um rádio pode conter estados adicionais (ligado, desligado, volume atual, estação atual) e pode possuir outros comportamentos (ligar, desligar, aumentar volume, diminuir volume, procurar uma estação).

O que acabamos de fazer foi construir um modelo simplificado de uma lâmpada e um rádio. Também notamos que os objetos variam em complexidade, existem uns mais complexos que outros. Você também pode notar que alguns objetos, em geral, são compostos por outros objetos. Estas observações do mundo real podem ser traduzidas para a programação orientada a objetos.

O paradigma de programação orientado a objetos é o paradigma de programação que utiliza objetos (que possuem um estado e comportamento), criados a partir de modelos, para representar e processar dados usando programas de computadores. As técnicas de programação incluem características como **abstração**, **encapsulamento**, **modularidade**, **polimorfimo** e **herança**. Estes conceitos serão estudados durante esta unidade.

Classes

Os programadores que utilizam o paradigma orientado a objetos criam e usam objetos a partir de classes, que são relacionadas diretamente com modelos. Desta forma, **classes** são estruturas das linguagens de programação OO para conter, para determinado modelo, os dados que devem ser representados e as operações que devem ser efetuadas com estes dados (estado e comportamento). Cada classe deve ter um nome que seja facilmente

associado ao modelo que a classe representa.

Objetos

Os objetos possuem um estado e comportamento e são criados a partir de um modelo (a classe). Para a representação de dados específicos usando classes será necessária a criação de **objetos** ou **instâncias** desta classe. Um **objeto** ou **instância** é a materialização de uma classe (modelo utilizado para representar dados e executar ações). Para que os objetos ou instâncias possam ser manipulados, é necessária a criação de **referências** a estes objetos, que são basicamente variáveis do "tipo" classe.

Como já apresentado, um objeto possui um estado e um comportamento. Um objeto armazena o seu estado em atributos (possíveis informações armazenadas no objeto) e disponibiliza o seu comportamento através dos métodos (procedimentos que formam os comportamentos e serviços oferecidos por um objeto de uma classe).

Atributos

Os dados contidos em uma classe são conhecidos como **campos** ou **atributos** (variáveis em algumas linguagens de programação) daquela classe. Cada campo deve ter um nome e tipo, que será ou um tipo de dado nativo da linguagem ou uma classe existente na linguagem ou definida pelo programador. Se a classe é usada para que várias instâncias sejam criadas a partir dela, cada uma dessas instâncias terá valores próprios para cada atributo.

Métodos

Os **métodos** são definidos na declaração de uma classe e definem o comportamento (operações) dos objetos daquela classe (funções em algumas linguagens de programação). Estes métodos operam sobre os atributos internos e servem como mecanismo primário para comunicação entre objetos. Métodos são geralmente chamados ou executados explicitamente a partir de outros trechos de código na classe que o contém ou a partir de outras classes. Em um programa OO, os objetos de um sistema **trocam mensagens** para que as tarefas sejam realizadas. Esta troca de mensagens, na verdade, são chamadas de métodos.

A Figura 1 apresenta um exemplo da relação de classe e objeto. A classe Carro define um modelo de um objeto Carro, definindo o seu estado (atributos) e seu comportamento (métodos). A materialidade da classe Carro é representada pelas instâncias. A instanciação é o processo de criação de um objeto e implica em alocação de um espaço em memória para armazenar este objeto. Na Figura 1, o Fusca, Gallardo, Kombi e Aurea são instâncias da classe Carro e cada instância possui os seus valores próprios dos atributos.

Figura 1. Exemplo de classe, objetos e atributos

A programação orientada a objetos fornece entre outras características a abstração. A **abstração** é um mecanismo ou prática que permite reduzir ou esconder detalhes que não são importantes em um dado momento. A grande vantagem da abstração é permitem gerenciar a complexidade e concentrar a atenção nas características essenciais de um objeto, por exemplo. O encapsulamento, polimorfirmo e herança são técnicas que permitem a abstração.

Encapsulamento

O encapsulamento é um conceito chave para trabalhar com orientação a objetos. Os objetos possuem comportamento que diz respeito a operações realizadas por um objeto e também ao modo pelo qual essas operações são executadas. O **mecanismo de encapsulamento** é uma forma de restringir o acesso ao comportamento interno de um objeto, ou seja, caso um objeto necessite de colaboração de outro objeto para realizar uma tarefa, ele simplesmente envia uma mensagem a este último.

O encapsulamento nada mais é do que **combinar dados e comportamento especificados num mesmo módulo** e ocultar os detalhes de implementação do objeto. Isso permite que cada objeto envie mensagens a outros objetos para realizar certas tarefas, sem se preocupar como as tarefas são realizadas. A única coisa que um objeto precisa saber para pedir colaboração de outro objeto é conhecer a sua interface. Um objeto possui uma interface que define os serviços que ele pode realizar e consequentemente as mensagens que ele recebe.

A Figura 2 apresenta o que o encapsulamento fornece: a implementação fica oculta para os outros objetos que conhecem apenas a interface do objeto e acessam através das mensagens (chamadas de métodos).

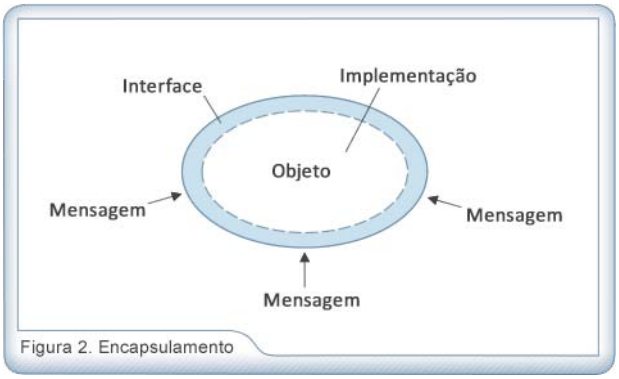


Figura 2. Encapsulamento

Herança

A herança é outra forma de abstração em orientação a objetos onde classes semelhantes são agrupadas em hierarquias. Cada nível de hierarquia pode ser visto como um nível de abstração, onde cada classe em um nível da hierarquia herda as características das classes dos níveis acima.

Esse mecanismo, também conhecido como generalização/especialização, facilita o compartilhamento de comportamento comum entre um conjunto semelhante de classes. Diferenças ou variações de uma classe em particular podem ser organizadas de forma mais clara.

A Figura 3 apresenta um exemplo de herança, onde a classe *Figura* é a classe Figura mais genérica e pode ser especializada em Linha ou Figura Geométrica. Já as classes Círculo e Quadrado são um tipo de Figura Geométrica. A classe mais geral é chamada de superclasse. A classe mais especializada é herda da subclasse é chamada de subclasse.

A herança permite estender classes existentes adicionando atributos e métodos, porém os atributos e métodos da superclasse são herdados e podem ser utilizados na subclasse. Uma das grandes vantagens da herança é a reutilização de código. Existem dois tipos de herança: simples, onde um objeto herda diretamente de uma classe apenas e múltipla, onde um objeto herda de duas ou mais superclasses. Java possui apenas o mecanismo de herança simples.

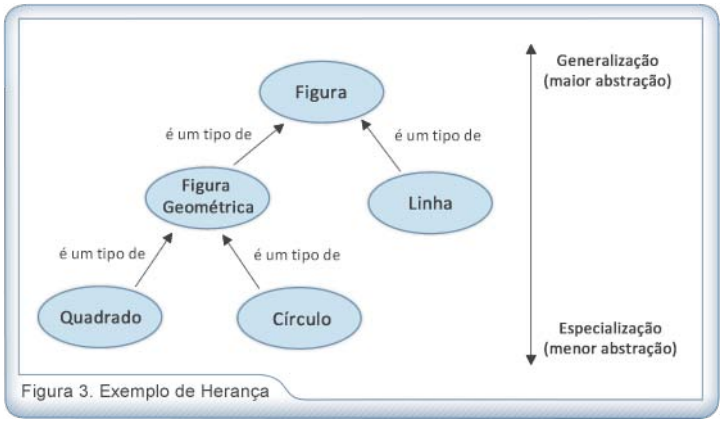


Figura 3. Exemplo de Herança

Polimorfismo

O polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface. A mesma computação funciona para objetos de muitas formas e adpta-se a natureza dos objetos.

O mecanismo de herança permite a criação de classes a partir de outras já existentes. O polimorfismo permite que duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para cada classe derivada, usando para isso uma referência a um objeto do tipo da superclasse. A decisão sobre qual o método que deve ser selecionado é tomada em tempo de execução através do mecanismo de ligação dinâmica que considera o tipo do objeto.

Esta seção apresentou os conceitos básicos sobre orientação a objetos. Na próxima seção iniciaremos a aprender como estes conceitos podem ser utilizados na linguagem de programação Java.

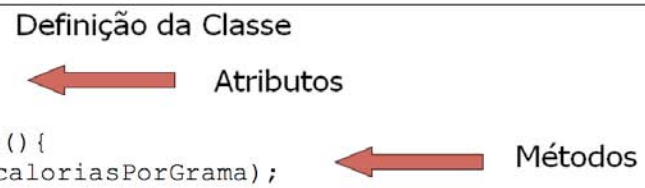
Classes, métodos e atributos em Java

Como vimos na seção anterior, classes são estruturas que contem, para determinado modelo, os dados que devem ser representados e as operações que devem ser efetuadas com estes dados (é um molde). Um programa Java é composto de uma ou, normalmente, várias classes. Tudo em Java deve ser colocado dentro de uma classe. A Figura 4 apresenta a organização de uma classe (como já apresentado na unidade A). Uma classe pode conter um ou mais métodos. Lembre-se que pelo padrão de codificação, nomes de classe sempre devem iniciar por letra maiúscula.

Figura 4. Organização de uma classe

A apresentação das classes, atributos e métodos, será realizada através de exemplos. O código a seguir apresenta a definição da classe **Fruta**. Esta classe Fruta possui dois atributos (**gramas e caloriasPorGrama** do tipo inteiro) e um método chamado *totalCalorias*

```
1.  public class Fruta {
2.          int gramas;
3.          int caloriasPorGrama;
4.
5.          public int totalCalorias(){
6.                  return (gramas*caloriasPorGrama);
7.          }
8.  }
```



Na definição da classe *Fruta* foi utilizada a palavra reservada *public*, que significa que a classe pode ser visualizada por outras classes. Foram declarados dois atributos na classe e um método.

Existem quatro tipos de variáveis na linguagem Java:

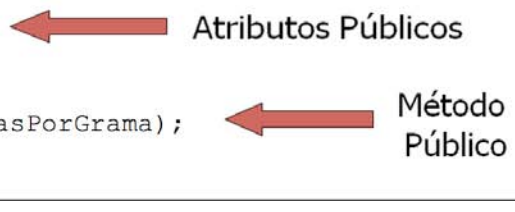
- Variáveis de instância (ou atributos de objetos): onde os objetos armazenam os seus estados. Os valores armazenados nestas variáveis são únicos para cada objeto;
- Variáveis de Classe (ou atributos de classe): uma variável que é única para todos os objetos (iremos apresentar mais na frente);
- Variáveis Locais: declaradas dentro de métodos. São similares as variáveis de instância, porém possuem visibilidade dentro do método que é declarada;
- Parâmetros: variáveis utilizadas dentro como parâmetros em métodos.

As variáveis *gramas* e *caloriasPorGrama* são consideradas variáveis de instância. Java permite a restrição ao acesso a atributos (campos) e métodos de classe por intermédio de **modificadores de acesso** que são declarados dentro das classes, antes dos campos e métodos. São quatro tipos de modificadores de acesso:

- **public** (público): garante que o campo ou método da classe poderá ser acessado ou executado a partir de qualquer classe;
- **private** (privado): os campos e métodos podem ser acessados apenas pela própria classe;
- **protected** (protegido): funciona como o modificador private, exceto que as classes herdeiras ou derivadas também terão acesso aos campos e métodos (será visto nas próximas seções);
- *default*: campos e métodos declarados sem modificador são visíveis (podem ser acessados) por todas as classes dentro de um mesmo pacote (será visto nas próximas seções).

O código a seguir mostra o código da classe Fruta com atributos públicos e método também público. Isso significa que os atributos e o método são visíveis (podem ser acessados e modificados por qualquer classe). Isso é ruim, pois não propicia o encapsulamento, pois não restringe o acesso ao comportamento interno de um objeto.

```
1.  public class Fruta {
2.          public int gramas;
3.          public int caloriasPorGrama;
4.
5.          public int totalCalorias(){
6.                  return (gramas*caloriasPorGrama);
7.          }
8.  }
```



Para solucionar esse problema, podemos declarar os atributos como privados. Desta forma, o encapsulamento é propiciado. Cabe salientar, que o método continua público, pois desejamos que outras classes acessem este método, porém em uma classe podem existir métodos privados (que realizam apenas computações internas a classe). O código a seguir mostra a classe Fruta com atributos privados.

```
1.  public class Fruta {
2.          private int gramas;
3.          private int caloriasPorGrama;
4.
5.          public int totalCalorias(){
6.                  return (gramas*caloriasPorGrama);
7.          }
8.  }
```

Atributos Privados

Método Público

Um método pode ser dividido em duas partes: declaração e definição. A declaração define a interface do método e é composta por um valor de retorno, nome do método e parâmetros. Já a definição possui o corpo (código do método em si). Embora o programador tenha liberdade para criar os identificadores (nomes) dos métodos, existem algumas convenções que são seguidas pelos programadores Java. Quando é necessário uma classe oferecer acesso a um atributo, seja para obter o valor do atributo, seja para armazenar um valor em um atributo o mais adequado é fornecer através dos métodos conhecidos como *getters* e *setters*. O código a seguir mostra o exemplo de métodos *getters* e *setters.*

```
1.          public int getGramas() {
2.              return gramas;
3.          }
4.          public void setGramas(int valor) {
5.              gramas = valor;
6.          }
```

A declaração de um método é realizada da seguinte forma:

<acesso><tipo><nome>(<parâmetros>)

Os métodos podem possuem modificadores de acesso e podem ser: **públicos**, onde podem ser acessados a partir de qualquer classe e **privados**, onde só podem ser acessados a partir de métodos da própria classe que a pertencem. A assinatura de um método é composta do nome mais tipos e números de parâmetros (indenpendente do nome das variáveis). O código a seguir mostra exemplos de declarações de métodos.

```
1.      void metodo1() // não têm parâmetro e não retorna nada
2.      public int metodo2() // público, não têm parâmetro e
3.                           // retorna um dado inteiro
4.      private int metodo3(int valor) // privado, possui um inteiro como
5.                                     // parâmetro e retorna um inteiro
```

Uma pergunta que resta: quando devemos utilizar atributos e métodos privados ? A Figura 5 apresenta resume esta pergunta. Atributos e métodos privados devem ser utilizados dentro da classe. Já métodos públicos devem ser utlizados como interface para iteração com o objeto (não é aconselhável o uso de atributos públicos).

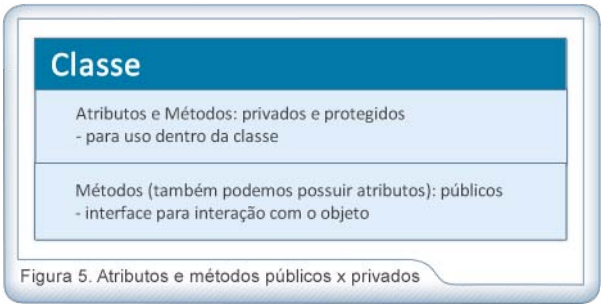


Figura 5. Atributos e métodos públicos x privados

A relação entre os atributos, métodos público e privados e as mensagens pode ser visto na Figura 6. Os métodos públicos servem como elemento de comunicação do objeto e os métodos privados e atributos ficam ocultos dentro do objeto fortalecendo o princípio do encapsulamento.

