J100

Programação orientada a objetos com Java™ 2 SE

Helder da Rocha (helder@acm.org)

argonavis.com.br

Objetivos

- *Este curso tem como objetivo iniciá-lo em Java*
  - *... mas não apenas isto*
- *Visa também a ajudá-lo a desenvolver boas práticas de desenvolvimento que tornarão sua vida de programador mais fácil no futuro.*
  - *Uso eficiente da documentação,*
  - *Uso de diagramas de classe,*
  - *Estilo e convenções de codificação,*
  - *Técnicas de depuração, testes e manutenção*
  - *Uso e conhecimento de padrões de projeto*

Pré-requisito fundamental

Saber programar em C

ou

Saber programar em uma linguagem orientada a objetos

Assuntos abordados

- Este curso explora os seguintes assuntos
  - Como escrever programas em Java, como compilá-los e como executá-los
  - Conceitos essenciais de programação orientada a objetos
  - A sintaxe da linguagem Java
  - Recursos fundamentais como estruturas de dados utilitárias, manipulação de strings, leitura e gravação de bytes e caracteres
  - Tópicos essenciais de programação concorrente
  - Como utilizar a documentação
  - Boas práticas, testes e noções de padrões de projeto
  - Como montar um ambiente de desenvolvimento Java baseado no Ant e outras ferramentas de código aberto

Assuntos abordados superficialmente

- Os seguintes assuntos são abordados de maneira muito superficial e incompleta neste curso
  - Como criar aplicações gráficas usando JFC/Swing (javax.swing) ou AWT e Applets
  - Como desenvolver aplicações integradas a bancos de dados usando JDBC (java.sql)
  - Como desenvolver aplicações de rede (java.net)
  - Como desenvolver aplicações de objetos distribuídos (java.rmi e org.omg.CORBA)
  - Multithreading
  - Expressões regulares, reflection, class loader
  - Padrões de projeto (design patterns)

Assuntos não abordados

- Os seguintes assuntos não são abordados neste curso
  - Programação elementar estruturada (estruturas de dados, variáveis, funções, laços de controle, compilação, pilhas, algoritmos, etc.): como foi mencionado antes, saber programar é um pré-requisito essencial para este curso.
  - Como desenvolver aplicações para a Web (servlets e JSP)
  - Como criar aplicações e componentes para servidores de aplicação transacionais (Enterprise JavaBeans)
  - Análise, design ou técnicas e práticas para desenvolver aplicações robustas, seguras e eficientes
  - UML (Unified Modelling Language)
  - Uso de ambientes integrados (IDEs) gráficos, debuggers, profilers, sistemas de controle de versão, etc.

Roteiro (I)

- Parte I - Introdução prática
  - 1. Conceitos básicos e plataforma Java
  - 2. Programação orientada a objetos
  - 3. Como configurar e usar o ambiente
  - 4. Como usar a documentação da API
- Parte II - Programação OO com Java
  - 5. Tipos, literais, operadores e controle de fluxo
  - 6. Como criar classes e objetos
  - 7. Pacotes e encapsulamento
  - 8. Gerenciamento de projetos com o Ant

A distribuição de tempo em cada módulo é iregular. Alguns são bem mais longos que outros

Roteiro (II)

- *Parte III - Mais programação OO com Java*
  - *9. Reuso com herança e composição*
  - *10. Interfaces e polimorfismo*
  - *11. Controle de erros, exceções e asserções*
  - *12. Testes de unidade com o JUnit **
- *Parte IV - Threads, strings e I/O*
  - *13. Fundamentos de programação concorrente*
  - *14. Coleções, propriedades, resources e strings*
  - *15. Entrada e saída, logs e serialização*
  - *16. Classes internas*

* *módulo opcional*

Roteiro (III)

- *Parte V - Swing e persistência de dados*
  - *17. Fundamentos de Swing e aplicações gráficas*
  - *18. Fundamentos de JDBC (java.sql)*
  - *19. Fundamentos de Sockets (java.net)*
  - *20. Fundamentos de Objetos remotos (java.rmi) **

A abordagem dos assuntos nos módulos da parte V é superficial.

** módulo opcional*

Características importantes sobre este curso

- Este curso dedica 70% do tempo à aprendizagem dos conceitos fundamentais da linguagem
  - Orientação a objetos
  - Metologias de desenvolvimento, padrões
  - Boas práticas, testes, roteiros, organização
  - Utilização da documentação
- Por outro lado, sobra menos tempo para tratar de APIs (são abordadas superficialmente)
  - Swing, sockets, I/O, JDBC, RMI, Reflection
  - Muitas exigiriam bem mais tempo
- Se seus conceitos são sólidos, aprender novas APIs será muito mais fácil

Metas

- *Ao final deste curso você deve deve ser capaz de*
  - *Desenvolver aplicações simples em Java (inclusive aplicações gráficas), compilá-las e executá-las*
  - *Analisar programas maiores, identificar seus componentes e compreender seu funcionamento*
  - *Consultar a documentação da API e descobrir como usar novas classes, objetos e métodos.*
  - *Descrever os principais recursos do pacote Java 2 SE*
  - *Construir e utilizar um ambiente de desenvolvimento Java baseado em ferramentas gratuitas*
  - *Explorar assuntos mais complexos em OO e Java*
  - *Entender os assuntos requeridos para a certificação de programador Java da Sun*

Como tirar o melhor proveito deste curso

- *Faça perguntas*
- *Faça os exercícios*
- *Explore os exemplos*
- *Vá além dos exemplos e exercícios: invente exemplos similares, teste trechos de código*
- *Explore e se familiarize com a documentação*
- *Procure desenvolver um projeto que utilize Java, seja no trabalho, seja no seu tempo livre*
  - *Não fique sem programar nos próximos meses ou todo o esforço terá sido em vão!*
- *Leia revistas, artigos e livros sobre Java e mantenha-se atualizado.*

O todo e as partes

- Este curso introduz uma nova linguagem e muitos novos conceitos
- Como qualquer novo conhecimento, vários de seus conceitos mais complexos dependem de outros mais simples
- Nem sempre é possível compreender um conceito mais abrangente na primeira vez
  - Ele as vezes depende do conhecimento de partes que só poderão ser abordadas mais adiante
  - Mas as partes, às vezes dependem dele!
- Solução: repetição. Assuntos complexos serão abordados superficialmente e depois revisitados mais de uma vez
  - Se tiver dúvidas, pergunte na hora
  - Cada dia haverá mais dúvidas novas e menos dúvidas antigas

Exercícios, testes e projetos

- Exercícios são propostos ao final de cada módulo
  - Incluídos na carga-horária
  - Geralmente aplicações triviais (para fixar conceitos)
  - Distribuídos separadamente
- Projetos e testes (opcionais)
  - Não incluídos na carga-horária
  - Projetos usando Java e as principais APIs do J2SE são propostos para quem desejar fixar os conceitos aprendidos. Alguns são aplicações que devem ser completadas. Fazer pelo menos um dos projetos (leva + ou - entre 2 e 8 horas) é fortemente recomendado
  - Testes (similares aos de certificação) são propostos como uma revisão dos principais conceitos abordados no curso

Fontes suplementares

- Este material serve apenas de roteiro de aula
  - Use-o como um resumo
- Para informações mais detalhadas, exemplos extras, testes e projetos utilize um dos livros abaixo
  - "Thinking in Java 2", Bruce Eckel www.bruceeckel.com (PDF - download gratuito) - livro-texto principal
  - "The Java Tutorial", da Sun, por Mary Campione e Kathy Walrath. java.sun.com/tutorial - livro-texto para exemplos com Swing
  - "Java: como programar", Deitel & Deitel (em português)
  - "Aprenda OO em 21 dias", A. Sintes (em português)
  - "Core Java 2", Cay Hortsmann et al. (em português)

Apresentações

- Instrutor: Helder da Rocha (helder@acm.org)
  - Utiliza Java desde 1995
  - XML, J2EE, JSP, servlets, Web
  - http://www.argonavis.com.br
- Alunos?
  - Nome?
  - O que faz? Onde trabalha?
  - Background (sabe C? Java? Web? Que linguagem?)
  - Expectativas?

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Investindo em Java desde 1995

Introdução à tecnologia Java

Helder da Rocha
www.argonavis.com.br

Assuntos abordados neste módulo

- Conceitos
  - Tecnologia Java
  - Linguagem e API Java
  - Máquina virtual Java
  - Ambiente de execução (JRE) e desenvolvimento (SDK)
  - Carregador de classes (ClassLoader) e CLASSPATH
  - Verificador de bytecodes
  - Coletor de lixo (garbage collector)
- Introdução prática
  - Como escrever uma aplicação Java
  - Como compilar uma aplicação Java
  - Como executar uma aplicação Java
  - Como depurar erros de compilação e execução

Parte 1: Tecnologia Java

- O nome "Java" é usado para referir-se a
  - Uma **linguagem de programação** orientada a objetos
  - Uma coleção de **APIs** (classes, componentes, frameworks) para o desenvolvimento de aplicações multiplataforma
  - Um **ambiente de execução** presente em browsers, mainframes, SOs, celulares, palmtops, cartões inteligentes, eletrodomésticos
- Java foi lançada pela Sun em 1995. Três grandes revisões
  - Java Development Kit (JDK) 1.0/1.0.2
  - Java Development Kit (JDK) 1.1/1.1.8
  - Java 2 Platform (Java 2 SDK e JRE 1.2, 1.3, 1.4)
- A evolução da linguagem é controlada pelo **Java Community Process** (www.jcp.org) formado pela Sun e usuários Java
- Ambientes de execução e desenvolvimento são fornecidos por fabricantes de hardware e software (MacOS, Linux, etc.)