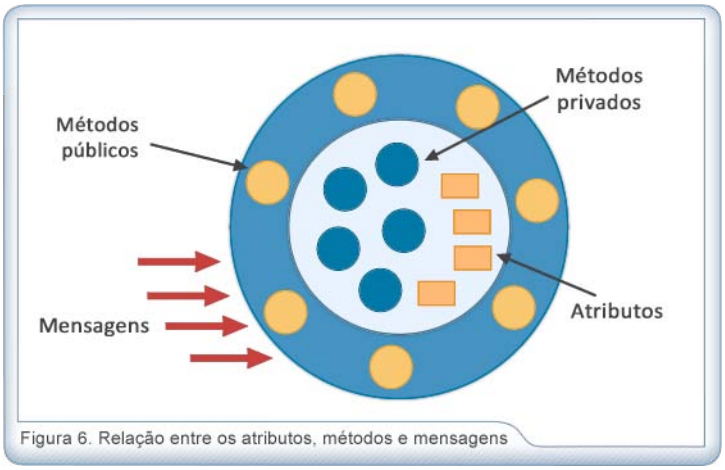


Figura 6. Relação entre os atributos, métodos e mensagens

Manipulando Objetos em Java

Como já apresentado, os objetos possuem um estado e comportamento e são criados a partir de um modelo (a classe). Um **objeto** ou **instância** é a materialização de uma classe.

A criação de um objeto possui duas partes: declaração da variável que referencia o objeto e a instanciação (criação propriamente dita) do objeto. Na declaração é feita a associação do tipo da classea um nome de variável (variável de objeto). Após, na instanciação é realizada a criação de um novo objeto de uma classe com o operador *new*.

O código a seguir apresenta na linha 4 a declaração de uma variável do tipo *Fruta* com nome *pera* e na linha 6 a instanciação (criação) de um objeto da classe *Fruta* com o operador *new* (neste momento esta estrutura de dados é alocada na memória). A linha 8 mostra a declaração e instanciação na mesma linha.

```
1.  public class TestaFruta {
2.      public static void main(String[] args) {
3.
4.          Fruta pera;          <- Declarando
5.
6.          pera = new Fruta();  <- Instanciando
7.
8.          Fruta uva = new Fruta();  <- Declarando e Instanciando
9.
10. }
```

Após a instanciação de dois objetos da classe Fruta, cada uma das variáveis (*pera e uva*) fazem referência a um objeto na memória conforme apresentado na Figura 7. Cabe salientar que, os conteúdos dos atributos de ambos objetos não possuem dados algum.

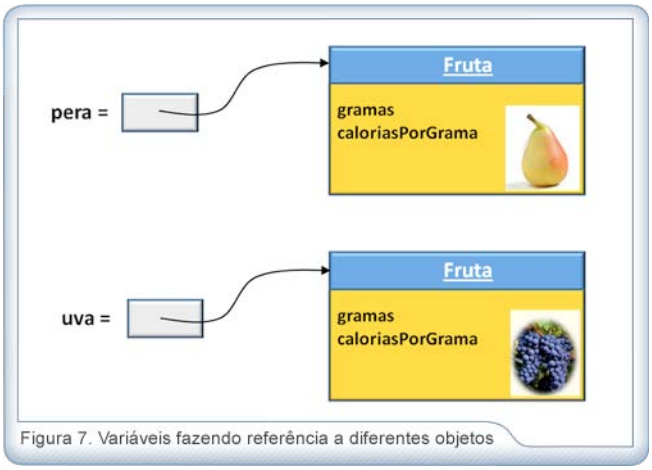


Figura 7. Variáveis fazendo referência a diferentes objetos

Dica:

Para facilitar o teste das suas classes, crie uma nova classe Java (um novo arquivo) chamada, por exemplo, de *TestaFruta* com o método *main* (para poder executar as suas aplicações). Agora, a sua aplicação já possui duas classes (classe *Fruta* e *TestaFruta*).

A chamada de métodos de um dado objeto é realizada da seguinte forma:

```
<variável de objeto>.<nome do método>(<parâmetros>)
```

O código a seguir apresenta a chamada de métodos os objetos referenciados pelas variáveis pera e uva.

```
1.      public class TestaFruta {
2.          public static void main(String[] args) {
3.              Fruta pera = new Fruta();
4.              Fruta uva = new Fruta();
5.
6.              int calPera, calUva;
7.              calPera = pera.totalCalorias();
8.              calUva = uva.totalCalorias();
9.  }
```

Chamando métodos

O acesso direto a atributos de um objeto (variáveis de instância) pode ser realizado da seguinte forma:

```
<variável de objeto>.<nome do atributo do objeto>
```

Contudo, essa forma de acesso somente poderá ser realizada se o atributo for declarado como público. Porém não é adequado declarar atributos como públicos. O código a seguir mostra o exemplo do acesso ao atributo gramas do objeto Fruta referenciado pela variável pera.

```
1.      public class TestaFruta {
2.          public static void main(String[] args) {
3.              Fruta pera = new Fruta();
4.              Fruta uva = new Fruta();
5.
6.              pera.gramas = 150;
7.
8.  }
```

Acesso Legal (sendo o atributo público)

Existe um operador em Java que serve para verificar se um objeto é instância de uma determinada classe. O operador instanceOf retorna uma variável booleana. A sintaxe é a seguinte:

```
<referência à instância> instanceof <nome da classe> : boolean
```

O código a seguir mostra um exemplo testando se a variável pera é uma instância da classe *Fruta*.

```
1.  if (pera instanceof Fruta) {
2.          System.out.println("pera é uma instância de Fruta");
3.  }
```

Os objetos podem ser utlizados como parâmetros/passados como argumentos/devolvidos como resultados (Exemplo: *Fruta somaFrutas(Fruta f) { ... }*). Os objetos também podem ser atribuídos, com isto um objeto será substituído por outro objeto (diferente de tipos primitivos que copiam valores). Exemplo, *uva = pera*, provocará que as duas variáveis de objetos referenciem o mesmo objeto na memória, ou seja, múltiplas variáveis de objetos podem conter referências para o mesmo objeto. O código abaixo e a Figura 8 exemplifica esta atribuição.

```
1.      public class TestaFruta {
2.          public static void main(String[] args) {
3.              Fruta pera = new Fruta();
4.              Fruta uva = new Fruta();
5.              // pera e uva farão a referência para o mesmo objeto
6.              uva = pera;
7.      }
```

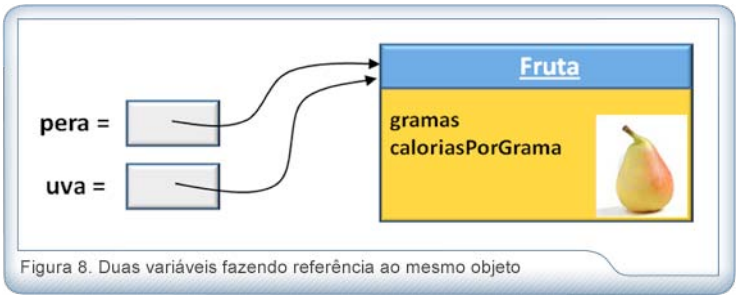


Figura 8. Duas variáveis fazendo referência ao mesmo objeto

Você pode realizar a impressão das identificações dos objetos (IDs), na verdade as variáveis de objeto descrevem o endereços a objetos na memória. O código a seguir mostra como é feita a impressão.

```
1.  public class TestaFruta {
2.      public static void main(String[] args) {
3.          Fruta pera = new Fruta();
4.          System.out.println("Id obj pera = " + pera);
5.          Fruta uva = new Fruta();
6.          System.out.println("Id obj uva = " + uva);
7.          uva = pera;
8.          // pera e uva farão a referência para o mesmo objeto
9.          System.out.println("Id obj uva = " + uva);
10. }
```

Dica:

Caso você deseje imprimir uma string qualquer mais uma variável, você pode na passagem dos parâmetros do método *println* utilizar o operador "+". Exemplo: *System.out.println("Id obj uva = " + uva);*

Construtores

Um construtor especifica como se deve inicializar um objeto. Os construtores são métodos especiais executados uma única vez quando um objeto é criado (*new*). De maneira semelhante aos métodos, os construtores geralmente são declarados como públicos para permitir que qualquer código em um programa construa novos objetos de classe. Diferentemente dos métodos os construtores não tem tipo de retorno.

O nome do construtor é sempre igual ao nome da classe. Quando nenhum construtor é definido, Java define um construtor padrão. Porém, geralmente um construtor fornece valores iniciais (inicialização) para o(s) campo(s). Além disso, pode existir mais de um construtor para cada classe (com diferentes assinaturas), o padrão é possuir um construtor não parametrizado e os demais construtores com diferentes parâmetros (entraremos em detalhes mais na frente).

O código a seguir apresenta o uso de um construtor padrão (não parametrizado).

```
1.    public class Fruta {
2.        private int gramas;
3.        private int caloriasPorGrama;
4.        // Construtor de Fruta
5.        public Fruta(){
6.                gramas = 0;
7.                caloriasPorGrama = 0;
8.        }
9.        // demais metodos
10.   }
```

Construtor sem parâmetros

O código a seguir apresenta o uso de um construtor parametrizado para permitir a inicialização das variáveis de instância (atributos) dos objetos.

```
1.    public class Fruta {
2.        private int gramas;
3.        private int caloriasPorGrama;
4.        // Construtor de Fruta
5.        public Fruta(int g, int c){
6.                gramas = g;
7.                caloriasPorGrama = c;
8.        }
9.        // demais metodos
10.   }
```

Construtor com parâmetros

O uso de um construtor parametrizado pode ser observado no código a seguir. Após a instanciação dos dois objetos, os objetos possuem valores nos seus atributos conforme mostrado na Figura 9.

```
1.    public class TestaFruta {
2.        public static void main(String[] args) {
3.           int valorCalorico;
4.           Fruta pera=new Fruta(150,2);
5.           Fruta laranja=new Fruta(200,3);
6.           valorCalorico=laranja.totalCalorias();
7.           System.out.println("Valor calorico da laranja:"+ valorCalorico);
8.         }
9.    }
```

Utilizando um Construtor

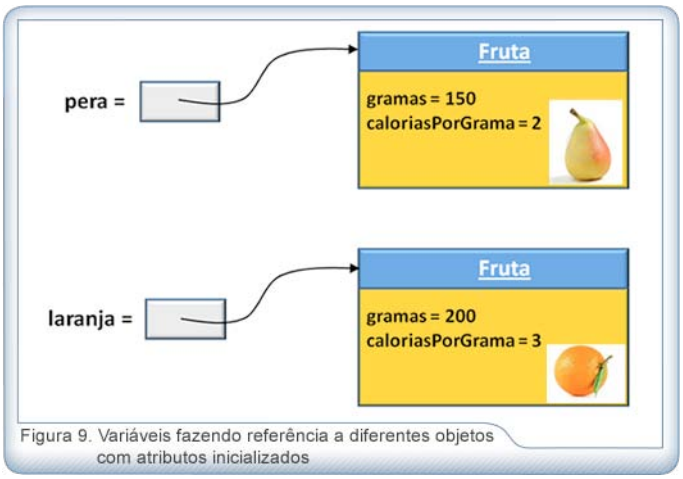


Figura 9. Variáveis fazendo referência a diferentes objetos com atributos inicializados

Palavra reservada static

Variáveis declaradas como *static* são chamadas atributos de classe ou variáveis de classe. Como visto no início da unidade, as variáveis de Classe (ou atributos de classe) são variáveis únicas para todas as instâncias de objetos (independente do número de instâncias da classe). Caso, uma instância alterar valor, todas as outras instâncias irão detectar esta mudança.

As duas maiores utilidades de variáveis estáticas em classes são manter uma informação ou estado para todas as instâncias e armazenar valores que não serão modificados nem serão variáveis de instância de classes, isto é, valores constantes.

O código a seguir mostra o a declaração de uma variável estática onde todas as instâncias da classe Pessoa poderão acessar um atributo estático com o número todas de instâncias criadas (ou número de pessoas).

```
1.    public class Pessoa {
2.        private String nome;
3.        private int idade;
4.        private static int nroPessoas;
5.        Pessoa(String nome, int idade) {
6.              this.nome = nome;
7.              this.idade = idade;
8.              nroPessoas ++;
9.        }
10.       public int getNroInstancias(){
11.             return nroPessoas;
12.       }
13.   }
```

Métodos estáticos em uma classe também são declarados com o modificador *static*. A principal diferença entre métodos estáticos (também conhecidos como métodos de classe) e métodos não estáticos é que os métodos estáticos podem ser chamados sem a criação de instâncias de classe as quais pertencem.

Até este momento, nós utilizamos pelo menos dois métodos estáticos, como o método *printl (System.out.println)* e o método *main* que permite que uma classe seja executada como uma aplicação ou programa (é possível executar o método *main*, sem que seja necessário criar uma instância da classe principal).

Pacotes

Uma aplicação Java, mesmo que simples, envolve a criação de várias classes (um programa consiste de várias classes relacionadas). Claramente, para aplicações e projetos mais complexos, é uma organização das classes de

forma que se saiba a qual aplicação ou projeto uma determinada classe pertence.

Java oferece uma forma de estruturação adicional, que consiste em um mecanismo de agrupamento de classes em pacotes (em inglês, *packages*), com qual podemos criar grupos de classes que possuem relação entre si. Para a criação de pacotes, basta uma declaração em cada classe e uma organização de diretórios.

As principais vantagens são de oferecer um mecanismo de estruturação das classes e de reutilização de software, além de fornecem uma convenção para nomes de classes únicos.

Até agora, em todos os nosso testes na ferramenta NetBeans, nós colocávamos todos os arquivos no pacote *default*, que é um pacote especial (não tem nome). Isso ocorre quando não é incluída qualquer instrução de **package** na parte superior do arquivo-fonte, desta forma as classes serão colocadas no pacote *default*.

A biblioteca Java é organizada em pacotes. Alguns destes são apresentados na Tabela 1.

Pacote	Explicação	Exemplos Classes
java.lang	Pacote defauld para suporte da linguagem	String, Math, ...
java.util	Conjunto de classes utilitárias em geral	Arrays, Date, Random, ...
java.io	Conjunto de classes para entrada e saída	File, Reader, Writer, ...
java.swing	Conjunto de classes para manipulação de interfaces	JFrame, JButoon, ...
java.applet	Conjunto de classes para criação de applets	Applet, ...
java.sql	Conjunto de classes para acesso a banco de dados	ResultSet, Date, DriverManager, ...

Tabela 1: Exemplos de pacotes da biblioteca Java.

Para colocar classes em um pacote

É muito simples colocar classes em um pacote, basta colocar parte superior do arquivo-fonte (classe Java) a seguinte sintaxe:

```
package <nome do pacote>;
```

O código a seguir apresenta a criação da classe Fruta dentro do pacote ifsul.tisad.poo.aula3

```
1.  package ifsul.tsiad.poo.aula3;
2.
3.  public class Fruta {
4.          private int gramas;
5.          private int caloriasPorGrama;
6.          // Construtor de Fruta
7.          public Fruta(int g, int c){
8.                  gramas = g;
9.                  caloriasPorGrama = c;
10.         }
11.         // demais metodos
12. }
```

A localização dos arquivos não acontece apenas pelo o nome do pacote, o próprio arquivo da classe deve ser colocado em um local especial. Os pacotes são mecanismos eficientes para organização de classes e bons para evitar conflito de nomes. Os conflitos de nomes acontecem geralmente em grandes projetos, pois é inevitável

que duas pessoas apresentem o mesmo nome para o mesmo conceito (o mesmo nome para classes construídas por diferentes programadores). Exemplo na API Java: java.util.Date (manipula datas no formato long int) e java.sql.Date (manipula datas no formato SQL).

O Java possui uma recomendação para convenção para nomes de pacotes: se a empresa ou organização tiver domínio na internet, você terá um identificador único e os pacotes devem iniciar com o nome de domínio da Internet em ordem inversa. Exemplo para o caso do IFsul (domínio ifsul.edu.br) o nome do pacote deveria ser: **br.edu.ifsul**.

Para importar pacotes

Para utilizar as classes de um pacote, é necessário no início do código fazer referência ao pacote que será utilizado. Porém não é necessário importar todas as classes do mesmo pacote. Para importar um pacote a sintaxe é a seguinte:

```
import <nome do pacote>;
```

O código a seguir mostra quatro exemplos de importação de pacotes.

```
1.  import ifsul.tsiad.poo.aula3.Fruta; // importa apenas a classe Fruta
2.                                      // do pacote ifsul.tsiad.poo.aula3
3.  import ifsul.tsiad.poo.aula3.*; // importa todas as classes do pacote
4.                                  // ifsul.tsiad.poo.aula3
5.  import java.util.Date; // Importa apenas a classe Date do pacote java.util
6.  import java.util.*; // // importa todas as classes do pacote java.util
```

Para criar um classe e colocar em um pacote no NetBeans é extremamente simples. Quando criar uma nova classe no menu “Arquivo” | “Novo arquivo” e após escolher “Classe Java”, coloque o nome da classe e o nome do pacote (Figura 10).

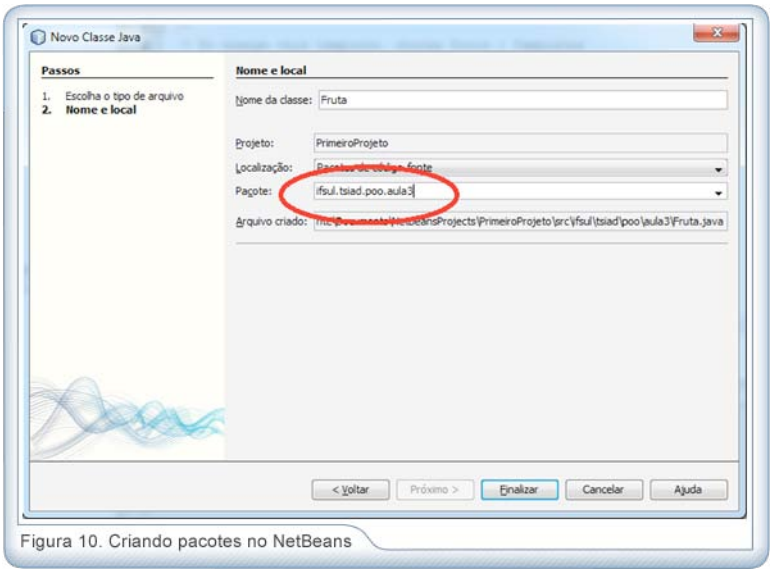


Figura 10. Criando pacotes no NetBeans

Na tela de projetos (Figura 11) você irá notar que dentro do pacote (*ifsul.tisad.poo.aula3*) irá conter a classe *Fruta.java*.

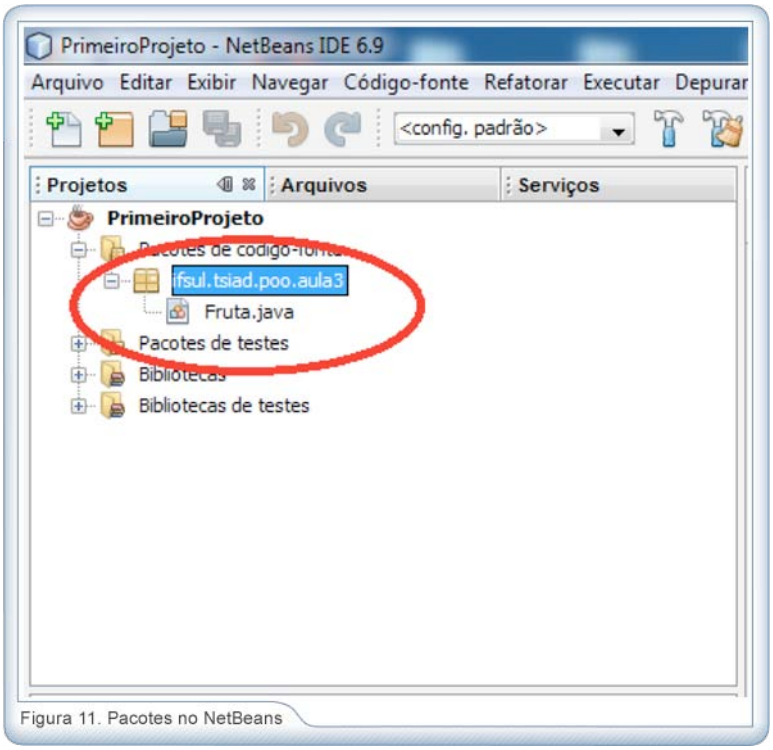


Figura 11. Pacotes no NetBeans

Dicas:

- Nunca é necessário importar explicitamente as classes do pacote java.lang (da API Java). Este pacote que **contém** as classes básicas de Java e a importação é automático.
- Como apresentado anteriormente, campos e métodos declarados sem modificador de acesso são visíveis por todas as classes dentro de um mesmo pacote.

Nesta unidade, foram apresentados os principais conceitos envolvidos na Programação Orientada a Objetos e a programação destes conceitos básicos na linguagem de Programação Java. Na próxima unidade, continuaremos a conhecer a Programação Orientada a Objetos em Java com conceitos mais avançados.

UNIDADE B

CONCEITOS AVANÇADOS DE ORIENTAÇÃO A OBJETOS EM JAVA

Caro (a) aluno (a),

Estamos estudando nesta unidade os principais conceitos envolvidos na programação orientada a objetos em Java. Você já estudou os conceitos básicos de orientação a objetos em Java e agora irá aprender conceitos mais avançados como herança, polimorfismo e sobrecarga de métodos, classes abstratas e interfaces e, por fim relacionamento entre classes.

Herança

Como apresentado anteriormente, o mecanismo de herança é uma das formas de abstração em orientação a objetos onde classes semelhantes são agrupadas em hierarquias. Cada nível de hierarquia pode ser visto como um nível de abstração, onde cada classe em um nível da hierarquia herda as características das classes dos níveis acima.

Este mecanismo facilita o compartilhamento de comportamento comum entre um conjunto semelhante de classes. A herança, também conhecida como generalização/especialização é um relacionamento do tipo classe filha/mãe, subclasse/superclasse ou classe derivada/base. Uma subclasse conterá todos os campos e métodos declarados na superclasse mais os campos e métodos declarados na própria subclasse.

A linguagem Java não permite herança múltipla, ou seja, uma classe somente pode herdar atributos e método de uma única classe. Porém é possível simular herança múltipla através de interfaces (que estudaremos mais na frente). Para utilizar herança no Java é utilizada a palavra reservada *extends.*

A Figura 1 apresenta um exemplo de herança, onde a classe Automóvel é a classe mais genérica (superclasse ou classe mãe) e pode ser especializada em Carro ou Ônibus (subclasses ou classes filha).

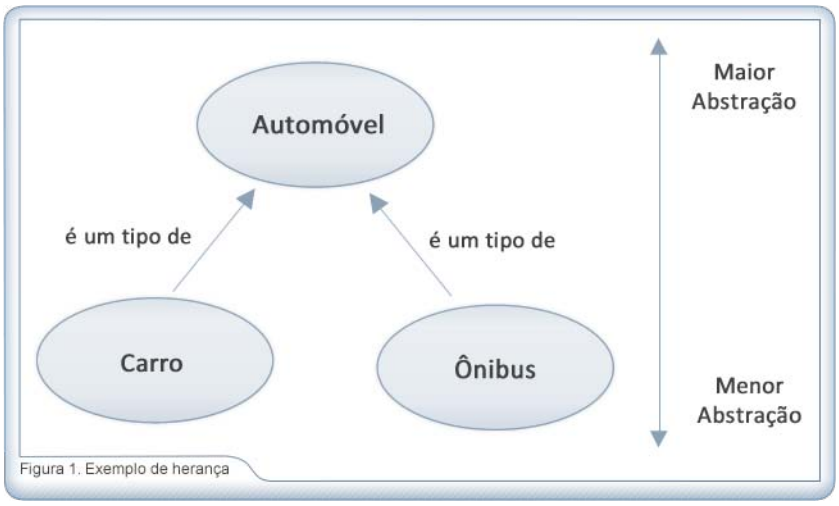


Figura 1. Exemplo de herança

O código a seguir apresenta o uso de herança em Java do exemplo mostrado na Figura 1. A superclasse Automovel define três atributos *protected* do tipo *String* (placa, marca e modelo). Na definição das subclasses *Carro* e *Onibus* é utilizada a palavra *extends* que realiza o mecanismo de herança.

```
1.  public class Automovel {
2.      // Atributos
3.      protected String placa;
4.      protected String marca;
5.      protected String modelo;
6.  }
```

```
1.  public class Carro extends Automovel {
2.      protected int numPortas;
3.      /* Atributos Herdados
4.        protected String placa;
5.        protected String marca;
6.        protected String modelo; */
7.  }
```

```
1.  public class Onibus extends Automovel {
2.      protected int numPassageiros;
3.      protected int numEmbratur;
4.      /* Atributos Herdados
5.         protected String placa;
6.         protected String marca;
7.         protected String modelo; */
8.
9.  }
```

Você deve possuir cuidado com os modificadores de acesso (vistos anteriormente), os modificadores **public** (público) e **protected** (protegido) permitem que os atributos sejam herdados pelas classes filhas, diferentemente do modificador **private** (privado). Contudo, deve-se possuir cuidado no uso do modificador **public** devido a encapsulamento.

No exemplo anterior, as classes filha *Carro* e *Onibus* herdam os atributos da classe mãe *Automovel*. Além dos atributos herdados as classes filhas podem definir outros atributos. Ao definir uma subclasse, você especifica os atributos de instância adicionados, os métodos adicionados e os métodos alterados ou redefinidos (métodos que possuem o mesmo nome de algum método da superclasse).

Deve ficar bem claro no uso de herança o que é herdado ou não. Uma subclasse herda:

- Atributos e métodos públicos e protegidos (protected).

Uma subclasse NÃO herda:

- Atributos e métodos privados;
- Construtores;
- Métodos de mesma assinatura (os métodos são redefinidos);
- Atributos de mesmo nome (o atributo é escondido).

Os métodos redefinidos e atributos escondidos serão detalhados na seção seguinte. No exemplo de código a seguir, a subclasse que herdar da classe *Automovel* irá herdar os atributos protegidos (placa, marca e modelo) e o método público *imprimir*, porém não irá herdar o construtor da classe.

```
1.  public class Automovel extends Object {
2.      // Atributos
3.      protected String placa;
4.      protected String marca;
5.      protected String modelo;
6.
7.      public Automovel(String p, String mar, String mod){
8.          placa = p;
9.          marca = mar;
10.         modelo = mod;
11.     }
12.     public void imprimir(){
13.         System.out.println("Placa: " + placa);
14.         System.out.println("Marca: " + marca);
15.         System.out.println("Modelo: " + modelo);
16.     }
17. }
```

Em Java, cada classe que não estende especificamente outra classe é uma subclasse da classe *Object* (definida na API Java). Cada classe estende a classe *Object* diretamente ou indiretamente (na verdade todas as classes criadas em Java estenderam a classe *Object*) e esta classe possui um número reduzido de métodos comuns a todas as classes.

No exemplo de código acima (linha 1), a classe *Automovel* estende a classe *Object*, porém produziria o mesmo efeito se não fosse colocado explicitamente "*extends Object*", já que o Java sempre irá realizar esta herança de forma automática.

Polimorfismo e Sobrecarga de Métodos

O termo "polimorfismo" origina-se do grego e quer dizer "o que possui várias formas". Como apresentado nos conceitos iniciais de OO, o polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface. A mesma computação funciona para objetos de muitas formas e adpta-se a natureza dos objetos.

O mecanismo de herança permite a criação de classes a partir de outras já existentes. O polimorfismo permite que duas ou mais classes derivadas de uma mesma superclasse possam invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para cada classe derivada, usando para isso uma referência a um objeto do tipo da superclasse. A decisão sobre qual o método que deve ser selecionado é tomada em tempo de execução através do mecanismo de ligação dinâmica que considera o tipo do objeto.

No polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação (assinatura), sendo utilizado o mecanismo de redefinição de métodos (ou sobrescrita de métodos ou *overriding*).

Um outro mecanismo, a sobrecarga (*overloading*) de método permite a existência de vários métodos de mesmo nome, porém com assinaturas levemente diferentes, ou seja, variando o número, o tipo de parâmetros e o retorno. Quando o código é compilado, o compilador determina os métodos a serem executados analisando as listas de argumentos e os retornos esperados, informados nas chamadas.

Alguns autores diferenciam o polimorfismo em dois tipos: estático e dinâmico. O polimorfismo estático é considerado como a sobrecarga de métodos (resolvido em tempo de compilação) e o polimorfismo dinâmico é considerado como sobrescrita de métodos (resolvido em tempo de execução). Porém, para alguns autores, a sobrecarga é normalmente entendida com uma característica da linguagem e não como uma forma de polimorfismo.

Sobrecarga de Construtores e Métodos

Na sobrecarga de construtores e métodos, os construtores e métodos possuem mesmo nome, porém devem possuir a assinatura diferente (parâmetros diferentes em número ou tipo). Normalmente, existe mais de um construtor para cada classe (com diferentes assinaturas), o padrão é possuir um construtor não parametrizado e os demais construtores com diferentes parâmetros.

O código a seguir mostra o exemplo do uso de dois construtores na classe *Pessoa*, um não parametrizado (linha 5) e outro parametrizado (linha 9).

```
1.  public class Pessoa {
2.      protected String nome;
3.      protected char sexo;
4.
5.      public Pessoa(){
6.          nome = "sem nome";
7.          sexo = '?';
8.      }
9.      public Pessoa(String n, char s){
10.         nome = n;
11.         sexo = s;
12.     }
13.     public void imprimir(){
14.         System.out.println("Nome: " + nome);
15.         System.out.println("Sexo: " + sexo);
16.     }
17. }
```

O código a seguir apresenta uma classe de teste que cria objetos a partir da classe *Pessoa* utilizado dois construtores diferentes. A variável de objeto *pessoa1* referencia um objeto *Pessoa* criado a partir do construtor não parametrizado (linha 3) e a variável *pessoa2* a partir do construtor parametrizado (linha 6). A sobrecarga dos construtores será definida em tempo de compilação, onde a definição de qual será o construtor chamado será feita pela quantidade de argumentos da chamada do construtor.

```
1.  public class TestePessoa {
2.      public static void main(String[] args) {
3.          Pessoa pessoa1 = new Pessoa();
4.
5.
6.          Pessoa pessoa2 = new Pessoa("João", 'M');
7.
8.          pessoa1.imprimir();
9.          System.out.println("--------");
10.         pessoa2.imprimir();
11.     }
12. }
```

Utilizando o Construtor não parametrizado

Utilizando o Construtor parametrizado

Após é realizada a impressão a partir do método *imprimir* dos dois objetos criados. A saída no console é apresentada a seguir.

```
Nome: sem nome
Sexo: ?
--------
Nome: João
Sexo: M
```

A sobrecarga também pode ser realizada em métodos. Por exemplo, podemos criar dois métodos com o mesmo nome (*alteraAtributo*) na classe Pessoa porém possuem parâmetros de tipos diferentes. No caso de uma chamada ao método *alteraAtributo*, o compilador determina o método que será executado analisando o argumento passado na chamada do método. Se for do tipo *String* executará o método da linha 1, se o argumento for do tipo char executará o método da linha 4.

```
1.      void alteraAtributo(String n){
2.          nome = n;
3.      }
4.      void alteraAtributo(char s){
5.          sexo = s;
6.      }
```

```
1.  public class Pessoa {
2.      protected String nome;
3.      protected char sexo;
4.
5.      // Código dos construtores
6.
7.      public void imprimir(){
8.          System.out.println("Nome: " + nome);
9.          System.out.println("Sexo: " + sexo);
10.     }
11. }
```

Sobrescrita de Métodos (Redefinição de Métodos)

Quando é utilizado herança, ao definir os métodos de uma subclasse, existem três possibilidades:

- Você pode herdar os métodos da superclasse (são herdados automaticamente);
- Você pode definir novos métodos;
- Você pode sobrescrever métodos da superclasse. É possível especificar um método com a mesma assinatura (ou seja, o mesmo nome e tipos de parâmetro) e este método redefine o método da superclasse.