Linguagem Java

- Linguagem de programação orientada a objetos
  - Familiar (sintaxe parecida com C)
  - Simples e robusta (minimiza bugs, aumenta produtividade)
  - Suporte nativo a threads (+ simples, maior portabilidade)
  - Dinâmica (módulos, acoplamento em tempo de execução)
  - Com coleta de lixo (menos bugs, mais produtividade)
  - Independente de plataforma
  - Segura (vários mecanismos para controlar segurança)
  - Código intermediário de máquina virtual interpretado (compilação rápida - + produtividade no desenvolvimento)
  - Sintaxe uniforme, rigorosa quanto a tipos (código mais simples, menos diferenças em funcionalidades iguais)

Produtos e APIs

- *Java possui uma coleção de APIs (bibliotecas) padrão que podem ser usadas para construir aplicações*
  - *Organizadas em **pacotes** (`java.*`, `javax.*` e extensões)*
  - *Usadas pelos ambientes de execução (**JRE**) e de desenvolvimento (**SDK**)*
- *As principais APIs são distribuídas juntamente com os produtos para desenvolvimento de aplicações*
  - ***Java 2 Standard Edition (J2SE)**: ferramentas e APIs essenciais para qualquer aplicação Java (inclusive GUI)*
  - ***Java 2 Enterprise Edition (J2EE)**: ferramentas e APIs para o desenvolvimento de aplicações distribuídas*
  - ***Java 2 Micro Edition (J2ME)**: ferramentas e APIs para o desenvolvimento de aplicações para aparelhos portáteis*

Ambiente de execução e desenvolvimento

- ***Java 2 System Development Kit*** *(J2SDK)*
  - *Coleção de ferramentas de linha de comando para, entre outras tarefas, compilar, executar e depurar aplicações Java*
  - *Para habilitar o ambiente via linha de comando é preciso colocar o caminho* `$JAVA_HOME/bin` *no PATH do sistema*
- ***Java Runtime Environment*** *(JRE)*
  - *Tudo o que é necessário para executar aplicações Java*
  - *Parte do J2SDK e das principais distribuições Linux, MacOS X, AIX, Solaris, Windows 98/ME/2000 (exceto XP)*
- *Variável JAVA_HOME (opcional: usada por vários frameworks)*
  - *Defina com o local de instalação do Java no seu sistema. Exemplos:*
  - *Windows:* `set JAVA_HOME=c:\j2sdk1.4.0`
  - *Linux:*
    ```
    JAVA_HOME=/usr/java/j2sdk1.4.0
    export JAVA_HOME
    ```

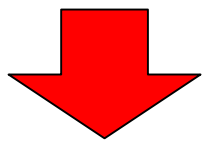
Compilação para bytecode

- **Bytecode** é o código de máquina que roda em qualquer máquina através da **Máquina Virtual Java** (JVM)
- Texto contendo código escrito em linguagem Java é traduzido em bytecode através do processo de **compilação** e armazenado em um arquivo `*.class` chamado de **Classe Java**

Código Java (texto)

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

HelloWorld.java



compilação (`javac`)

HelloWorld.class

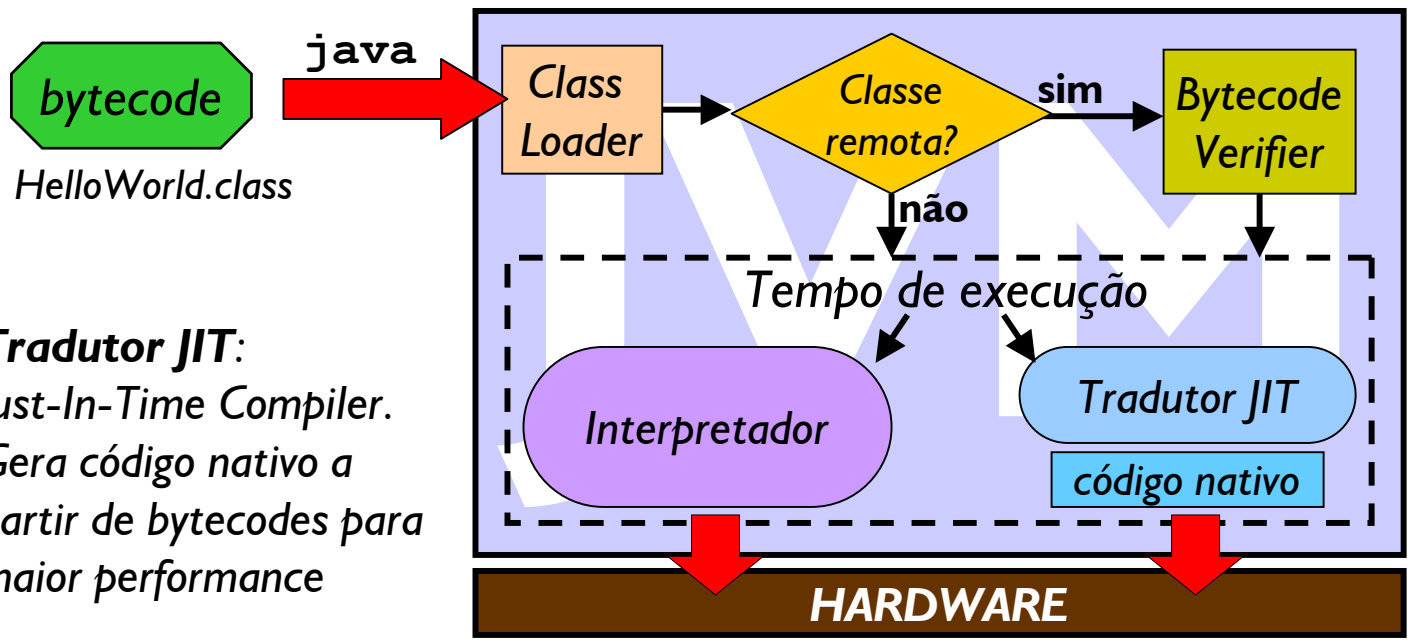
```
F4 D9 00 03 0A B2 FE FF FF 09 02 01 01 2E 2F 30 62 84 3D 29 3A C1
```

Bytecode Java (código de máquina virtual)

Uma "classe" Java

Máquina Virtual Java (JVM)

- "*Máquina imaginária implementada como uma aplicação de software em uma máquina real*" [JVMS]
- *A forma de execução de uma aplicação depende ...*
  - *... da origem do código a ser executado (remoto ou local)*
  - *... da forma como foi implementada a JVM pelo fabricante (usando tecnologia JIT, HotSpot, etc.)*



Tradutor JIT:
Just-In-Time Compiler. Gera código nativo a partir de bytecodes para maior performance

Class Loader e CLASSPATH

- *Primeira tarefa executada pela JVM: carregamento das classes necessárias para rodar a aplicação. O* ***Class Loader***
  1. *Carrega primeiro as classes nativas do* ***JRE*** *(APIs)*
  2. *Depois carrega extensões do* ***JRE****: JARs em* `$JAVA_HOME/jre/lib/ext` *e classes em* `$JAVA_HOME/jre/lib/classes`
  3. *Carrega classes do sistema local (a ordem dos caminhos no* ***CLASSPATH*** *define a precedência)*
  4. *Por último, carrega possíveis classes remotas*
- ***CLASSPATH****: variável de ambiente local que contém todos os caminhos locais onde o Class Loader pode localizar classes*
  - *A CLASSPATH é lida depois, logo, suas classes nunca substituem as classes do JRE (não é possível tirar classes JRE do CLASSPATH)*
  - *Classes remotas são mantidas em área sujeita à verificação*
  - *CLASSPATH pode ser redefinida através de parâmetros durante a execução do comando* `java`

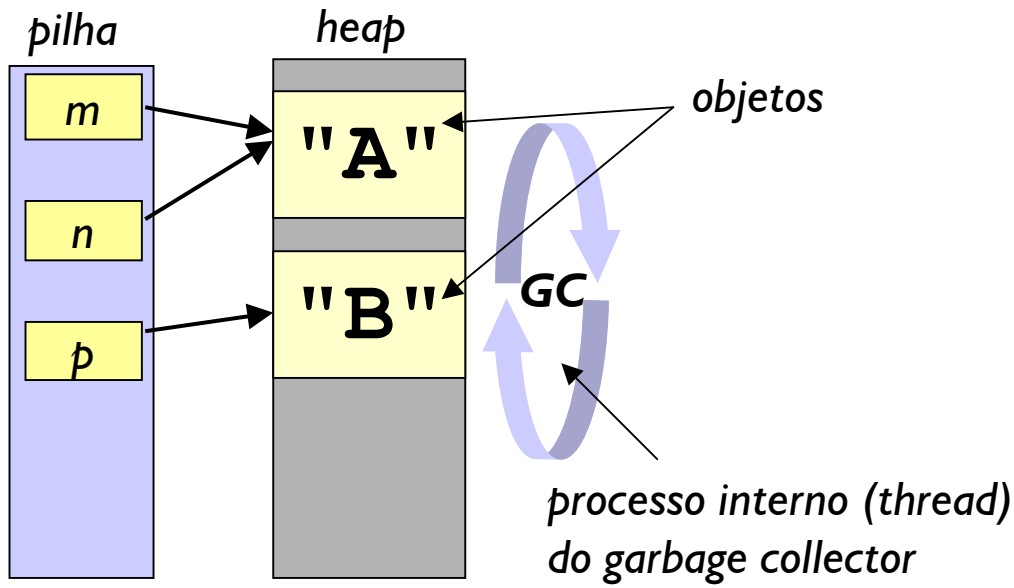
Bytecode Verifier

- *Etapa que antecede a execução do código em classes carregadas através da rede*
  - *Class Loader distingue classes locais (seguras) de classes remotas (potencialmente inseguras)*
- *Verificação garante*
  - *Aderência ao formato de arquivo especificado* [JVMS]
  - *Não-violação de políticas de acesso estabelecidas pela aplicação*
  - *Não-violação da integridade do sistema*
  - *Ausência de estouros de pilha*
  - *Tipos de parâmetros corretamente especificados e ausência de conversões ilegais de tipos*