Neste caso, a classe filha oferece outra implementação para um método herdado com assinaturas idênticas, porém não elimina o acesso ao método da classe ancestral e o interpretador decide dinamicamente (em tempo de execução) qual dos métodos será chamado. A razão de redefinirmos métodos é que métodos de classes herdeiras geralmente executam tarefas adicionais que os métodos das classes ancestrais executam.

O exemplo a seguir apresenta o código da classe *Pessoa*, onde o método imprimir é definido e também apresenta o código da classe *Aluno* onde existe a redefinição do método *imprimir*. A definição de qual método será executado, no caso de realizar a chamado do método, é feita em tempo de execução.

```
1.  public class Aluno extends Pessoa {
2.      private int matricula;
3.
4.      // Código dos construtores
5.
6.      public void imprimir(){
7.          System.out.println("Nome: " + nome);
8.          System.out.println("Sexo: " + sexo);
9.          System.out.println("Matricula: " + matricula);
10.     }
11. }
```

```
1.  public class Aluno extends Pessoa {
2.      private int matricula;
3.
4.      // Código dos construtores
5.
6.      public void imprimir(){
7.          System.out.println("Nome: " + nome);
8.          System.out.println("Sexo: " + sexo);
9.          System.out.println("Matricula: " + matricula);
10.     }
11. }
```

Ocultação de atributos de instância

No uso de herança, no momento de definir os atributos de instância você possui duas alternativas:

- Você pode herdar os atributos da superclasse (todos os atributos são herdados automaticamente);
- Você pode definir novos atributos, quaisquer novos atributos definidos na subclasse estão presentes apenas nos objetos da subclasse.

Caso você defina um novo atributo com o mesmo nome de um atributo da superclasse, então existirão dois atributos de instância com o mesmo nome. O atributo da subclasse oculta (esconde ou sombreia) o atributo da superclasse. Ao contrário da sobrescrita de métodos a ocultação não oferece grandes vantagens e gera confusão.

Caso um atributo é declarado em uma superclasse e oculto em uma subclasse, e métodos que acessam este campo são herdados, estes métodos farão referência ao atributo da classe onde foram declarados.

As Palavras-chave “this” e “super”

A palavra-chave *this* faz uma referência à própria instância (objeto atual). Observe o código a seguir onde o construtor da classe utiliza como parâmetros os mesmos nomes dos atributos, neste caso se não houvesse a palavra *this* haveria confusão se uma determinada variável é um atributo da instância ou um parâmetro. A palavra-chave *this* oferece um meio de referenciar o atributo em vez do parâmetro com o mesmo nome.

```
1.  public class Pessoa {
2.      protected String nome;
3.      protected char sexo;
4.
5.      public Pessoa(){
6.          nome = "sem nome";
7.          sexo = '?';
8.      }
9.      public Pessoa(String nome, char sexo){
10.         this.nome = nome;
11.         this.sexo = sexo;
12.     }
13.     // demais métodos
14. }
```

Poderíamos contornar este problema, utilizando nomes diferentes para os atributos e parâmetros, porém isso beneficia a legibilidade do código (utilizando o mesmo nome dos parâmetros do que utilizados para os atributos).

A palavra-chave *this* também pode ser utilizada para realizar o reaproveitamento de código dentro da própria classe. O código a seguir mostra o construtor não parametrizado da classe Pessoa (linha 5) chamando o construtor parametrizado definido na linha 10 através do *this*. Isso elimina a necessidade das linhas de código 7 e 8.

```
1.  public class Pessoa {
2.      protected String nome;
3.      protected char sexo;
4.
5.      public Pessoa(){
6.          this("sem nome", '?');
7.          //nome = "sem nome";
8.          //sexo = '?';
9.      }
10.     public Pessoa(String nome, char sexo){
11.         this.nome = nome;
12.         this.sexo = sexo;
13.     }
14.     // demais métodos
15. }
```

Já a palavra-chave *super* permite o acesso aos métodos da superclasse. O acesso a métodos da classe ancestral é útil para aumentar a reutilização de código: se existem métodos na classe ancestral que podem efetuar parte do processamento, devemos utilizar este código já existente.

Os códigos apresentados a seguir mostram a superclasse Pessoa (que possui o método imprimir) e a subclasse *Aluno* que no método imprimir invoca o método imprimir da superclasse *Pessoa* (através da palavra-chave *super*) reutilizando o código de impressão dos atributos *nome* e *sexo*.

```
1.  public class Pessoa {
2.      protected String nome;
3.      protected char sexo;
4.
5.      // Código dos construtores
6.
7.      public void imprimir(){
8.          System.out.println("Nome: " + nome);
9.          System.out.println("Sexo: " + sexo);
10.     }
11. }
```

```
1.  public class Aluno extends Pessoa {
2.      private int matricula;
3.
4.      // Código dos construtores
5.
6.      public void imprimir(){
7.          super.imprimir();
8.          System.out.println("Matricula: " + matricula);
9.      }
10. }
```

Algumas observações devem ser cuidadas no uso da palavra-chave *super*:

- Métodos da superclasse podem ser chamados pela palavra-chave *super* seguida de um ponto e do nome do método;
- Somente métodos da superclasse imediata podem ser chamados usando a palavra-chave *super* (não existe *super.super);*
- Se um método de uma classe ancestral for herdado pela classe descendente, ele pode ser chamado diretamente sem a necessidade da palavra-chave *super*.

Como já sabemos, na herança, os construtores não são herdados. Porém a palavra-chave *super* também pode ser utilizada para chamar o construtor da superclasse. Devem ser observadas algumas condições:

- Construtores somente podem ser chamados dentro de construtores da subclasse;
- Mesmo assim, somente se for declarado na primeira linha de código do construtor da subclasse;
- Somente construtores da superclasse imediata podem ser chamados usando a palavra-chave *super.*

O código a seguir mostra a chamada do construtor da classe ancestral *Pessoa* dentro do construtor da classe *Aluno* através da palavra-chave *super* (linha 7).

```
1.  public class Aluno extends Pessoa {
2.      private int matricula;
3.
4.      // Código dos construtores
5.
6.      public Aluno(String nome, char sexo, int matricula){
7.          super(nome, sexo);
8.          this.matricula = matricula;
9.      }
10.
11.     // Demais métodos
12.
13. }
```

Palavra reservada final

A palavra reservada, ou modificador, *final* faz com que os valores de atributos, depois de declarados, não possam mais ser modificados, o que é desejável para campos representados por constantes.

Muitas vezes podemos declarar constantes como *final* e *static* (para que sejam o mesmo valor independente da quantidade de instâncias que venham a ser criadas). O exemplo a seguir mostra a declaração de um atributo (NUM_MAXIMO_DISCIPLINAS_SEMESTRE) como *final*. Java convenciona que o nome das constantes (declaradas com *final*) são todos em letra maiúscula.

```
1.  public class Aluno extends Pessoa {
2.      private int matricula;
3.
4.      final static int NUM_MAXIMO_DISCIPLINAS_SEMESTRE = 5;
5.
6.      // Código dos construtores
7.
8.      // Demais métodos
9.
10. }
```

Métodos também podem ser declarados como *final*. Estes métodos são herdados por subclasses, porém não podem ser sobrescritos (redefinidos) a não ser que a assinatura seja diferente. Classes inteiras também podem ser declaradas como *final*, neste caso, todos os métodos da classe serão finais, mas não os seus atributos. A declaração de uma classe como *final* efetivamente impede o mecanismo de herança (o compilador não compilará uma classe declarada como herdeira de uma classe *final*).

Arrays de Instâncias

Arrays de instâncias de classes em Java podem ser declaradas de forma similar a arrays de valores de tipos nativos. A principal diferença é que a inicialização do array deve ser seguida da inicialização dos elementos do array, que deve ser realizada da mesma forma que instâncias de classe (através da palavra-chave *new* e da chamada do construtor).

Uma diferença entre arrays de tipos nativos e instâncias de classe é que array nativos devem conter o mesmo tipo de dados, enquanto que arrays de instância podem conter instâncias de qualquer classe que seja derivada da classe usada para declaração do array. Por exemplo, podemos declarar o array como instâncias da classe *Automovel* e inicializar com objetos da classe *Carro* e *Onibus* (herdeiras da classe Pessoa).

Os códigos das classes apresentados a seguir mostram a classe *Carro* e a classe de teste que instância um conjunto de três objetos *Carro* e armazenam as suas referências em um vetor de instâncias do tipo *Carro* chamado *vetorCarros*.

```
1.  public class Carro {
2.      private String modelo;
3.      private String placa;
4.
5.      public Carro(String modelo, String placa){
6.          this.modelo = modelo;
7.          this.placa = placa;
8.      }
9.      public void mostrarCarro(){
10.         System.out.println(this.modelo + " " + this.placa);
11.     }
12. }
```

O código da classe TesteCarros mostra o seguinte:

- Linha 3: cria o array *vetorCarros* para armazenar instâncias da classe *Carro* e possui um tamanho fixo de 3 elementos;
- Linhas 4 a 6: acontece a criação dos três objetos do tipo *Carro* e suas referências são armazenadas nas posições 0, 1 e 2 do array;
- Linhas 7 a 9: é realizada a impressão das informações de todos os carros através da chamada do método *mostrarCarro.*

```
1.  public class TesteCarros {
2.      public static void main(String[] args) {
3.          Carro vetorCarros[] = new Carro[3];
4.          vetorCarros[0] = new Carro("Brasilia", "III8564");
5.          vetorCarros[1] = new Carro("Kombi", "JJJ7675");
6.          vetorCarros[2] = new Carro("Chevette", "KKK5643");
7.          for(int i=0; i<vetorCarros.length; i++){
8.             vetorCarros[i].mostrarCarro();
9.          }
10.     }
11. }
```

O grande problema deste tipo de arrays (veremos outros tipos de estruturas de dados mais na adiante) é que possui um tamanho pré-determinado. O interessante é utilizar uma estrutura dinâmica, onde conforme for necessário o tamanho da estrutura possa ser aumentada em tempo de execução.

Classes Abstratas e Interfaces

A linguagem Java possui dois mecanismos que permitem a criação de classes somente com o objetivo de conter descrições de atributos e métodos que devem implementar, porém sem efetivamente implementar esses métodos. Estes tipos de mecanismos são interessantes para criar hierarquia de classes. Os dois mecanismos são: classes abstratas e interfaces.

Classes Abstratas

As classes abstratas representam um conceito abstrato e não devem ser instanciadas (gerar objetos), servem apenas para permitir a derivação de novas classes. Identificamos uma classe como abstrata pela palavra reservada *abstract.*

A declaração de uma classe abstrata possui a seguinte forma:

```
public abstract class nomeClasse { ... }
```

Apesar destas classes serem criadas apenas para serem estendidas elas possuem:

- Atributos: que são herdados pelas classes filhas;
- Construtores: que podem ser chamados através da palavra-chave super pelas classes filhas;
- Métodos: que podem ser divididos em dois tipos, métodos concretos e métodos abstratos.

Os atributos não podem ser declarados como *abstract* e são herdados pelas classes filhas desde que não sejam private. Da mesma forma, o construtor não pode ser declarado como *abstract* e pode ser chamado na classe filha pelo uso do super.

Os métodos concretos são métodos que não são declarados como abstratos e são herdados pelas classes descendentes. Já os métodos abstratos são identificados com a palavra-chave *abstract* e não possuem implementação (definem apenas a interface), porém **devem ser obrigatoriamente implementados na classe filha.**

Uma classe que possua um método abstrato deve ser obrigatoriamente declarada como *abstract*, porém pode existir uma classe abstrata que não possua nenhum método abstrato.

O uso de classes abstratas permite a padronização no comportamento das classes filhas tornando a superclasse apenas uma guia de que atributos e métodos devem ser implementados nas subclasses. Além disso, em vários casos este tipos de classes representam um conceito abstrato e não existe a razão de criar instâncias de classe a partir dela.

O código a seguir mostra o código de uma classe abstrata *Funcionario*. Em um sistema de folha de pagamentos de uma empresa, por exemplo, não existiria uma representação de um "Funcionário" e sim, de um operário, diretor, supervisor, etc. Neste caso, a classe *Funcionario* seria apenas a superclasse que seria utilizada pelas classes *Operario, Diretor e Supervisor.* Os métodos abstratos *exibirDados* e *calcularPagamento* seriam definidos nas classes filhas, a classe abstrata apenas os padronizaria.

```
1.  public abstract class Funcionario {
2.      protected String nome;
3.      protected String cpf;
4.      protected double salario;
5.
6.      public Funcionario(){}
7.
8.      public Funcionario(String nome, String cpf, double salario){
9.          this.nome = nome;
10.         this.cpf = cpf;
11.         this.salario = salario;
12.     }
13.
14.     public void setNome(String nome){
15.         this.nome = nome;
16.     }
17.     public String getNome(String nome){
18.         return this.nome;
19.     }
20.
21.     public abstract void exibirDados();
22.     public abstract void calcularPagamento();
23. }
```

Atributos

Construtores

Métodos Concretos

Métodos Abstratos

Um outro exemplo de uso de classe abstrata é a classe Pessoa. Em um sistema de registros acadêmicos, a classe Pessoa seria a superclasse e as subclasses poderiam ser as classes *Aluno* e *Professor.*

Interfaces

As classes abstratas podem conter métodos não abstratos que serão herdados e poderão ser utilizados pelas classes filhas. Se uma classe não possuir nenhum método concreto poderemso criá-la como uma interface.

Interfaces são classes abstratas "puras" que fornecer uma especificação para uma classe sem definir como será implementada (fica a cargo das classes que utilizam a interface) e assim como a classe abstrata não pode ser instanciada.

Diferença entre classes abstratas e interfaces é que uma classe herdeira pode herdar de apenas uma única classe (abstrata ou não) enquanto qualquer classe pode implementar várias interfaces simultaneamente. Isto permite um mecanismo simplificado de implementação de herança múltipla em Java.

As interfaces possuem as seguintes características:

- São declaradas através da palavra-chave *interface* e o modificador de acesso *public*;
- Não existem construtores;
- Não tem variáveis de instância (a não ser constantes estáticas);
- Todos os métodos são abstratos e públicos (a classe que implementa a interface deve implementar todos os métodos definidos na interface).

As interfaces podem ser úteis para implementação de bibliotecas de constantes (pois todos os campos devem ser declarados como *static* e *final*).

A Figura 2 apresenta um bom exemplo de aplicação de interfaces. A Figura Geométrica apenas define métodos que deverão ser implementados nas classes filhas.

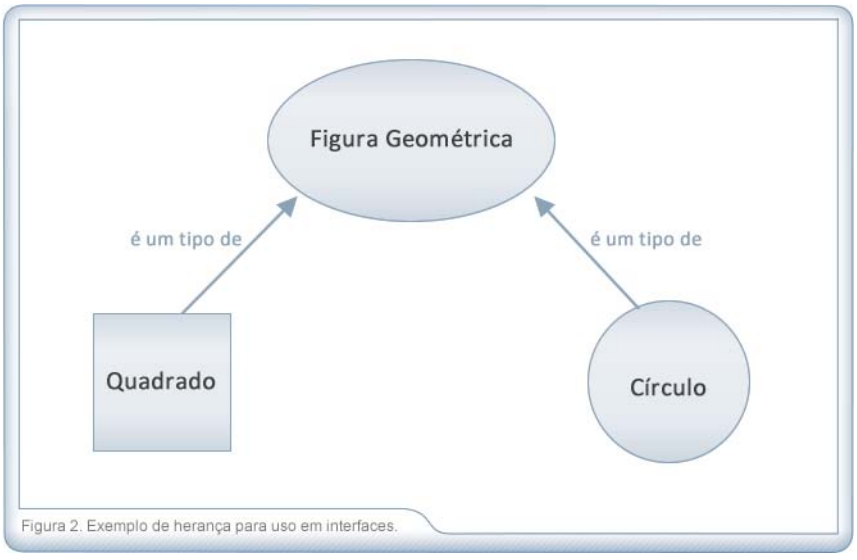


Figura 2. Exemplo de herança para uso em interfaces.

O código a seguir mostra a interface *FiguraGeometrica* que será implementada nas subclasses *Circulo* e *Quadrado.* A relação de herança entre uma interface e uma classe é feita declarando-se a classe como implementando a interface, utilizando a palavra-chave implements. As classes *Circulo* e *Quadrado* devem obrigatoriamente implementar os métodos definidos na interface.

```
1.  public interface FiguraGeometrica {
2.
3.      public double calculaArea();
4.
5.      public double calculaPerimetro();
6.
7.  }
```

Interface

Métodos que devem ser obrigatoriamente implementados nas subclasses

```
1.  public class Circulo implements FiguraGeometrica {
2.      private double raio;
3.
4.      public Circulo(double raio) {
5.          this.raio = raio;
6.      }
7.      public double calculaArea(){
8.          return Math.PI * raio * raio;
9.      }
10.     public double calculaPerimetro(){
11.         return 2.0 * Math.PI * raio;
12.     }
13. }
```

Classe que implementa a Interface

```
1.  public class Quadrado implements FiguraGeometrica {
2.      private double lado;
3.
4.      public Quadrado(double lado) {
5.          this.lado = lado;
6.      }
7.      public double calculaArea(){
8.          return lado * lado;
9.      }
10.     public double calculaPerimetro(){
11.         return lado * 4;
12.     }
13. }
```

Classe que implementa a Interface

Você pode observar na classe Circulo que para obter o valor do π foi utilizada a classe Math da API Java (entraremos em detalhe na próxima unidade).

Relacionamento entre classes

No desenvolvimento de um software orientado a objetos, existe um número muito pequeno de classes que trabalham sozinhas e, na maioria das vezes, muitas classes necessitam de outras para fazer o seu trabalho (colaboram umas com as outras de várias maneiras).

Na modelagem de um sistema, o projetista necessita identificar os itens que formam o vocabulário do sistema (para identificar as classes) e também modelar o relacionamento entre os itens (relacionamento entre as classes). Este relacionamento é uma conexão entre itens.

Para visualizar os relacionamentos entre classes, os programadores utilizam diagramas de classe da notação UML (Unified Modeling Language) que será estudada na disciplina de "Análise e Projeto de Sistemas de Informação Orientados a Objetos". Contudo, nesta seção os exemplos já serão apresentados, utilizando estes diagramas.

Podemos afirmar que uma classe depende de outra, se utiliza objetos dessa classe. Se muitas classes de um programa dependem uma da outra, dizemos que o **acoplamento** entre classes é alto. Caso contrário, se houver poucas dependências entre classes, dizemos que o acoplamento é baixo. Uma boa prática é minimizar o acoplamento (isto é, dependência) entre classes, pois isso facilita vários aspectos no projeto do software como a manutenção.

A seguir são apresentados os principais relacionamentos em orientação a objetos e exemplos de uso. Os relacionamentos apresentados são os seguintes: generalização, associação, agregação e composição.

Generalização

O relacionamento de generalização conecta classes generalizadas com outras mais especializadas e já foi apresentado como herança. A generalização é um relacionamento de itens gerais (superclasses) e itens mais específicos (subclasses) e é considerado um como "**é um tipo de**".

No exemplo da Figura 3, deve-se considerar que *Carro* **é um tipo de** *Automovel*, como *Onibus* também **é um tipo de Automovel**. Na notação UML a generalização é representada graficamente como linhas sólidas apontando para a classe mãe. No exemplo, as setas apontam para a classe *Automovel* (superclasse ou classe mãe).

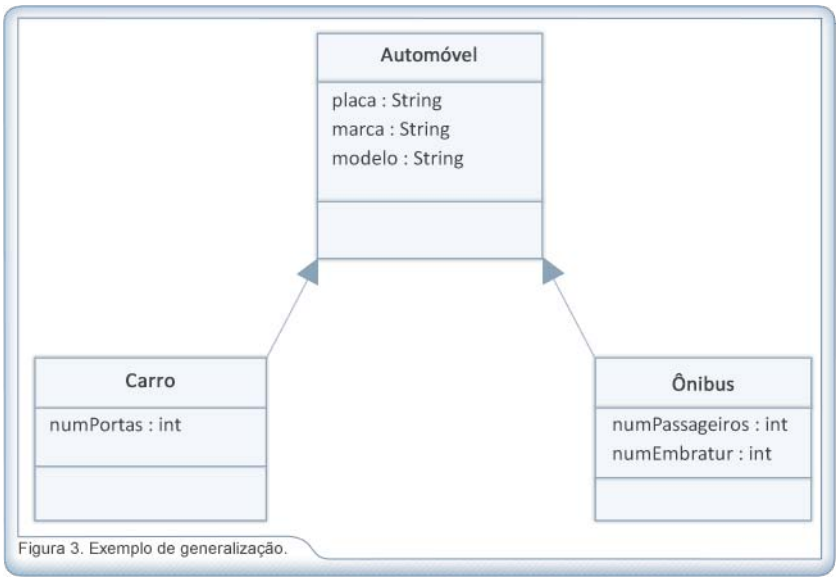


Figura 3. Exemplo de generalização.

Como já apresentado anteriormente, a herança (ou o relacionamento de generalização) é expresso em um código Java através da palavra reservada *extends*. O exemplo de código a seguir apresenta a definição das classes *Carro* e *Onibus* herdam da classe *Automovel.*

```
1.  public class Carro extends Automovel {
2.
3.  }
```

```
1.  public class Onibus extends Automovel {
2.
3.  }
```

Associação

A associação é um relacionamento estrutural entre instâncias (especifica objetos de um item conectados a objetos de outro item). Como por exemplo, salas são formadas por paredes e outros itens; paredes podem ter portas e janelas embutidas, etc.

Uma pura associação entre duas classes representa um relacionamento estrutural entre pares onde as classes estão em um mesmo nível e uma não é mais importante que outras. Também é possível a partir de uma associação conectando duas classes, navegar do objeto de uma classe até o objeto de outra classe e vice-versa. Em associação um objeto **"usa um"** outro objeto.

No exemplo da Figura 4, deve-se considerar que *Carro* possui um *Motorista*. Na notação UML a associação é representada graficamente por uma linha sólida conectando as duas classes. Em uma associação simples sem adornos – linha sem seta – é possível navegar do objeto de uma classe até o objeto de outra classe e vice-versa.

Figura 4. Exemplo de associação sem adornos.

O código a seguir apresenta como o relacionamento deve ser realizado em uma associação simples (sem adornos). A novidade aqui é que a classe *Automovel* possui um atributo que é a referência para outro objeto do tipo *Motorista* (acessado pela variável de objeto pessoa). Não confunda *Motorista* (maiúscula) com *motorista* (minúscula). Normalmente os programadores Java utilizam o mesmo nome da Classe (neste caso, *Motorista*), porém em minúscula para a variável de objeto. No exemplo a seguir, pode-se navegar (acessar) a partir do objeto da classe *Automovel* um objeto da classe *Motorista* e vice-versa.

```
1.  public class Automovel {
2.      // Atributos
3.      protected Motorista motorista;
4.  }
```

```
1.  public class Motorista {
2.      // Atributos
3.      protected Automovel automovel;
4.  }
```

Associação pode ser simples com adornos (apontando a direção a ser seguida). No exemplo da Figura 5, a partir de um objeto da classe *Automovel* será possível de encontrar objetos da classe *Motorista* correspondente, porém considerando *Motorista* não será possível acessar *Automovel*. Observe que o tipo da ponta da seta é diferente da generalização).

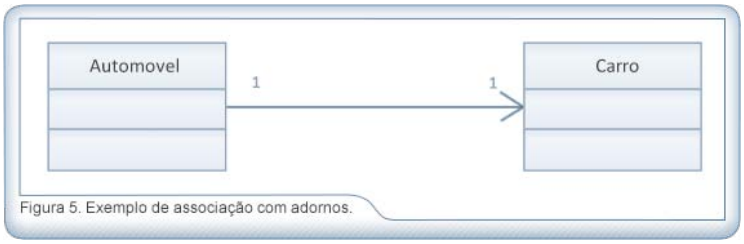


Figura 5. Exemplo de associação com adornos.

O código a seguir mostra o exemplo da Figura 5 que utiliza uma associação simples com adornos.

```
1.  public class Automovel {
2.      // Atributos
3.      protected Motorista motorista;
4.  }
```

```
1.  public class Motorista {
2.      // Atributos
3.  }
```

Agregação

Em uma pura associação entre duas classes, estas classes estão em um mesmo nível e uma não é mais importante que outras. Contudo, em certas modelagens é necessário representar uma classe que representa um item maior (o "todo") que é formada por itens menores (as "partes"). Na agregação um objeto **é composto por** (**ou é parte de**) outros objetos. Em outras palavras uma agregação é uma associação mais específica.

A Figura 6 mostra uma agregação, onde a Empresa (todo) **é composta por** Departamentos (parte). Você pode observar que o que a linha possui um adorno diferente em um dos lados (um losango).

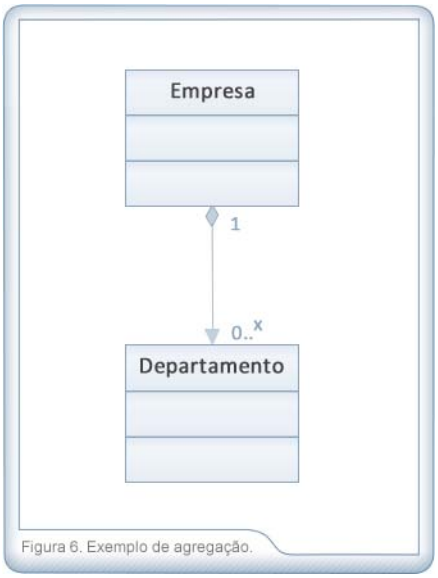


Figura 6. Exemplo de agregação.

O código a seguir apresenta um exemplo de agregação onde uma empresa pode possui um único departamento, pois um objeto da classe *Empresa* poderá referenciar apenas um único departamento (através da variável de objeto depto).

```
1.  public class Empresa {
2.      // Atributos
3.      protected Departamento depto;
4.  }
```

```
1.  public class Departamento {
2.
3.  }
```

Para solucionar este problema, de referenciar um único departamento, podemos declarar a variável depto como um array de instância de classe. Assim, um objeto da classe *Empresa* poderá referenciar vários objetos do tipo *Departamento*. O código a seguir mostra esta declaração.

```
1.  public class Empresa {
2.      // Atributos
3.      protected Departamento depto[];
4.  }
```

```
1.  public class Departamento {
2.
3.  }
```

Composição

A composição é uma forma de agregação, uma especialização da agregação, com propriedade bem definida e tempo coincidente como parte do todo. Na agregação, se a instância do **todo** for removida, suas **partes** não são necessariamente removidas. Já na composição, se a instância do **todo** for removida, suas **partes** também deverão ser removidas.

Estes dois relacionamentos (agregação e composição) definem um relacionamento **"todo/parte".** Porém a agregação define uma "dependência fraca" entre o todo e suas partes e a composição define uma "dependência forte" entre o todo e suas partes.

A Figura 7 apresenta o diagrama de classes UML da composição. Pode-se notar que a losango (diferentemente da agregação) é preenchido. Neste exemplo, a Empresa (todo) é composta por Departamentos (parte). Na composição, se o **todo** for removido, as suas **partes** também deixam de existir. Já na agregação, se o todo deixa de existir, as suas partes podem deixar de existir (mas não é obrigatório).

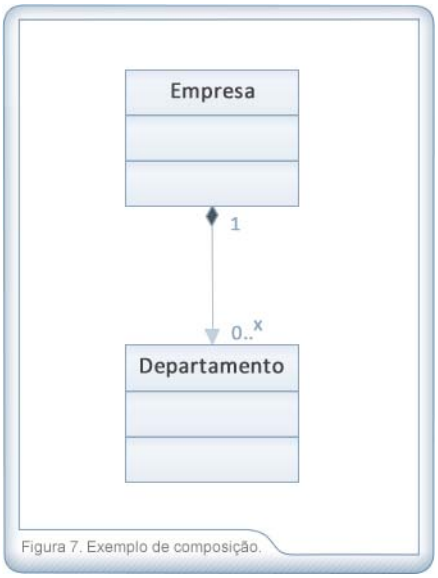


Figura 7. Exemplo de composição.

A forma de declarar os atributos na composição é a mesma da agregação (não existem mudança alguma), o que muda é a forma que o programador irá implementar caso um objeto *Empresa* seja destruído no código.

Resumindo, podemos utilizar os seguintes relacionamento entre classe:

- Generalização: **"é um tipo de"** (as propriedades são herdadas de outra classe);
- Associação: **"usa"** (um objeto envia uma mensagem para outro objeto);
- Agregação: **"composto por"** (um objeto é composto por outros objetos)
- Composição: **"composto por"** ou **"é parte essencial de"** (um objeto é composto por outros objetos).

Conclusão

Nesta unidade, foram apresentados conceitos avançados de Java como herança, polimorfismo e sobrecarga de métodos, classe abstratas e interfaces e por fim relacionamento entre classes. Na próxima unidade, estudaremos um pouco da biblio

UNIDADE C

USO DA BIBLIOTECA JAVA

A unidade anterior apresentou os conceitos fundamentais de Orientação a Objetos e Java. Nesta unidade, estudaremos alguns dos pacotes, classes e interfaces disponibilizadas pela biblioteca Java.

Introdução

A linguagem Java é distribuída com um vasto conjunto de bibliotecas (ou APIs - *Application Programming Interface*). Isto se deve a fato da plataforma Java não ser dependente de qualquer sistema operacional. Desta forma, as aplicações não podem depender das bibliotecas desses sistemas. Para solucionar esse problema, a plataforma Java disponibiliza um grande conjunto padronizado de bibliotecas, que contém um número enorme de funções.