Coleta de lixo

- Memória alocada em Java não é liberada pelo programador
  - Ou seja, objetos criados não são destruídos pelo programador
- A criação de objetos em Java consiste de
  1. Alocar memória no heap para armazenar os dados do objeto
  2. Inicializar o objeto (via construtor)
  3. Atribuir endereço de memória a uma variável (referência)
- Mais de uma referência pode apontar para o mesmo objeto

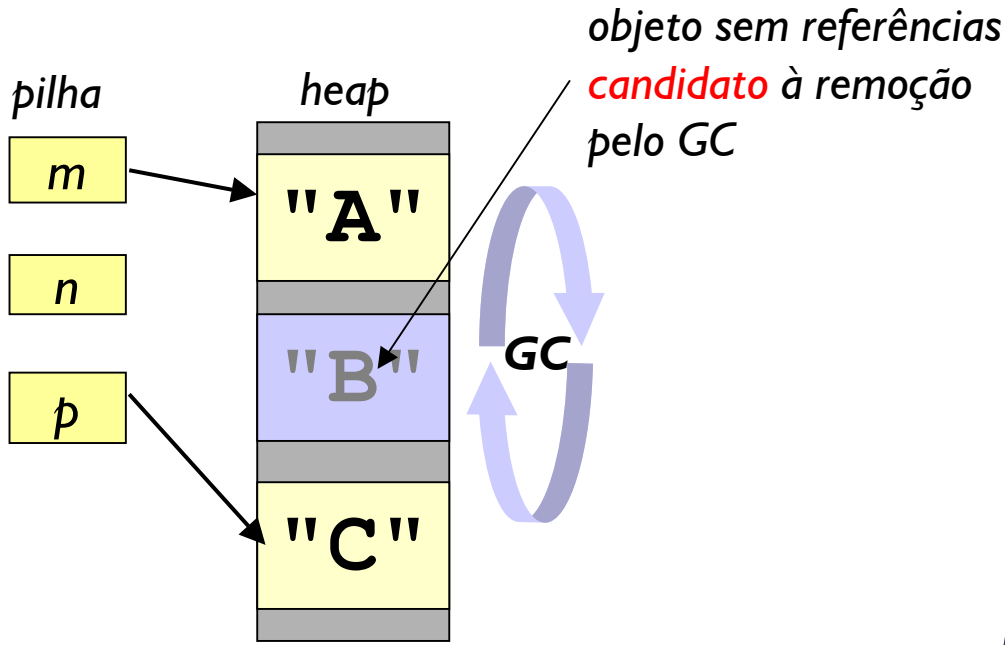
```
Mensagem m, n, p;

m = new Mensagem("A");

n = m;

p = new Mensagem("B");
```



Coleta de lixo (2)

- *Quando um objeto não tem mais referências apontando para ele, seus dados não mais podem ser usados, e a memória deve ser liberada.*
- *O coletor de lixo irá liberar a memória na primeira oportunidade*

```
n = null;
p = new Mensagem("C");
```



O que Java não faz

- Java não suporta herança múltipla de implementação
  - Herança múltipla é característica comum a várias linguagens OO, e permite reuso de código de várias classes em outra classe
  - Tem vantagens porém aumenta a complexidade
  - Java oferece uma solução que preserva as principais vantagens da herança múltipla e evita os problemas
- Java não suporta aritmética de ponteiros
  - Ponteiros, ou referências, são usados em várias linguagens, inclusive Java, para manipular eficientemente grandes quantidades de informação na memória
  - Com ponteiros, em vez de copiar uma informação de um lugar para outro, copia-se apenas o seu endereço
  - Em linguagens como C, o programador pode manipular o endereço (que é dependente de plataforma) diretamente
  - Isto aumenta a complexidade e diminui a portabilidade

O J2SDK

- O J2SDK (**J**ava **2** **S**ystem **D**evelopment **K**it) é o ambiente padrão distribuído pela Sun para desenvolvimento de aplicações Java
- O J2SDK consiste de
  - JRE (Java Runtime Environment) - também distribuído separadamente: ambiente para execução de aplicações
  - Ferramentas para desenvolvimento: compilador, debugger, gerador de documentação, empacotador JAR, etc.
  - Código-fonte das classes da API
  - Demonstrações de uso das APIs, principalmente Applets, interface gráfica com Swing e recursos de multimídia
- A documentação é distribuída separadamente

Como compilar

- *Use o **java c**ompiler (linha de comando)*
  - `javac NomeDaClasse.java`
  - `javac -d ../destino Um.java Dois.java`
  - `javac -d ../destino *.java`
  - `javac -classpath c:\fontes -d ../destino *.java`
- *Algumas opções (opcionais)*
  - `-d` *diretório onde serão armazenadas as classes (arquivos* `.class`*) geradas*
  - `-classpath` *diretórios (separados por* `;` *ou* `:`*) onde estão as classes requeridas pela aplicação*
  - `-sourcepath` *diretórios onde estão as fontes*
- *Para conhecer outras opções do compilador, digite* `javac` *sem argumentos*
- *Compiladores de outros fabricantes (como o **Jikes**, da IBM) também podem ser usados no lugar do javac*

Como executar

- *Use o interpretador **java** (faz parte do JRE)**
  - `java NomeDaClasse`
  - `java pacote.subpacote.NomeDaClasse`
  - `java -classpath c:\classes;c:\bin;. pacote.Classe`
  - `java -cp c:\classes;c:\bin;. pacote.Classe`
  - `java -cp %CLASSPATH%;c:\mais pacote.Classe`
  - `java -cp biblioteca.jar pacote.Classe`
  - `java -jar executavel.jar`
- *Para rodar aplicações gráficas, use **javaw***
  - `javaw -jar executavel.jar`
  - `javaw -cp aplicacao.jar;gui.jar principal.Inicio`
- *Principais opções*
  - `-cp` *ou*
    `-classpath` *classpath novo (sobrepõe v. ambiente)*
  - `-jar` *executa aplicação executável guardada em JAR*
  - `-D`***`propriedade`***`=`***`valor`*** *define propriedade do sistema (JVM)*

* *sintaxe de PATH em Unix é diferente*

Algumas outras ferramentas do SDK

- Debugger: **jdb**
  - Depurador simples de linha de comando
- Profiler: **java -prof**
  - Opção do interpretador Java que gera estatísticas sobre uso de métodos em um arquivo de texto chamado java.prof
- Java Documentation Generator: **javadoc**
  - Gera documentação em HTML (default) a partir de código-fonte Java
- Java Archiver: **jar**
  - Extensão do formato ZIP; ferramenta comprime, lista e expande
- Applet Viewer: **appletviewer**
  - Permite a visualização de applets sem browser
- HTML Converter: **htmlconverter.jar**
  - Converte <applet> em <object> em páginas que usam applets
- Disassembler: **javap**
  - Permite ler a interface pública de classes

Parte 2: Introdução Prática

Parte 2: Introdução prática

- Nesta seção serão apresentados alguns exemplos de aplicações simples em Java
  1. Aplicação HelloWorld
  2. Aplicação HelloWorld modificada para promover reuso e design orientado a objetos (duas classes)
  3. Aplicação Gráfica Swing (três classes)
  4. Aplicação para cliente Web (applet)
- Compile código-fonte no CD
  - cap01/src/
- Todos os assuntos apresentados nesta seção serão explorados em detalhes em aulas posteriores
  - Conceitos como classe, objeto, pacote
  - Representação UML
  - Sintaxe, classes da API, etc.

1. Aplicação HelloWorld

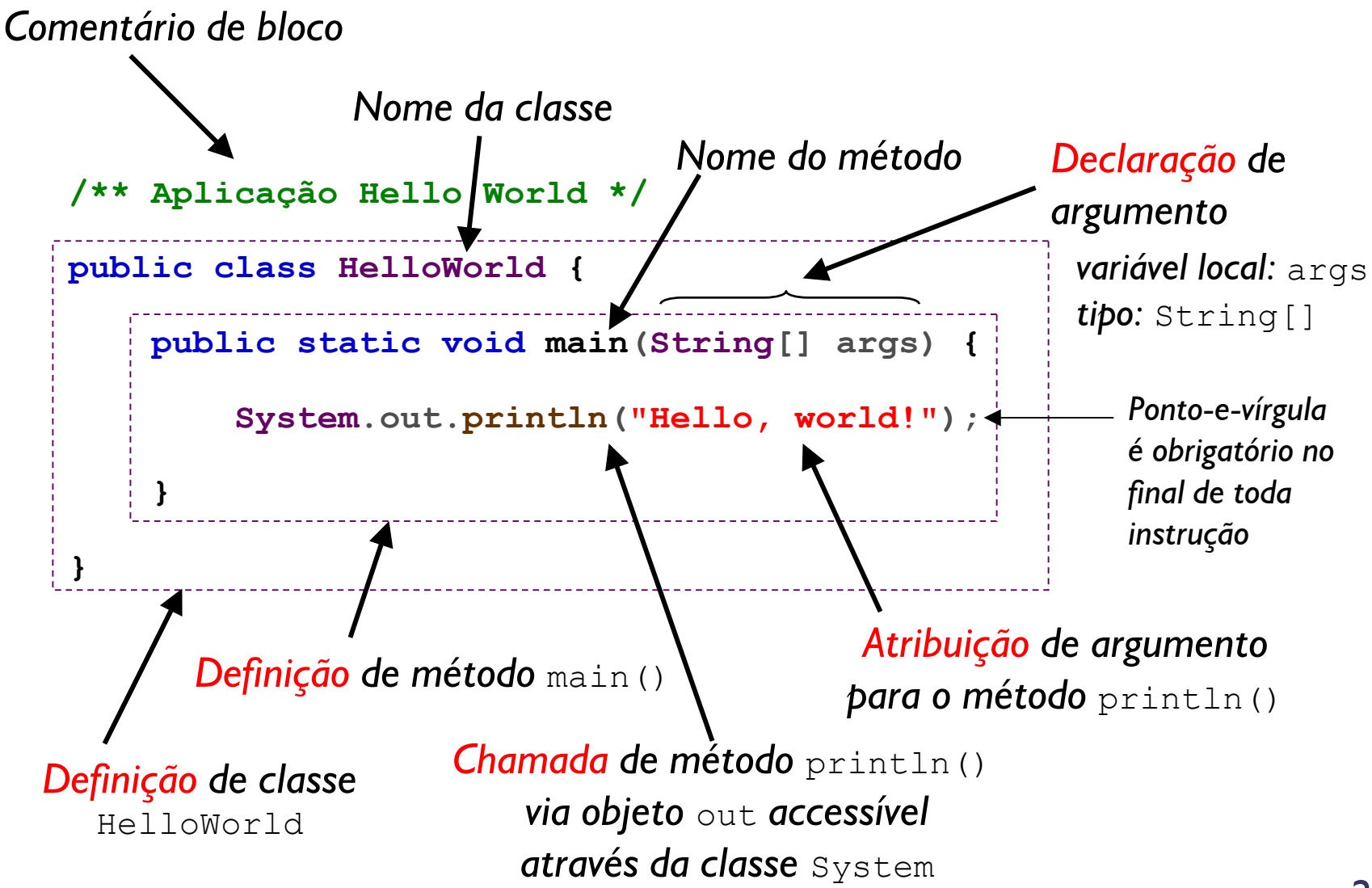
- *Esta mini-aplicação em Java imprime um texto na tela quando executada via linha de comando*

```
/** Aplicação Hello World */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

HelloWorld.java

- *Exercício: Use-a para testar seu ambiente e familiarizar-se com o desenvolvimento Java*
  - *Digite-a no seu editor de textos*
  - *Tente compilá-la*
  - *Corrija eventuais erros*
  - *Execute a aplicação*

Anatomia


Comentário de bloco
Nome da classe
Nome do método
Declaração de argumento
variável local: args
tipo: String[]
/** Aplicação Hello World */
public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello, world!");
}
}
Ponto-e-vírgula é obrigatório no final de toda instrução
Definição de método main()
Atribuição de argumento para o método println()
Definição de classe HelloWorld
Chamada de método println() via objeto out accessível através da classe System

2. Uma classe define um **tipo de dados**

- *Esta classe representa objetos que guardam um texto (tipo String) em um atributo (**msg**) publicamente acessível.*
- *Além de guardar um String, retorna o texto em caixa-alta através do método **lerNome()**.*

Definição da classe (tipo) Mensagem em Java

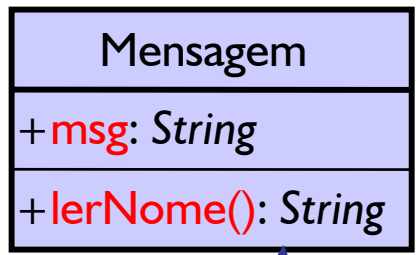
```
public class Mensagem {

    public String msg = "";

    public String lerNome() {
        String nomeEmMaiusculas =
                        msg.toUpperCase();
        return nomeEmMaiusculas;
    }
}
```

atributo

método

Membros da classe. Outras classes podem acessá-los, se declarados como "public", usando o operador ponto "."

Representação em UML



*Esta é a **interface pública** da classe. É só isto que interessa a quem vai usá-la. Os detalhes (código) estão **encapsulados**.*

Classe executável que **usa** um tipo

- *Esta outra classe **usa** a classe anterior para criar um objeto e acessar seus membros visíveis por sua interface pública*
  - *Pode alterar ou ler o valor do atributo de dados* **`msg`**
  - *Pode chamar o método* **`lerNome()`** *e usar o valor retornado*

```
public class HelloJava {
  private static Mensagem nome;

  public static void main(String[] args) {
    nome = new Mensagem();    // cria objeto

    if (args.length > 0) {    // há args de linha de comando?
       nome.msg = args[0];    // se houver, copie para msg
    } else {
       nome.msg = "Usuario"; // copie palavra "Usuario"
    }

    String texto = nome.lerNome(); // chama lerNome()
    System.out.println("Bem-vindo ao mundo Java, "+texto+"!");
  }
}
```

atributo **`nome`** *é do tipo Mensagem*

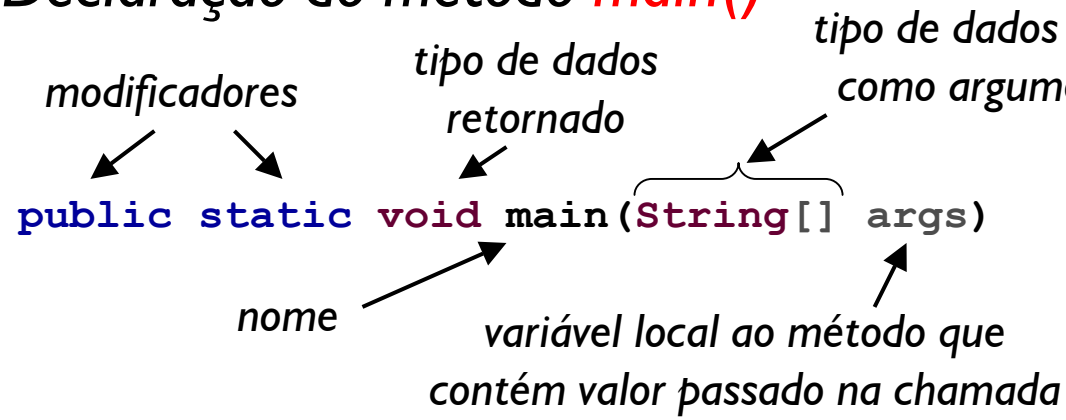
Este método é chamado pelo interpretador

Operador de concatenação

Veja demonstração

Detalhes

- *Declaração do método main()*

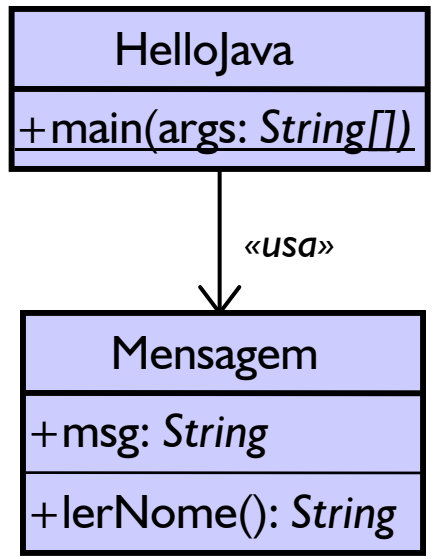


- *O método main() é chamado pelo interpretador Java, automaticamente*
  - *Deve ter sempre a assinatura acima*
- *O argumento é um vetor formado por textos passados na linha de comando:*

```
> java NomeDaClasse Um "Dois Tres" Quatro
```

args[0] *args[1]* *args[2]*

Dependência entre as duas classes (HelloJava tem referência para Mensagem)



3. Primeira aplicação gráfica

- *A aplicação abaixo cria um objeto do tipo JFrame (da API Swing) e reutiliza a classe Mensagem*