A documentação da API Java pode ser visualizada diretamente no site:

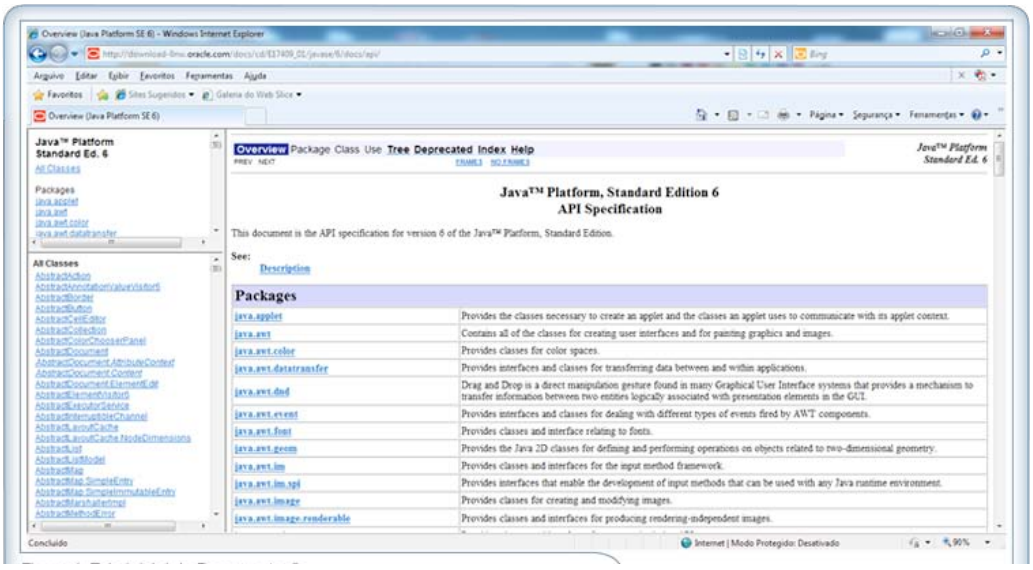
(http://java.sun.com/javase/6/docs/api/)

ou pode ser realizado o download da documentação através do site

http://java.sun.com/javase/downloads/index.jsp#docs.

Uma das coisas mais importantes é saber procurar na documentação, pois um bom programador Java deve conhecer muito bem a API Java. Na Figura 1 é mostrada a tela com o arquivo de início da API Java:

- O frame no canto superior esquerdo lista os pacotes Java disponíveis;
- O frame no canto inferior esquerdo lista as classes do pacote escolhido no frame anterior;
- O frame grande que ocupa o restante da página fornece a documentação da classe que foi escolhida.



A API Java é composta por vários pacotes (alguns listados na Tabela 1). Nesta unidade, estudaremos algumas classes e interfaces de alguns pacotes da API Java.

Pacote	Descrição
java.lang	Funcionalidades básicas da linguagem e tipos fundamentais
java.util	Classes de coleções de estrutura de dados
java.io	Operações com arquivos
java.math	Funções aritméticas
java.nio	Novo framework de I/O para o Java
java.net	Operações de rede, sockets, DNS, etc.
java.security	Geração de chaves e criptografia
java.sql	JDBC (Java Database Connectivity) para acesso a banco de dados
javax.swing	Hierarquia de pacotes independente de plataforma para componentes gráficos
java.applet	Classes para criação e implementação de applets

Tabela 1: Alguns pacotes da API Java.

Classe *Object*

A classe *Object* pertence ao pacote java.lang. Em Java, cada classe que não estende outra classe automaticamente estende a classe *Object*. Ou seja, toda a classe é subclasse (direta ou indiretamente) da classe *Object*, em outras palavras, a classe *Object* é a superclasse de todas as classes do Java.

Os métodos da classe *Object* são métodos mais gerais que podem ser aplicados a todas as classes. A Tabela 2 apresenta alguns destes métodos.

Método	Descrição
Object **clone**()	Cria e retorna uma cópia do objeto.
boolean **equals**(Object obj)	Indica de um objeto é igual a este objeto.
String **toString**()	Retorna uma representação String do objeto.

Tabela 2: Métodos da classe Object.

Não é necessário importar o pacote *java.lang* pois ele é automaticamente importado. O método *clone* pode ser utilizado caso para cópia de objetos, já o método *equals* serve para testar se dois objetos são iguais. Como o método *toString*, o método *equals* pode ser redefinido, ou seja, a sua classe pode redefinir o método (com mesma assinatura) porém com uma implementação diferente.

O código a seguir mostra a impressão da variável objeto *carro*. Neste caso, o método *toString* herdado da classe *Object* pela classe *Carro* é invocado e é impresso algo que representa o ID da classe (Carro@3ae48e1b).

```
1.  package ifsul.tsiad.poo.api;
2.
3.  public class Carro {
4.      private String modelo;
5.      private String placa;
6.      public Carro(){
7.          this.modelo = "Sem modelo";
8.          this.placa = "Sem placa";
9.      }
10.     public Carro(String modelo, String placa){
11.         this.modelo = modelo;
12.         this.placa = placa;
13.     }
14. }
```

```
1.  package ifsul.tsiad.poo.api;
2.
3.  public class TesteCarro {
4.      public static void main(String[] args) {
5.          Carro carro = new Carro("Fusca", "IIJ4949");
6.          System.out.println(carro);
7.      }
8.  }
```

Para resolver o problema da impressão, você pode redefinir o método *toString* na classe *Carro* (apenas alterar a classe Carro incluindo a redefinição do método). O código a seguir mostra esta redefinição nas linhas 14 a 16.

```
1.  package ifsul.tsiad.poo.api;
2.
3.  public class Carro {
4.      private String modelo;
5.      private String placa;
6.      public Carro(){
7.          this.modelo = "Sem modelo";
8.          this.placa = "Sem placa";
9.      }
10.     public Carro(String modelo, String placa){
11.         this.modelo = modelo;
12.         this.placa = placa;
13.     }
14.     public String toString(){
15.         return (this.modelo + " " + this.placa);
16.     }
17. }
```

```
1.  package ifsul.tsiad.poo.api;
2.
3.  public class TesteCarro {
4.      public static void main(String[] args) {
5.          Carro carro = new Carro("Fusca", "IIJ4949");
6.          System.out.println(carro);
7.      }
8.  }
```

Dica:

A documentação sobre a classe Object está no link:

http://java.sun.com/j2se/1.6.0/docs/api/java/lang/Object.html

Classe String

A classe *String* também é do pacote *java.lang*. A classe *String* representa uma sequência de caracteres, sendo que as strings são objetos da classe *String*. Você pode notar que as strings são objetos (são declarados com maiúsculo) ao contrário dos tipos básicos *int* e *double*, por exemplo.

As strings podem ser concatenadas com o operador "+". Já utilizados isto durante as impressões na tela. Contudo, a classe *String* possui um conjunto enorme de métodos, alguns desses são apresentados na Tabela 3.

Método	Descrição
char **charAt**(int index)	Retorna um valor char de uma posição específica.
int **compareTo**(String anotherString)	Compara duas strings lexicograficamente.
int **compareToIgnoreCase**(String str)	Compara duas strings lexicograficamente, ignorando as maiúsculas e minúsculas.
boolean **equalsIgnoreCase**(String anotherString)	Compara esta String com uma outra String, ignorando as maiúsculas e minúsculas.
int **lenght**()	Retorna o tamanho da string.
String **toLowerCase**()	Converte todos os caracteres da string para minúsculas.
String **toUpperCase**()	Converte todos os caracteres da string para maiúsculas.

Tabela 3: Métodos da classe String.

O código a seguir mostra o uso dos métodos *charAt, toLowerCase, toUpperCase* e *equals*. A impressão do programa na tela será:

```
t

abacate

ABACATE

é igual !
```

```
1.  package ifsul.tsiad.poo.api;
2.
3.  public class TesteStrings {
4.      public static void main(String[] args) {
5.          String s1 = "Abacate";
6.          System.out.println(s1.charAt(5));
7.          System.out.println(s1.toLowerCase());
8.          System.out.println(s1.toUpperCase());
9.          if(s1.equals("Abacate")){
10.             System.out.println("é igual !");
11.         } else {
12.             System.out.println("é diferente !");
13.         }
14.     }
15. }
```

Dica:

A documentação sobre a classe String está no link:

http://java.sun.com/j2se/1.6.0/docs/api/java/lang/String.html

Classe Math

A classe *Math* também é do pacote *java.lang*. A classe *Math* contém métodos para realizar operações matemáticas básicas como exponencial, logaritmo, raiz quadrada e funções trigonométricas. A Tabela 4 apresenta alguns dos métodos da classe *Math*.

Método	Descrição
static double **abs**(double a)	Retorna o valor absoluto de um valor *double*.
static float **abs**(float a)	Retorna o valor absoluto de um valor *float*.
static int **abs**(int a)	Retorna o valor absoluto de um valor *int*.
static long **abs**(long a)	Retorna o valor absoluto de um valor *long*.
static double **acos**(double a)	Retorna o arco cosseno de um valor, o valor retornado está entre 0.0 e PI.
static double **cbrt**(double a)	Retorna a raiz ao cubo de um valor *double*.
static double **cos**(double a)	Retorna o cosseno trigonométrico de um ângulo.
static double **exp**(double a)	Retorna o número de Euler e elevado a potência de um valor *double*.
static double **log**(double a)	Retorna o logaritmo natural (base e) de um valor *double.*
static double **log10**(double a)	Retorna o logaritmo na base 10 de um valor *double*.
static double **max**(double a, double b)	Retorna o maior valor de dois valores *double*.
static double **min**(double a, double b)	Retorna o menor valor de dois valores *double*.
static double **pow**(double a, double b)	Retorna o valor do primeiro argumento elevado a potência do segundo argumento.
static double **random()**	Retorna um valor *double* de sinal positivo, maior ou igual a 0.0 e menor que 1.0.
static long **round**(double a)	Retorna o valor *long* mais próximo do argumento.
static int **round**(float a)	Retorna o valor *int* mais próximo do argumento.
static double **sin**(double a)	Retorna o seno trigonométrico de um ângulo.
static double **sqrt**(double a)	Retorna corretamente arredondado possitivo da raiz quadrada de um valor *double.*

Tabela 4: Métodos da classe Math.

Você pode observar que o método *abs* realiza a sobrecarga, outros também realizam a sobrecarga de métodos, como por exemplo, *min* e *max*. Outra observação é que os métodos são *static*. Desta forma não operam sobre objetos e sim sobre tipos primitivos.

Nas chamadas dos métodos estáticos da *Math* deve-se colocar a palavra *Math* seguida do nome do método, por exemplo, *Math.sqrt(x)*. Essa chamada faz parecer que o método sqrt é aplicado ao objeto *Math*. Entretanto, *Math* é uma classe e não um objeto.

O código a seguir apresenta o uso de alguns métodos da classe *Math* como: round, *pow*, *random*, *sqrt* e *abs*.

```
1.  package ifsul.tsiad.poo.api;
2.
3.  public class TesteMath {
4.      public static void main(String[] args) {
5.          double x = 12.4556;
6.          double y = -4.0;
7.          double z = 2.0;
8.          // imprimindo a valor arrendado para long
9.          System.out.println(Math.round(x));
10.         // imprimindo o valor da raiz quadrada
11.         System.out.println(Math.pow(y, z));
12.         // imprmindo dois números randomincos
13.         System.out.println(Math.random());
14.         System.out.println(Math.random());
15.         // imprmindo a raiz quadrada
16.         System.out.println(Math.sqrt(x));
17.         // imprmindo a o valor absoluto
18.         System.out.println(Math.abs(y));
19.     }
20. }
```

Dica:

A documentação sobre a classe String está no link:

http://java.sun.com/j2se/1.6.0/docs/api/java/lang/Math.html

Classes Date, Calendar e FormatDate (Representação de Datas)

Para manipular datas em Java são utilizadas as classes *Date, Calendar* e *FormatDate*. Para quem está começando a trabalhar com datas em Java pode ser um pouco complicado, já que a classe Date não fornece todos os recursos necessários. Para completá-la é também utilizada a classe Calendar, que é uma classe abstrata que permite a manipulação de datas de forma mais fácil.

A classe *Date* representa um instante específico de tempo, com precisão de milissegundos. Até a versão 1.1 do JDK, a classe Date possui duas funções adicionais: permitia a interpretação das datas como anos, meses, dias, horas, minutos e segundos; e permitia formatar strings de datas. Contudo, estas funções não eram favoráveis para internacionalização.

Após a JDL 1.1, a classe *Calendar* passou a ser utilizada para converter campos de data e tempo e a classe *DateFormat* passou a ser utilizada para formatar strings de datas. Os métodos correspondentes na classe Datem tornaram-se legados.

Normalmente os programadores utilizam variáveis como Date, e na hora de manipular, utilizam *Calendar*. Acontece que alguns métodos de *Calendar* retornam Date, fazendo com que guarde o resultado em um determinado tipo de variável Date.

As classes *Date* e *Calendar* pertencem ao pacote *java.util*. Já a classe *FormatDate* pertence ao pacote *java.text*.

Na classe *Date* para se obter a data atual basta criar um objeto Date (exemplo criar um objeto Date referenciado pela variável de objeto data).

```
Date data = new Date();
```

A classe *Calendar* é uma classe abstrata e não pode ser instanciada com o operador *new*. Porém ela deve ser criada utilizando um método estático *getInstance()*.

```
Calendar c = Calendar.getInstance();
```

Com a instância de *Calendar* podemos configurar uma data e podemos fazer isto através de alguns métodos, como:

```
c.set(ano,mês,dia); // configura uma data (com ano, mês e dia)

c.set(ano,mês,dia,hora,minuto); // configura com data e tempo (minuto)

c.set(ano,mês,dia,hora,minuto,segundo); // configura com data e tempo

c.setTime(new Date()); // configura com a data atual
```

A classe *Calendar* também nos fornece várias constantes e outros métodos para manipulação de datas.

A classe DateFormat é uma classe abstrata para formatação de datas e tempo. A DateFormat fornece vários métodos para obter a data em formatos padrões de data e tempo. Os estilos de formato incluem com os respectivos exemplos:

- SHORT: 17/07/10
- MEDIUM: 17/07/2010
- LONG: 17 de Julho de 2010
- FULL: Sábado, 17 de Julho de 2010

O código a seguir mostra a manipulação de datas em Java com as classes apresentadas.

```
1.  import java.util.*;
2.  import java.text.*;
3.  public class TesteDate {
4.          public static void main(String[] args) {
5.              Date d = new Date();
6.              // imprimindo d instância de Date
7.              System.out.println(d);
8.              // utlizando a classe DateFormat para formatar d
9.              DateFormat dateformat =
10. DateFormat.getDateInstance(DateFormat.SHORT);
11.             System.out.println(dateformat.format(d.getTime()));
12.             // criamos c instânicia de Calendar
13.             Calendar c = Calendar.getInstance();
14.             // formatamos a data novamente
15.             dateformat = DateFormat.getDateInstance(DateFormat.MEDIUM);
16.             // coloca a data atual na instância c
17.             c.setTime(new Date());
18.             System.out.println(dateformat.format(c.getTime()));
19.             // coloca uma data específica na instância c
20.             c.set(1974,3,11);
21.             System.out.println(dateformat.format(c.getTime()));
22.         }
23. }
```

Dica:

A documentação sobre as classes *Date, Calendar* e *FormatDate* estão nos links:

http://java.sun.com/j2se/1.6.0/docs/api/java/util/Date.html
http://java.sun.com/j2se/1.6.0/docs/api/java/util/Calendar.html
http://java.sun.com/j2se/1.6.0/docs/api/java/text/FormatDate.html

Nesta unidade, foram apresentadas algumas das classes da API Java. Na próxima unidade, iniciaremos o estudo de coleções de objetos e tratamento de exceções.

UNIDADE D

COLEÇÕES DE OBJETOS E TRATAMENTO DE EXCEÇÕES

Nesta última unidade, você estudará um pouco sobre as Coleções de Objetos e Tratamento de Exceções. Das interfaces básicas das coleções, aprenderá as que implementam a interface List. Por fim, conhecerá Tratamento de Exceções - um mecanismo de processamento de erros em tempo de execução.