```
import javax.swing.*; // importa JFrame e JLabel
import java.awt.Container;

public class MensagemGUI {
  public MensagemGUI(String texto) {
    JFrame janela = new JFrame("Janela");
    Container areaUtil = janela.getContentPane();
    areaUtil.add( new JLabel(texto) );
    janela.pack();
    janela.setVisible(true);
  }
}
```

Quando objeto é criado, construtor MensagemGUI é chamado.

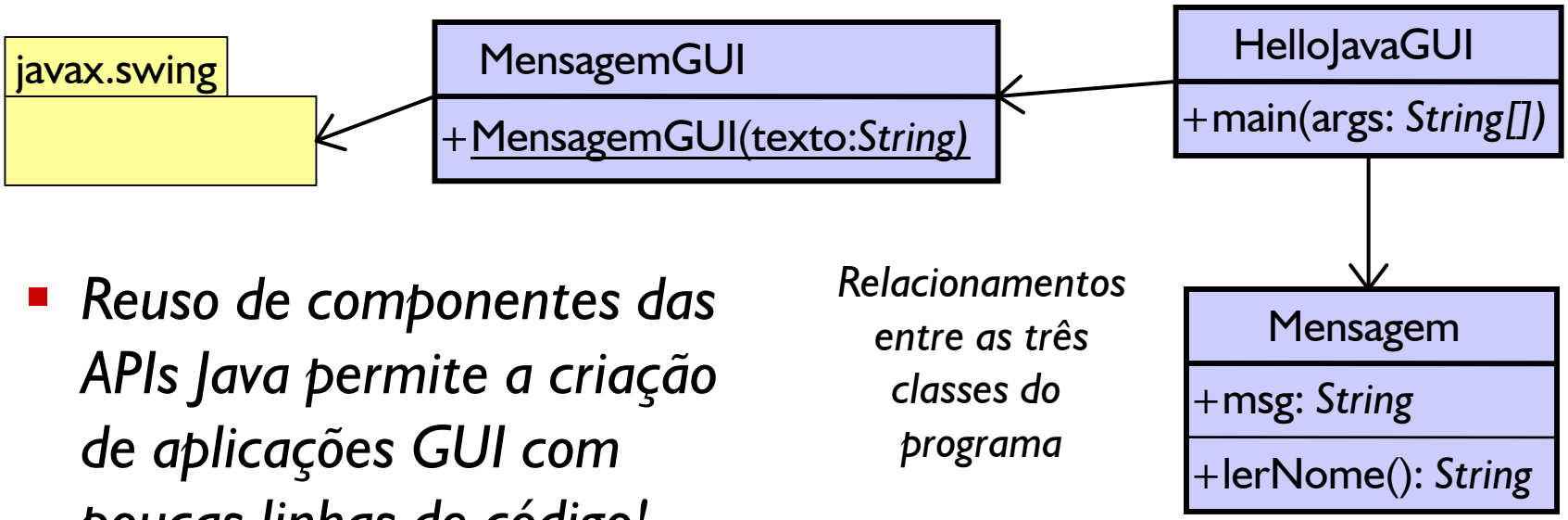
Construtor cria janela contendo texto recebido

```
public class HelloJavaGUI {
  private static Mensagem nome;
  public static void main(String[] args) {
    (... igual a HelloJava ...)
    String texto = nome.lerNome();
    new MensagemGUI
        ("Bem-vindo ao mundo Java, "+texto+"!");
  }
}
```

reuso!

No lugar de imprimir o texto, passa-o como parâmetro na criação de MensagemGUI

Componentes da aplicação gráfica



- *Reuso de componentes das APIs Java permite a criação de aplicações GUI com poucas linhas de código!*

Relacionamentos entre as três classes do programa

Execução da apliação passando parâmetro via linha de comando

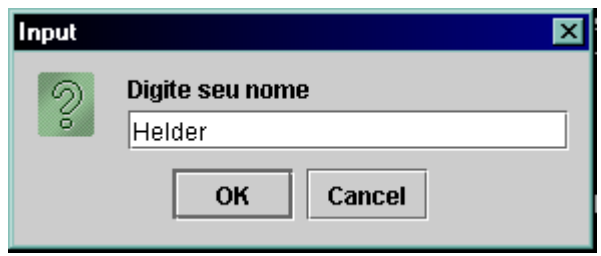
Entrada de dados via GUI

- *javax.swing.JOptionPane oferece uma interface gráfica para entrada de dados e exibição de informações*
- *Exemplo de exibição de caixa de diálogo*
  - `JOptionPane.showMessageDialog(null, "Hello, World!");`

É preciso importar javax.swing. ou javax.swing.JOptionPane*

- *Exemplo de diálogo para entrada de dados*

```
String nome =
    JOptionPane.showInputDialog("Digite seu nome");
if (nome != null) {
    JOptionPane.showMessageDialog(null, nome);
} else {
    System.exit(0);
}
```



4. Primeiro applet

- *Componentes gráficos que podem ser executados no browser*
- *Para criar e usar um applet é preciso*
  - *criar uma classe que herde da classe Applet ou JApplet (API Java)*
  - *criar uma página HTML que carregue o applet*
- *A classe abaixo implementa um JApplet*

```
import javax.swing.*; // importa JFrame e JLabel
import java.awt.Container;

public class HelloJavaApplet extends JApplet {
  private Mensagem nome;

  public void init() {
    nome = new Mensagem();
    nome.msg = this.getParameter("texto"); // parâmetro HTML
    String texto = nome.lerNome();
    Container areaUtil = this.getContentPane();
    JLabel label =
       new JLabel("Bem-vindo ao mundo Java, " +texto+ "!");
    areaUtil.add(label);
  }
}
```

Herança!

Chamado automaticamente pelo browser

Página HTML

- O elemento <applet> é usado para incluir applets antigos (Java 1.0 e 1.1) em páginas HTML ou servir de template para a geração de código HTML 4.0
- A seguinte página carrega o applet da página anterior

```
<html>
 <head>
  <title>Sem Título</title>
 </head>
 <body>
  <h1>Um Applet</h1>
  <applet code="HelloJavaApplet.class" height="50" width="400">
    <param name="texto" value="Helder">
  </applet>
 </body>
</html>
```

- Converta o código para HTML 4.0: ferramenta htmlconverter
  - Guarde uma cópia do original, e rode (use htmlconverter.bat)

```
> htmlconverter pagina.html
```

Detalhes

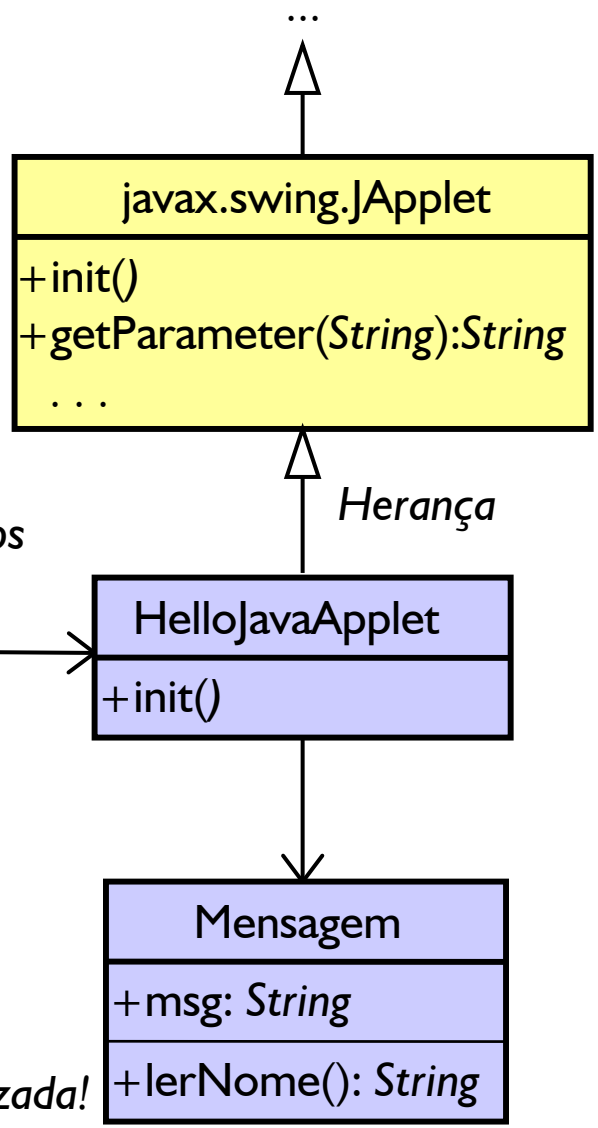
- *Para rodar o applet abra a página com seu browser ou use o appletviewer*

```
> appletviewer pagina.html
```

- *Mude o valor dos parâmetros do HTML e veja os resultados*

Browser oferece container para rodar o Applet

Relacionamentos



Resumo

- *Foram apresentados quatro exemplos de pequenas aplicações Java, demonstrando*
  - *Sintaxe elementar, compilação e execução*
  - *Classe como unidade de código para execução*
  - *Classe como definição de tipo de dados*
  - *Reuso de objetos através de associação (uso da classe Mensagem em três aplicações diferentes) e herança (infraestrutura de Applets reaproveitada)*
  - *Aplicações gráficas*
  - *Componentes de um framework (Applets) que executam em um container padrão (dentro do browser)*

Apêndice: como lidar com erros

Erros (I)

- Durante o desenvolvimento, erros podem ocorrer em dois domínios: tempo de compilação e tempo de execução
- Erros ocorridos durante a fase de compilação ocorrem quando se executa o **javac**, e são fáceis de corrigir. Há dois tipos:
  - Erros de processamento do arquivo (parsing): ponto-e-vírgula faltando, parênteses, aspas, chaves ou colchetes descasados. Identifica apenas o arquivo e a linha onde o erro pode ter iniciado. Um erro causa vários outros e nem sempre a mensagem é precisa.
  - Erros de compilação do código, realizada depois do parsing: além da linha e do arquivo, identificam a classe e método. Geralmente as mensagens são bastante elucidativas.
- É essencial aprender a identificar a causa da mensagem de erro
  - **LEIA** a mensagem e localize a linha onde o erro foi detectado
  - Corrija os erros na ordem em que eles aparecerem
  - Sempre recompile depois de corrigir cada erro de parsing (ponto-e vírgula, etc.) já que eles causam mensagens de erro falsas.

Alguns erros de compilação comuns

- *Cannot resolve symbol*: compilador é incapaz de localizar uma definição do símbolo encontrado. Causas comuns:
  - *Erro de sintaxe no nome de variável ou método*
  - *Variável/método não declarado*
  - *Classe usada não possui variável, método ou construtor*
  - *Número ou tipo de argumentos do método ou construtor incorretos*
  - *Definição de classe não encontrada no CLASSPATH*
- *Class Hello is public, should be declared in a file named Hello.java*: nome do arquivo tem que ser igual ao nome da classe pública*:
  - *Nome tem que ser Hello.java, literalmente. O nome hello.java causa este erro porque o "h" está minúsculo.*
  - *Para consertar altere o nome da classe no código ou no nome do arquivo para que sejam iguais.*

* Se classe não for pública, essa restrição não vale

Exemplos de erros de compilação

- *Erro de parsing*
  - *Na verdade, só há um erro no código, apesar do compilador acusar três*

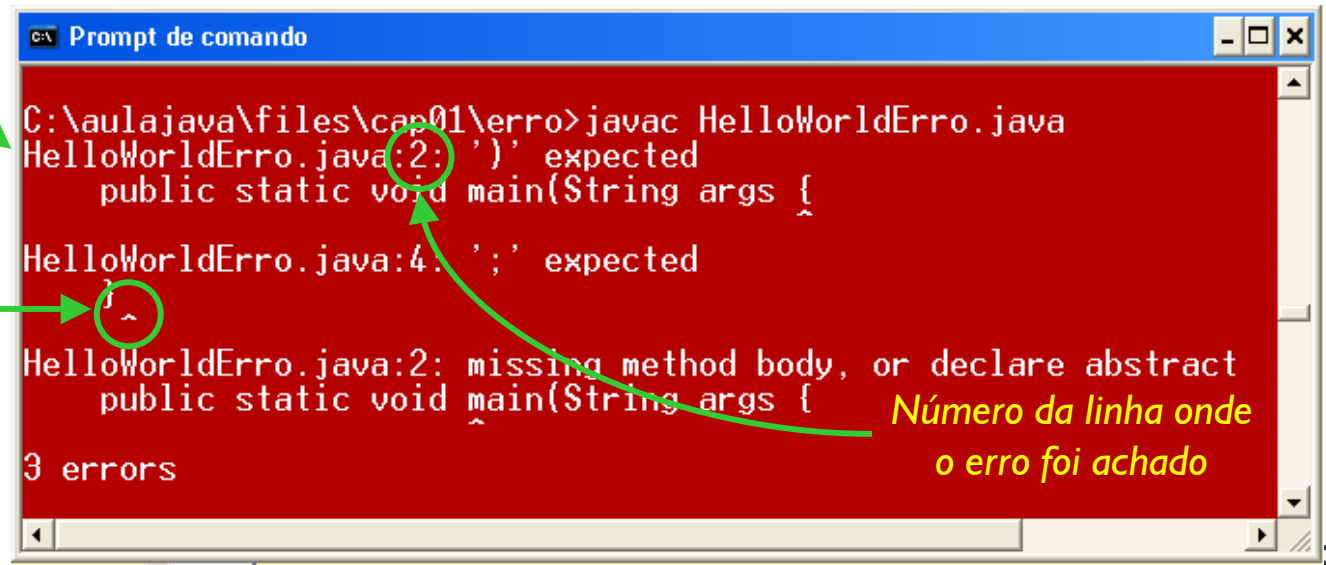
Apenas o primeiro erro é verdadeiro.

Ignore os outros. Eles foram causados pelo primeiro.

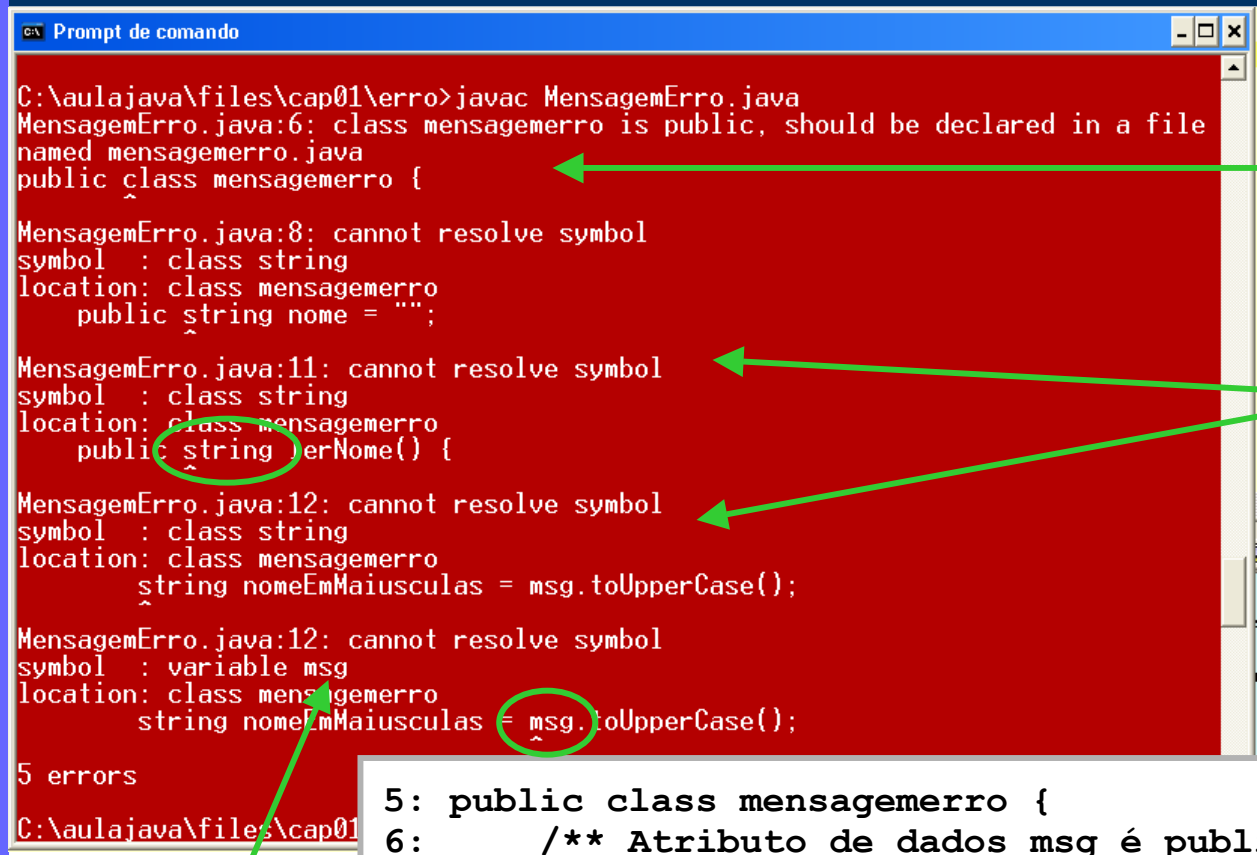
```
1: public class HelloWorldErro {
2:     public static void main(String args {
3:         System.out.println("Hello, 4: world!");
5:     }
6: }
```

Arquivo onde foi detectado o erro

Trecho do código e indicação da provável localização da causa do erro



Exemplos de erros de compilação (2)



Nome do arquivo é MensagemErro.java mas classe foi criada com nome mensagemerro.java

Compilador não sabe quem é string: O tipo String sempre tem um S maiúsculo (como todas as classes da API)

Compilador não sabe quem é msg: não foi declarada nenhuma variável com esse nome.

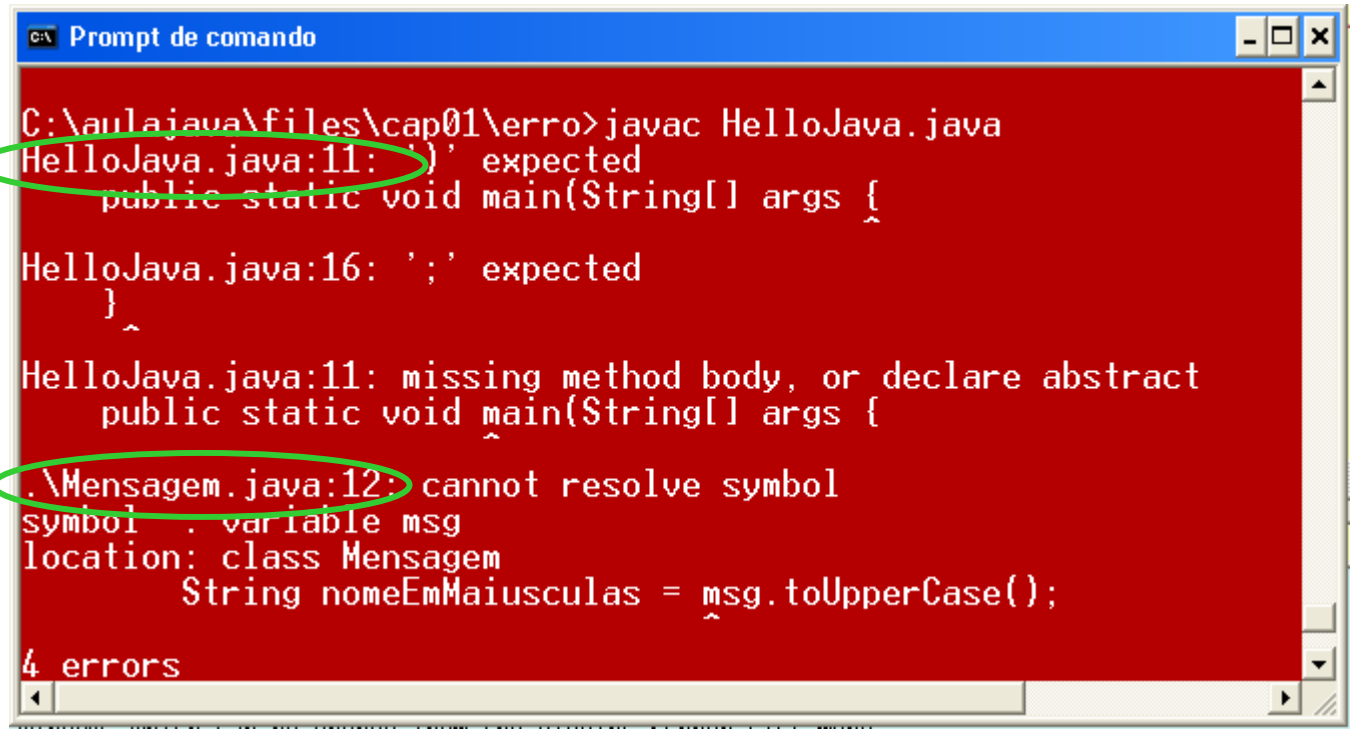
```
5: public class mensagemerro {
6:      /** Atributo de dados msg é publicamente visível */
7:      public string nome = "";
8:
9:      /** Método lerNome() devolve objeto do tipo String */
10:     public string lerNome() {
11:         string nomeEmMaiusculas = msg.toUpperCase();
12:         return nomeEmMaiusculas;
13:     }
14:}
```

Exemplos de erros de compilação (3)

- *Erros em múltiplas classes*
  - *Quando uma classe que possui dependências é compilada, suas dependências são compiladas primeiro e o compilador mostra mensagens de erros referentes a todas as classes envolvidas*
  - *Identifique sempre o arquivo e o número da linha*