Coleções de Objetos

Uma coleção de objetos é um objeto que representa um grupo de objetos, em outras palavras, é simplesmente um objeto que agrupa múltiplos elementos em uma simples unidade. As coleções também são chamadas *containers*.

As coleções são utilizadas para armazenar, recuperar, manipular, e comunicar dados agregados. São exemplos de coleções: baralho (uma coleção de cartas), caixa de correio (uma coleção de cartas) e lista telefônica (uma coleção de nomes e telefones).

As coleções fornecem estruturas de dados úteis e algoritmos que não necessitam ser escritos pelo programador, reduzindo assim o esforço de programação. Além disso, as coleções fornecem implementações de alto desempenho, reduzem o esforço de aprender APIs *ad hoc* e projetar e implementar novas APIs e também permite o reuso de software.

A linguagem Java define um framework de coleções (*Collections Framework*) que é uma arquitetura unificada para representar e manipular coleções. Este framework de coleções, entre outros, possui:

- **Interface:** são tipos de dados abstratos que representam coleções. As interfaces permitem as coleções serem manipuladas independentemente dos seus detalhes das suas implementações;
- **Implementações:** são as implementações concretas das interfaces das coleções. Na essência, são as estruturas de dados reusáveis;
- **Algoritmos:** são os métodos que executam as computações, como busca e ordenação nos objetos que implementam as interfaces das coleções.

No total são quatorze interfaces das coleções. A interface mais básica é a interface *Collection*. Várias outras interfaces estendem a *Collection*: Set, *List*, *SortedSet*, *NavigableSet*, *Queue*, *Deque*, *BlockingQueue* e *BlockingDeque*. Outras interfaces de coleções são: *Map*, *SortedMap*, *NavigableMap*, *ConcurrentMap* e *ConcurrentNavigableMap* (estas interfaces não estendem *Collection*).

As classes que implementam as interfaces as coleções tipicamente possuem nomes na forma: *<estilo de implementação><interface>*. Por exemplo, existe uma implementação de classe *ArrayList*, o estilo de implementação é um array que implementa a interface *List*. A Figura 1 apresenta as implementações de propósito geral, destacam-se as implementações que implementam as interfaces básicas:

- ***Set* (conjuntos):** definem coleções onde objetos em duplicata não são admitidos;
- ***List* (listas):** definem coleções de objetos onde elementos repetidos podem ocorrer e onde os elementos têm posições definidas;
- ***Map* (mapas):** os mapas são conjunto de pares de objetos (um chamado chave e o outro, chamado *valor*). Estes mapas não permitem chaves repetidas, mas permitem valores repetidos (chaves diferentes podem estar associadas a valores iguais).

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Figura 1. Implementações de propósito geral do *Framework Collections*

O framework de coleções também possui implementações legadas (utilizadas em versões anteriores do Java), como as implementações *Vector* e *HashTable*. A implementação *Vector* foi muito utilizada no passado.

Nesta unidade, estudaremos uma pequena parte do framework de coleções. Nossa atenção será em duas implementações da interface *List*: *ArrayList* e *LinkedList*.

Dica:

Leia mais sobre o framework de coleções na documentação Java (em inglês):

http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/technotes/guides/Collections/overview.html

Lista de Objetos (*List*)

Lista de objetos são coleções em que elementos repetidos podem ocorrer e onde os elementos têm posições definidas. De uma forma geral, podemos considerar com arrays com algumas capacidades adicionais, uma das quais é a capacidade de ter seu tamanho modificado de acordo com a necessidade.

Uma das grandes vantagens das coleções de objetos em geral é que estas estruturas não possuem um tamanho predefinido, diferentemente dos arrays, onde o tamanho de um array não pode ser modificado depois de ele ser criado.

A interface **List** que declara que métodos podem ser utilizados para manipulação de listas e duas classes que implementam esta interface: *ArrayList* e *LinkedList*. Cada uma destas classes possui um mecanismo diferente para a representação interna dos objetos nas listas. As classes *ArrayList* e *LinkedList* são encontradas no pacote *java.util*.

Dica:

A documentação das implementações *ArrayList* e *LinkedList*:

http://java.sun.com/j2se/1.6.0/docs/api/java/util/ArrayList.html
http://java.sun.com/j2se/1.6.0/docs/api/java/util/LinkedList.html

A interface *List* define uma lista de objetos em que elementos repetidos podem ocorrer e onde os elementos têm posições definidas. Na verdade podemos definir uma esta lista de objetos como uma lista. Porém as implementações de *ArrayList* e *LinkedList* possuem implementações internas diferentes:

- **ArrayList** implementa a lista internamente como um array e tem desempenho melhor, exceto por operações como inserção e remoção de elementos da lista;
- **LinkedList** tem melhor desempenho para as operações de inserção e remoção mas é, em geral, mais lenta para acesso sequencial aos elementos da lista. É mais conveniente para implementar pilhas e filas (métodos *addFirst*, *addLast*, *getFisrt*, *getLast*, *removeFisrt*, *removeLast*).

A implementação *ArrayList* pode ser vista como uma **lista de arrays**. Esta lista de arrays é uma sequência ordenada de itens de dados, podendo cada um ser acessado com um índice inteiro. A Figura 2 apresenta a visão abstrata de uma lista de arrays.

Figura 2. Visão abstrata de uma lista de *arrays*

Já, a implementação *LinkedList* pode ser vista como uma **lista encadeada**. Esta lista encadeada é uma sequência ordenada de itens de dados que podem ser percorridos (porém não podem ser acessados diretamente por um índice). A Figura 3 apresenta a visão abstrata de uma lista encadeada.

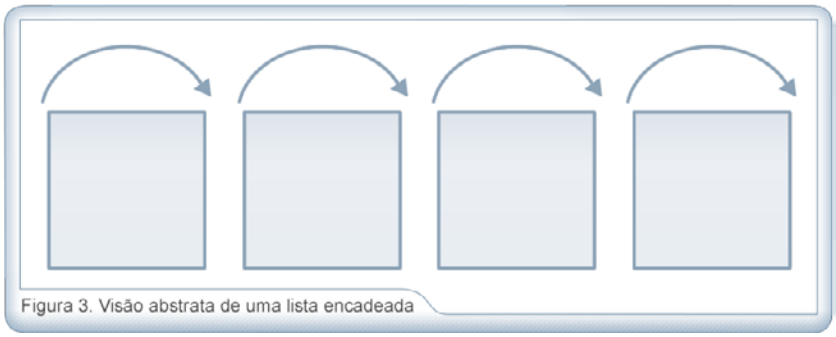


Figura 3. Visão abstrata de uma lista encadeada

As implementações concretas de uma lista encadeada e uma lista de arrays são bem diferentes. Por outro lado, à primeira vista as visões abstratas (Figura 2 e 3) podem parecer semelhantes. A Figura 4 e 5 apresentam, respectivamente, as visões concretas das implementações de lista de arrays e lista encadeada.

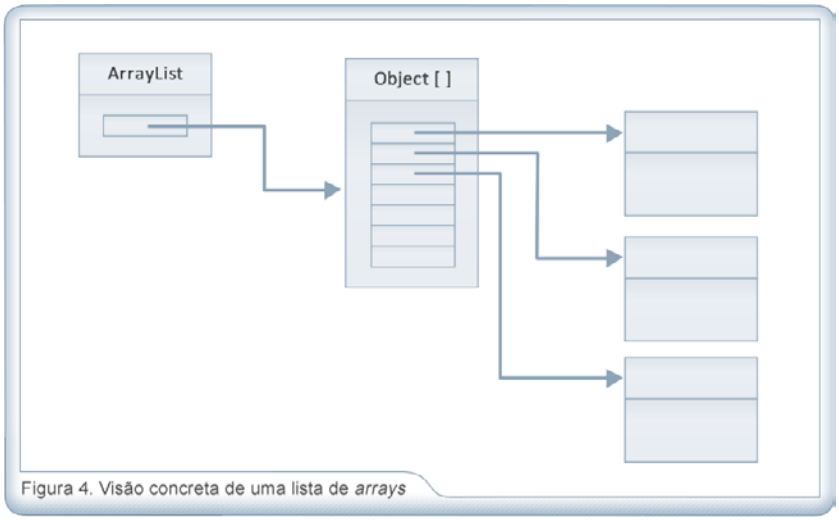


Figura 4. Visão concreta de uma lista de *arrays*

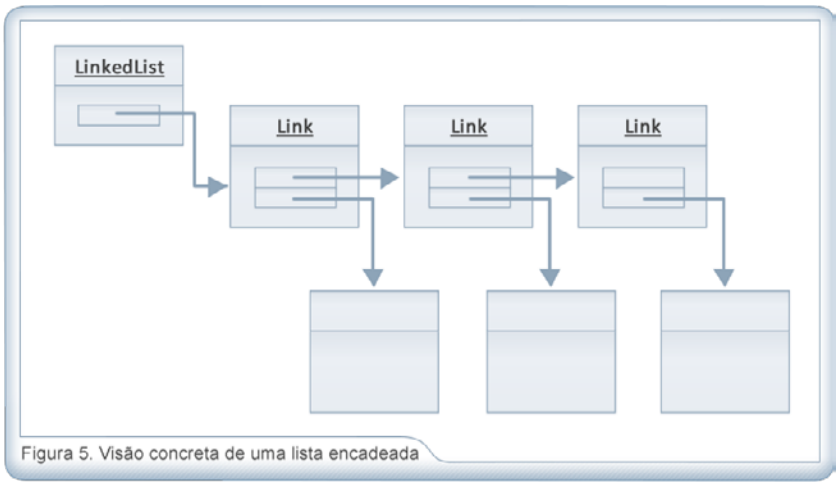


Figura 5. Visão concreta de uma lista encadeada

A lista de arrays permite um **acesso aleatório** a todos os elementos. Você especifica um índice inteiro e pode obter ou configurar o elemento correspondente. Por outro lado, com uma lista encadeada o acesso a um determinado elemento é um pouco mais complexo. Esse tipo de lista permite um acesso sequencial.

Utilizando *ArrayList*

A classe *ArrayList* implementada na API Java possuem uma série de construtores e métodos. A tabela 1 apresenta apenas alguns destes métodos (para todos os métodos consulte a documentação).

Método	Descrição
add(E e)	Adiciona um elemento específico no final da lista.
add(int index, E element)	Insere um elemento específico em uma posição específica da lista
clear()	Remove todos os elementos da lista.
contains(Object o)	Retorna true se a lista contém o elemento especificado.
get(int index)	Retorna o elemento de uma posição específica da lista.
indexOf(Object o)	Retorna o índice da primeira ocorrência do elemento especificado na lista.
isEmpty()	Retorna true se a lista não contém elementos.
remove(int index)	Remove o elemento da posição especificada da lista.
remove(Object o)	Remove a primeira ocorrência de um elemento específico da lista (caso esteja presente).
size()	Retorna o número de elementos da lista.

Tabela 1: Métodos da classe *ArrayList*.

Para utilizar a classe *ArrayList*, é necessário importar a classe do pacote *java.util*. O exemplo a seguir apresenta as seguintes ações:

- Importação da classe *ArrayList* do pacote *java.util* – linha 3;
- Criação de um objeto *ArrayList* (referenciado pela variável de objeto jogadores) – linha 6;
- Adição de objetos do tipo *String* na coleção através da chamada do método *add* – linhas 7 a 9;
- Impressão dos objetos da coleção: a impressão ocorre percorrendo toda a lista através de um laço. O método *size* é utilizado para determinar o limite do laço (retorna o número de elementos da lista) e o método *get* é utilizado para capturar cada objeto da coleção (acessando índice por índice) – linhas 11 a 13.

```
1.  package ifsul.tsiad.poo.colecoes;
2.
3.  import java.util.ArrayList;
4.  public class ArrayTest1 {
5.      public static void main(String[] args) {
6.          ArrayList jogadores = new ArrayList();
7.          jogadores.add("Ronaldo");
8.          jogadores.add("Robinho");
9.          jogadores.add("Kaka");
10.         System.out.println("Impressão do ArrayList (usando indices):");
11.         for(int i=0;i<jogadores.size();i++){
12.            System.out.println(jogadores.get(i));
13.         }
14.     }
15. }
```

O código a seguir apresenta as seguintes ações:

- Importação de todas as classes do pacote *java.util* – linha 3;
- Criação de um objeto *ArrayList* (referenciado pela variável de objeto carros) – linha 6;
- Adição de objetos do tipo Carro na coleção através da chamada do método add – linhas 7 a 9;
- Impressão dos objetos da coleção – linhas 11 a 13;
- Remoção do objeto localizado na posição 1 da lista (no caso o objeto "Chevette") através do método *remove* – linha 14;
- Impressão novamente dos objetos da coleção – linhas 16 a 18.

```
1.  package ifsul.tsiad.poo.colecoes;
2.
3.  import java.util.*;
4.  public class ArrayTest2 {
5.      public static void main(String[] args) {
6.          ArrayList carros = new ArrayList();
7.          carros.add(new Carro("Fusca", "IIJ4949"));
8.          carros.add(new Carro("Chevette", "III7777"));
9.          carros.add(new Carro("Kombi", "YYY1111"));
10.         System.out.println("Impressão do ArrayList (usando indices):");
11.         for(int i=0;i<carros.size();i++){
12.            System.out.println(carros.get(i));
13.         }
14.         carros.remove(1);
15.         System.out.println("Nova impressão:");
16.         for(int i=0;i<carros.size();i++){
17.            System.out.println(carros.get(i));
18.         }
19.     }
20. }
```

Utilizando LinkedList

A classe *LinkedList*, implementada na API Java, possui uma série de construtores e métodos. A tabela 2 apresenta apenas alguns destes métodos (para todos os métodos consulte a documentação).

Método	Descrição
add(E e)	Adiciona um elemento específico no final da lista.
addFirst(E e)	Insere um elemento específico no início da lista
addLast(E e)	Insere um elemento específico no final da lista
clear()	Remove todos os elementos da lista.
getFirst()	Retorna o primeiro elemento da lista.
getLast()	Retorna o último elemento da lista.
poll()	Recupera e remove o elemento cabeça (primeiro elemento) da lista.
pollFirst()	Recupera e remove o primeiro elemento da lista, ou retorna null se a lista é vazia.
pollLast()	Recupera e remove o último elemento da lista, ou retorna null se a lista é vazia.
remove()	Retorna e remove a cabeça (primeiro elemento) da lista.
removeFirst()	Remove e retorna o primeiro elemento da lista.
removeLast()	Remove e retorna o último elemento da lista.
size()	Retorna o número de elementos da lista.

Tabela 2: Métodos da classe *LinkedList*.

Frequentemente teremos que manipular objetos em uma lista encadeada de um a um. Para isso, devemos utilizar um mecanismo de iteração (definido pela interface *Iterator*). A interface *Iterator* permite acessar os elementos dentro de uma lista encadeada. Existe uma interface específica para listas, a *ListIterator*, esta interface declara três métodos de interesse:

- *hasNext*() – retorna true se existe um elemento seguinte;
- *next*() – mover a posição do iterador (retorna o próximo objeto do iterador);
- *remove*() – remove o último elemento retornado pela lista da coleção.