  - *Compile as dependências primeiro*

Erro na linha 11 do arquivo HelloJava.java

Erro na linha 12 do arquivo Mensagem.java localizado no mesmo diretório que HelloJava.java



Erros (II)

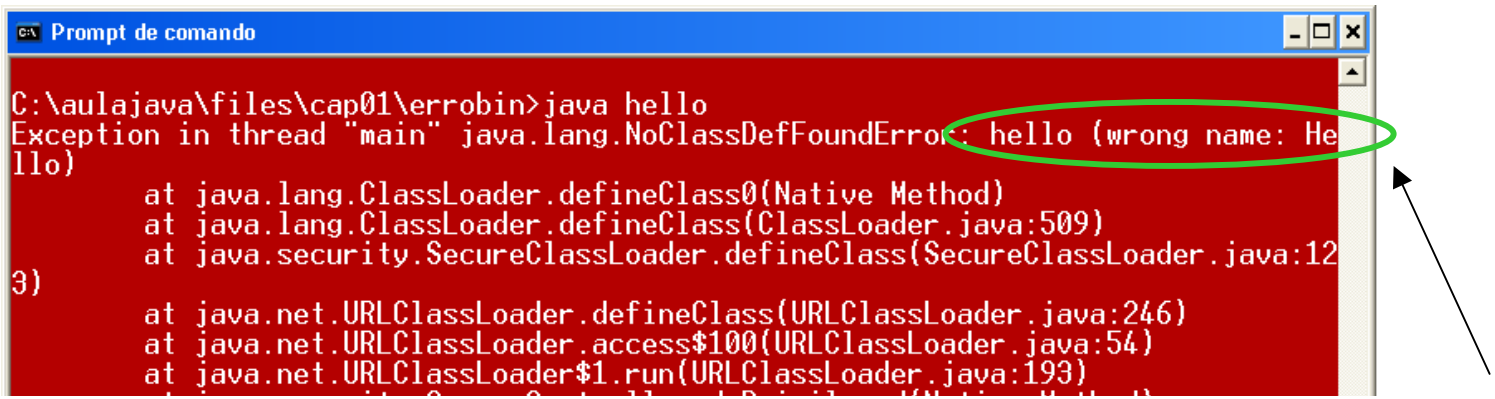
- Depois que o código compila com sucesso, os bytecodes (arquivos .class) são gerados e podem ser usados em um processo de execução
- Erros ocorridos durante a fase de execução (runtime) ocorrem quando se executa o interpretador ***java***, e são muito mais difíceis de localizar e consertar.
- A mensagem impressa geralmente é um "stack trace" e mostra todo o "caminho" percorrido pelo erro
  - Relaciona métodos e classes da sua aplicação e classes da API Java que sua aplicação usa (direta ou indiretamente)
  - Nem sempre mostra a linha de código onde o erro começou
  - O início do trace geralmente contém informações mais úteis
- Erros de runtime nem sempre indicam falhas no software
  - Freqüêntemente se devem a causas externas: não existência de arquivos externos, falta de memória, falha em comunicação de rede

Erros de execução comuns e possíveis causas

- *Exception in thread "main": NoClassDefFoundError: Classe*: *a classe "Classe" não foi encontrada no CLASSPATH.*
  - *O CLASSPATH não inclui todos os diretórios requeridos*
  - *O nome da classe foi digitado incorretamente ou requer pacote*
- *Exception in thread "main": NoSuchMethodError: main*: *o sistema tentou chamar main() mas não o encontrou.*
  - *A classe não tem método main() (talvez não seja executável)*
  - *Confira assinatura do main:* `public static void main(String[])`
- *ArrayIndexOutOfBoundsException*: *programa tentou acessar vetor além dos limites definidos.*
  - *Erro de lógica com vetores*
  - *Aplicação pode requerer argumentos de linha de comando*
- *NullPointerException*: *referência para objeto é nula*
  - *Variável de tipo objeto foi declarada mas não inicializada*
  - *Vetor foi declarado mas não inicializado*

Exemplos de erros de tempo de execução

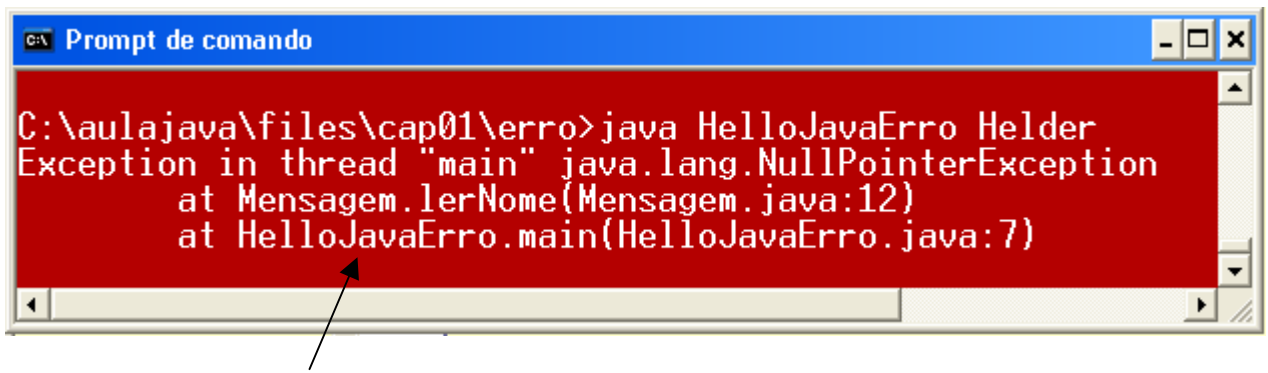
1. Na linha 13, está havendo uma referência a um índice de um vetor. Esse índice tenta acessar uma posição inexistente no vetor. Neste caso específico, o erro pode ser evitado passando-se um parâmetro após o nome da classe na linha de comando, porém a aplicação é pouco robusta pois não prevê que o erro possa acontecer.



2. A classe não foi encontrada. Pode ser que ela não esteja no CLASSPATH. Neste caso específico o interpretador nos sugere que o nome pode está errado. Para consertar, basta chamar a classe pelo nome correto: Hello. O Stack Trace mostra que esse erro teve origem em outras classes, mas foi nossa classe que, na verdade, o provocou.

Exemplos de erros de tempo de execução (2)

- Erros de tempo de execução freqüentemente ocorrem em dependências
  - Causa pode estar na dependência
  - Causa pode ter tido origem na dependência mas ter sido iniciada por erro na classe principal
- Stack Trace pode ajudar a localizar a origem do erro
  - As informações também podem desviar a atenção



Erro começou na linha 7 (método main) de HelloJavaErro mas teve origem na linha 12 do método lerNome de Mensagem

Como achar erros de tempo de execução

- Há dois tipos de erros de tempo de execução
  - Causados por situações externas, que fogem do controle do programador (ex: rede fora do ar)
  - Causados por erros de lógica de programação
- Devemos criar aplicações robustas que prevejam a possibilidade de erros de tempo de execução devido a fatores externos e ajam da melhor forma possível
- Devemos achar erros de lógica e evitar que sobrevivam além da fase de desenvolvimento. Para evitá-los:
  - Escreva código claro, fácil de entender, organizado, pequeno
  - Use endentação, siga convenções, nomes significativos, documente
  - Escreva testes para todo código e rode-os com freqüência
- Para achar os erros difíceis
  - Rode código de testes se os tiver; Ative nível de mensagens de log
  - Aprenda a usar um depurador para navegar no fluxo de execução

Exercícios

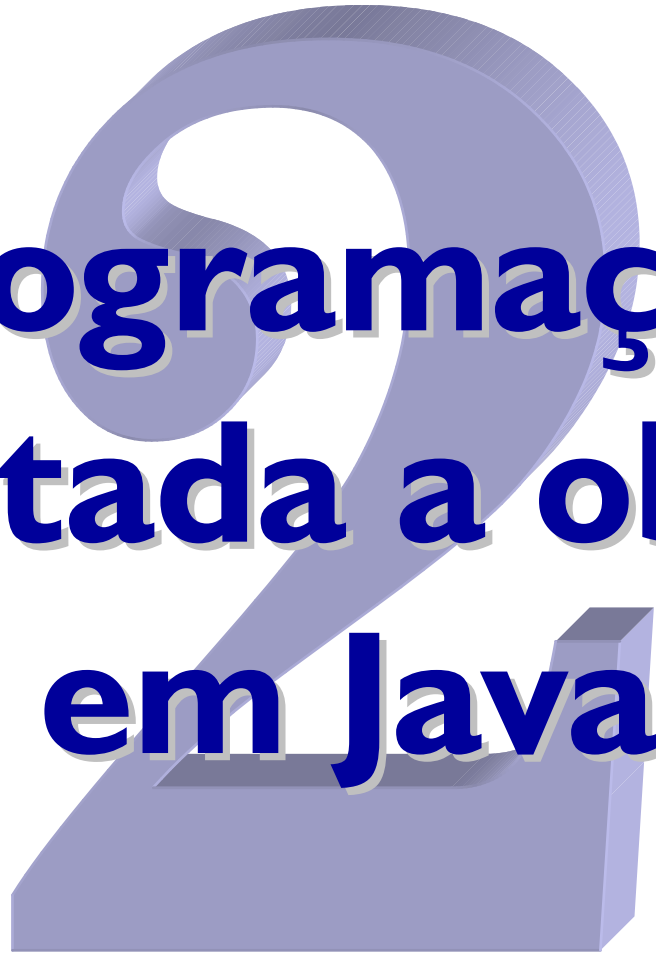
- *1. Compile e execute os exemplos localizados no subdiretório cap01/*