O código a seguir apresenta as seguintes ações:

- Importação da classe *LinkedList* e *ListIterator* do pacote *java.util* – linhas 3 e 4;
- Criação de um objeto *LinkedList* (referenciado pela variável de objeto jogadores) – linha 7;
- Adição de objetos do tipo *String* na coleção através da chamada do método *addLast* – linhas 8 a 10;
- Para percorrer a lista é necessário criar um iterador. A variável *iterator* recebe o valor retornado pelo método *listIterator*() da classe *LinkedList*, este método obtém um iterador de lista (aponta para o primeiro elemento) – linha 12;
- Impressão dos objetos da coleção: para percorrer a lista é simples, basta criar um laço e chamar o método *hasNext* do iterador, este método retorna true se a lista ainda contêm elementos – linhas 13 a 15;
- Por fim, dentro do laço é chamado o método *next* do iterador, este método retorna o próximo objeto da lista de iteração.

```
1.  package ifsul.tsiad.poo.colecoes;
2.
3.  import java.util.LinkedList;
4.  import java.util.ListIterator;
5.  public class LinkedListTest1 {
6.      public static void main(String[] args) {
7.          LinkedList jogadores = new LinkedList();
8.          jogadores.addLast("Ronaldo");
9.          jogadores.addLast("Robinho");
10.         jogadores.addLast("Ronaldinho");
11.         System.out.println("Impressao da LinkedList:");
12.         ListIterator iterator = jogadores.listIterator();
13.         while(iterator.hasNext()){
14.            System.out.println(iterator.next());
15.         }
16.     }
17. }
```

O código a seguir apresenta o mesmo código do exemplo anterior, porém após a impressão foi colocado um novo código que permite percorrer a lista fora do laço. As seguintes ações:

- A variável *iterator* recebe o valor retornado pelo método *listIterator*() da classe *LinkedList* (apontando novamente para o primeiro elemento) – linha 16;
- Após é realizada duas chamadas do método *next*, posicionando o iterador entre o segundo e terceiro elemento da coleção – linhas 17 e 18;
- Realiza a adição de um novo objeto *String* (“MANUEL”) na coleção – linha 19;
- Por fim, realiza a impressão novamente dos objetos da coleção.

```
1.  package ifsul.tsiad.poo.colecoes;
2.
3.  import java.util.LinkedList;
4.  import java.util.ListIterator;
5.  public class LinkedListTest2 {
6.      public static void main(String[] args) {
7.          LinkedList jogadores = new LinkedList();
8.          jogadores.addLast("Ronaldo");
9.          jogadores.addLast("Robinho");
10.         jogadores.addLast("Ronaldinho");
11.         System.out.println("Impressao da LinkedList:");
12.         ListIterator iterator = jogadores.listIterator();
13.         while(iterator.hasNext()){
14.            System.out.println(iterator.next());
15.         }
16.         iterator = jogadores.listIterator();
17.         iterator.next();
18.         iterator.next();
19.         iterator.add("MANUEL");
20.         System.out.println("Nova Impressao da LinkedList:");
21.         iterator = jogadores.listIterator();
22.         while(iterator.hasNext()){
23.             System.out.println(iterator.next());
24.         }
25.     }
26. }
```

O iterador também pode ser utilizado em uma implementação com *ArrayList*. O exemplo a seguir apresenta isto.

```
1.  package ifsul.tsiad.poo.colecoes;
2.
3.  import java.util.*;
4.  public class ArrayTest3 {
5.      public static void main(String[] args) {
6.          ArrayList jogadores = new ArrayList();
7.          jogadores.add("Ronaldo");
8.          jogadores.add("Robinho");
9.          jogadores.add("Kaka");
10.         System.out.println("Impressão do ArrayList (usando indices):");
11.         ListIterator iterator = jogadores.listIterator();
12.         while(iterator.hasNext()){
13.            System.out.println(iterator.next());
14.         }
15.     }
16. }
```

Utilize referências da interface para manipular as estruturas de dados. Por exemplo, armazene uma referência a um *ArrayList* ou *LinkedList* em uma variável do tipo *List*. Por exemplo, declare assim:

```
List nomes = new ArrayList();
```

Desta forma, o programador tem de alterar apenas uma linha se decidir usar um *LinkedList* em vez de um *ArrayList*. Os métodos que operam sobre as listas devem especificar parâmetros do tipo *List*. Por exemplo, uma declaração de método:

```
public static void print(List lista)
```

Dica:

Procure mais informações sobre o framework de coleções e conheça as outras implementações das interfaces *Set* e *Map* e as *Wrapper Classes*. Elas são bem interessantes.

2. Tratamento de Exceções

Existem diversos problemas que podem ocorrer em programas como entradas inválidas, problemas de acesso a arquivos, entre ouros. Ocorrem muitas situações anormais (exceções) ou inválidas (erros) durante o processo de execução e deve existir uma maneira do programa lidar com as falha de uma maneira previsível: detectando e recuperando esta falha.

Os erros em tempo de execução podem ser oriundos de diferentes fontes como:

- Erros de lógica de programação, exemplos: limites de vetores e divisões por zero;
- Erros devido a condições do ambiente de execução, exemplos: arquivo não encontrado e rede fora do ar;
- Erros graves sem possibilidade de recuperação, exemplos: falta de memória, erro interno da JVM e falta de espaço em disco.

Java permite o uso de mecanismo flexível de processamento de erros em tempo de execução. Este mecanismo passa o controle do ponto de detecção de erro para um tratador competente de recuperação através das instruções *try-catch* que servem para capturar exceções. Estas exceções são objetos criados a partir de classes especiais que são lançados quando ocorre uma exceção.

Uma exceção interrompe o fluxo de execução do programa. Este fluxo segue a exceção e se o método onde ela ocorrer não capturar, ela será propagada para o método que chamar este método e assim por diante. Se não houver captura da exceção, ela irá causar o término do programa, porém se ela for capturada o controle pode ser recuperado.

Existem dois tipos de exceções, as verificadas e não-verificadas. As exceções verificadas acontecem quando ao chamar um método que lança uma exceção, o programador deve dizer ao compilador o que está fazendo sobre a exceção se ela for lançada, ou seja, o compilar verifica se seu programa gerencia essas exceções. Em geral as exceções verificadas estendem a classe *Exception* (exemplos: *IOException*, FileNotFound*Exception*). Já as não verificadas estendem a classe *RuntimeException* ou *Error* (exemplos: *NumberFormatException*, *IllegalArgumentException*, *NullPointerException*).

A Figura 6 apresenta a hierarquia das classes de erros e exceções. As classe que estendem *Error* e *RuntimeException* não são verificadas pelo compilador, enquanto que as classes que estendem *Exception* ao verificadas.

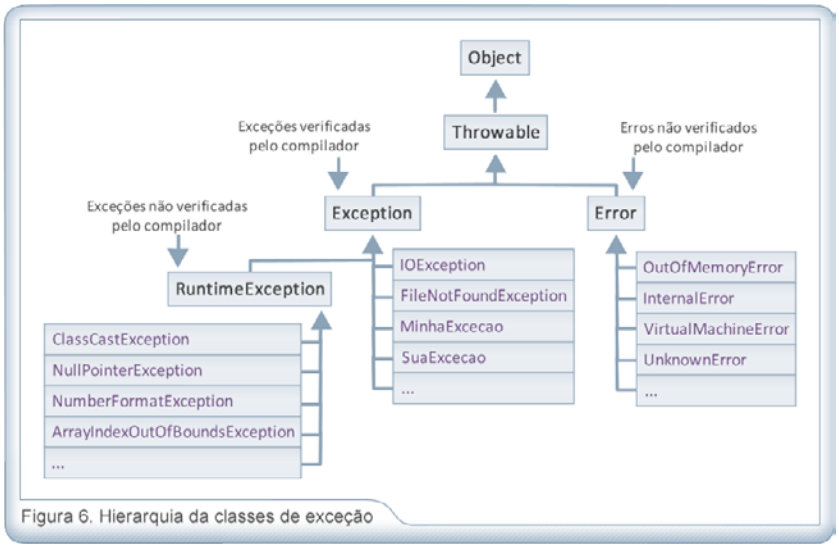


Figura 6. Hierarquia da classes de exceção

Dentro de um código, você pode lançar uma exceção através da instrução *throw*, capturar exceções através de um bloco *try-catch* e também projetar as suas próprias classes de exceções. Nesta unidade nós vamos nos focar apenas em capturar exceções.

O código a seguir mostra um exemplo de uma exceção não-verificada. Você pode compilar o código sem problemas, porém durante a execução irá acontecer um erro (será lançada uma exceção). No caso específico, na linha 9 é realizado um acesso a um posição (4) inexistente no array. Na execução do programa, será lançada uma exceção *ArrayIndexOutOfBoundsException*.

```
1.  package ifsul.tsiad.poo.excecoes;
2.
3.  public class TesteExcecoes1 {
4.      public static void main(String[] args) {
5.          int vetor[] = new int[3];
6.          vetor[0]=1;
7.          vetor[1]=2;
8.          vetor[2]=3;
9.          vetor[4]=5;
10.     }
11. }
```

Capturando Exceções

Cada exceção deve ser tratada em algum lugar no programa. Se uma exceção não é tratada, é impressa uma mensagem de erro e o programa termina. Para instalar um tratador de exceções, você deve colocar as instruções que podem causar uma exceção dentro de um bloco *try* e o tratador da exceção dentro da cláusula *catch*. Cada bloco *try* contém uma ou mais chamadas de método que podem causar uma exceção e cláusulas *catch* para todos os tipos de exceção que o bloco *try* pode tratar

Até o presente momento nós não utilizamos a leitura do console (ler do teclado). Isso se deve ao fato que o comum é usar a instância *System.in* da classe *InputStream* que permite a leitura de bytes ou linhas de texto como *strings*.

O código a seguir mostra a captura de uma exceção. Contudo, a primeira coisa que você deve verifica neste código é retirar as cláusulas *try* e *catch*. Você notará que o compilador informará erro, pois a exceção é verificada, ou seja, o programador para utilizar a classe *InputStreamReader* deverá utilizar o mecanismo de tratamento de exceções.

O código a realiza as seguintes ações:

- Devemos importar as classes *BufferedReader*, *IOException* e *InputStreamReader* do pacote *java.io* – linhas 3 a 5;
- O código que pode potencialmente gerar uma exceção é colocado no bloco *try*;
- É realiza a criação de um buffer de leitura *BufferedReader* que permitirá a leitura de bytes ou *strings* do console – linhas 10 e 11;
- É chamado o método *readLine* da classe *BufferedReader* que retorna uma linha armazenada no buffer – linha 13;
- É chamado o método *parseInt* da classe *Integer* para transformar uma *String* em um dados do tipo inteiro – linha 14;
- Por fim, dentro da cláusula *catch* deve ser colocado o que deve ser feito se acontecer uma exceção, neste caso apenas um impressão.

Neste exemplo:

- Caso não ocorra uma exceção: as linhas de código dentro do bloco *try* são executadas normalmente e o bloco *catch* é ignorado;
- Caso ocorra a exceção: as linhas de código dentro do bloco *try* são executadas até o ponto que gerou a exceção (as demais linhas são ignoradas) e as instruções do bloco *catch* são executadas.

```
1.  package ifsul.tsiad.poo.excecoes;
2.
3.  import java.io.BufferedReader;
4.  import java.io.IOException;
5.  import java.io.InputStreamReader;
6.
7.  public class TesteExcecoes2 {
8.      public static void main(String[] args) {
9.          try {
10.             BufferedReader in = new BufferedReader(new
11. InputStreamReader(System.in));
12.             System.out.println("Quantos anos voce tem ?");
13.             String inputLine = in.readLine();
14.             int idade = Integer.parseInt(inputLine);
15.             idade++;
16.             System.out.println("No próximo ano voce terá: " + idade);
17.         }
18.         catch (IOException exception){
19.             System.out.println("Erro de I/O: " + exception);
20.         }
21.     }
22. }
```

O código a seguir mostra um exemplo capturando múltiplas exceções. Para isto, o programador deve colocar várias cláusulas *catch* (uma para cada exceção que deseja capturar. No exemplo, caso você digite uma letra por exemplo (ao invés de um número) será lançada uma exceção *NumberFormatException* no momento de conversão do valor de *string* para inteiro (realizado pelo método *parseInt*).

Neste exemplo:

- Caso não ocorra uma exceção: as linhas de código dentro do bloco *try* são executadas normalmente e os blocos *catch* são ignorados;
- Caso ocorra a exceção: as linhas de código dentro do bloco *try* são executadas até o ponto que gerou a exceção (as demais linhas são ignoradas) e as instruções do bloco *catch* correspondente são executadas.

```
1.  package ifsul.tsiad.poo.excecoes;
2.
3.  import java.io.*;
4.  public class TesteExcecoes3 {
5.      public static void main(String[] args) {
6.          try {
7.              BufferedReader in = new BufferedReader(new
8.  InputStreamReader(System.in));
9.              System.out.println("Quantos anos voce tem ?");
10.             String inputLine = in.readLine();
11.             int idade = Integer.parseInt(inputLine);
12.             idade++;
13.             System.out.println("No próximo ano voce terá: " + idade);
14.         }
15.         catch (IOException exception){
16.             System.out.println("Erro de I/O: " + exception);
17.         }
18.         catch (NumberFormatException exception){
19.             System.out.println("A entrada não é um número ! ");
20.         }
21.     }
22. }
```

No exemplo de código a seguir, quando ocorre a exceção a cláusula *catch* não determina com precisão qual exceção ocorreu (foi capturada uma exceção do tipo *Exception* – a mais geral na hierarquia) e isto é muito utilizado para o compilador não reclamar da verificação de exceções, porém não é bom utilizar. Este tratador é fictício, já as exceções foram projetadas para transmitir relatórios do problema para um tratador competente.

```
1.  package ifsul.tsiad.poo.excecoes;
2.
3.  import java.io.*;
4.  public class TesteExcecoes4 {
5.      public static void main(String[] args) {
6.          try {
7.              BufferedReader in = new BufferedReader(new
8.  InputStreamReader(System.in));
9.              System.out.println("Quantos anos voce tem ?");
10.             String inputLine = in.readLine();
11.             int idade = Integer.parseInt(inputLine);
12.             idade++;
13.             System.out.println("No próximo ano voce terá: " + idade);
14.         }
15.         catch (Exception exception){
16.             // terminamos com as exceçoes
17.             // na verdade elas não foram tratadas corretamente
18.         }
19.     }
20. }
```

Por fim, o último exemplo apresenta o uso da cláusula *finally*. Quando é necessário fazer algo no caso de ocorrência de qualquer exceção. O código da cláusula *finally* é executado sempre que o fluxo de código sai do bloco *try* de qualquer uma das maneiras:

- Após completar a última instrução do bloco *try*;
- Quando uma exceção foi lançada no bloco *try* que está sendo passado para esse chamador do método;
- Quando uma exceção foi lançada no bloco *try* que foi tratado por uma das cláusulas *catch*.

```
1.  package ifsul.tsiad.poo.excecoes;
2.
3.  import java.io.*;
4.  public class TesteExcecoes5 {
5.      public static void main(String[] args) {
6.          try {
7.              BufferedReader in = new BufferedReader(new
8.  InputStreamReader(System.in));
9.              System.out.println("Quantos anos voce tem ?");
10.             String inputLine = in.readLine();
11.             int idade = Integer.parseInt(inputLine);
12.             idade++;
13.             System.out.println("No práximo ano voce terá: " + idade);
14.         }
15.         catch (IOException exception){
16.             System.out.println("Erro de I/O: " + exception); }
17.         catch (NumberFormatException exception){
18.             System.out.println("A entrada não é um número ! ·,· }
19.         finally{
20.             System.out.println("É sempre executado independentemente de
21. ocorrer uma exceção !"); }
22.         }
23. }
```

Dica:

Estude mais sobre exceções. Procure informações de como você pode lançar uma exceção e projetar as suas próprias classes de exceções.

Conclusão

Esta unidade finaliza o estudo de Orientação a Objetos e da linguagem Java nesta disciplina. Porém, ainda em outras duas disciplinas deste curso os conceitos de OO e Java serão tratados. No entanto, a tecnologia Java é enorme e muitos outros conceitos podem ser estudados.