- *2. Há vários arquivos no diretório cap01/exercicios/erro. Todos apresentam erros de compilação. Corrija os erros.*
- *3. Execute os arquivos executáveis do diretório errobin (quais são?). Alguns irão provocar erros de tempo de execução. Corrija-os ou descubra como executar a aplicação sem que eles ocorram.*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br



Programação orientada a objetos em Java

Helder da Rocha
www.argonavis.com.br

Assuntos abordados neste módulo

- Conceitos de programação orientada a objetos existentes na sintaxe da linguagem Java
  - Artefatos: pacote, classe, objeto, membro, atributo, método, construtor e interface
  - Características OO em Java: abstração, encapsulamento, herança e polimorfismo
- Sintaxe Java para construção de estruturas de dados
  - Tipos de dados primitivos
  - Componentes de uma classe
- Construção de aplicações simples em Java
  - Como construir uma classe Java (um tipo de dados) contendo métodos, atributos e construtores
  - Como construir e usar objetos
- Este módulo é longo e aborda muitos assuntos que serão tratados novamente em módulos posteriores

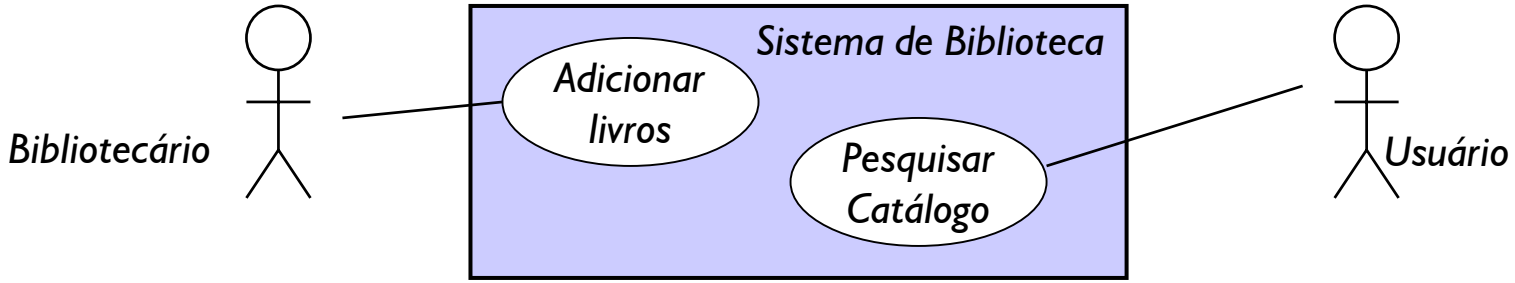
Por que OO é importante?

- *Java é uma linguagem orientada a objetos*
- *Para desenvolver aplicações e componentes de qualidade em Java é preciso entender e saber aplicar princípios de orientação a objetos ao programar*
- *É possível escrever programas em Java sem saber usar os recursos da OO, mas*
  - *Dificilmente você será capaz de ir além de programas simples com mais de uma classe*
  - *Será muito difícil entender outros programas*
  - *Seu código será feio, difícil de depurar e de reutilizar*
  - *Você estará perdendo ao usar uma linguagem como Java*
    (se quiser implementar apenas rotinas procedurais pode usar uma linguagem melhor para a tarefa como Shell, Fortran, etc.)

O que é Orientação a objetos

- *Paradigma do momento na engenharia de software*
  - *Afeta análise, projeto (design) e programação*
- *A **análise** orientada a objetos*
  - *Determina **o que** o **sistema** deve fazer: Quais os atores envolvidos? Quais as atividades a serem realizadas?*
  - *Decompõe o sistema em **objetos**: Quais são? Que tarefas cada objeto terá que fazer?*
- *O **design** orientado a objetos*
  - *Define **como** o sistema será implementado*
  - *Modela os relacionamentos entre os objetos e atores (pode-se usar uma linguagem específica como UML)*
  - *Utiliza e reutiliza abstrações como classes, objetos, funções, frameworks, APIs, padrões de projeto*

Abstração de casos de uso em (1) análise OO e (2) análise procedural

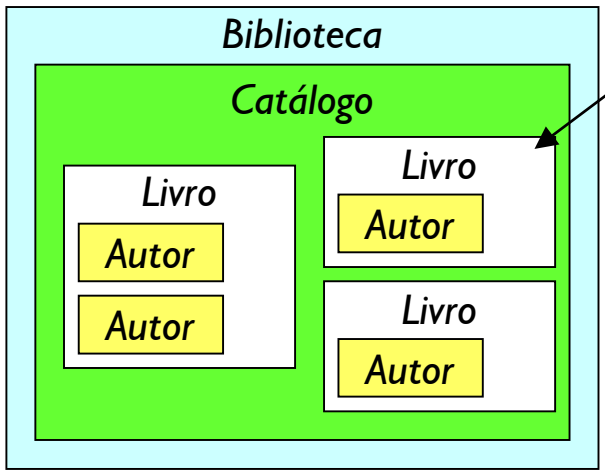


(1) Trabalha no **espaço do problema** (casos de uso simplificados em objetos)

- Abstrações mais simples e mais próximas do mundo real

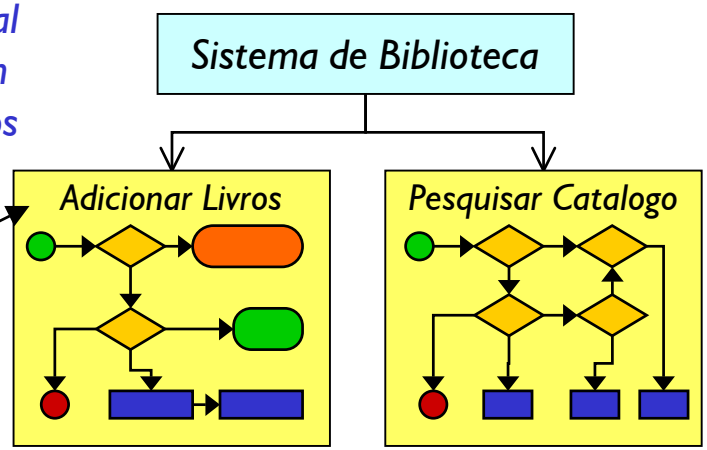
(2) Trabalha no **espaço da solução** (casos de uso decompostos em procedimentos algorítmicos)

- Abstrações mais próximas do mundo do computador



Lógica procedural encapsulada em objetos pequenos

Lógica exposta e espalhada por todo o sistema



O que é um objeto?

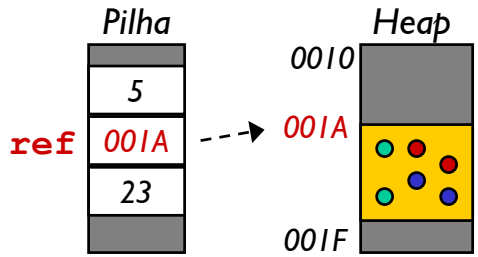
- Objetos são **conceitos** que têm
  - identidade,
  - estado e
  - comportamento
- Características de Smalltalk, resumidas por Allan Kay:
  - Tudo (em um programa OO) são objetos
  - Um programa é um monte de objetos enviando mensagens uns aos outros
  - O espaço (na memória) ocupado por um objeto consiste de outros objetos
  - Todo objeto possui um tipo (que descreve seus dados)
  - Objetos de um determinado tipo podem receber as mesmas mensagens

Ou seja...

- *Em uma linguagem OO pura*
  - *Uma variável é um objeto*
  - *Um programa é um objeto*
  - *Um procedimento é um objeto*
- *Um objeto é composto de objetos, portanto*
  - *Um programa (objeto) pode ter variáveis (objetos que representam seu estado) e procedimentos (objetos que representam seu comportamento)*
- *Analogia: abstração de um telefone celular*
  - *É composto de outros objetos, entre eles bateria e botões*
  - *A bateria é um objeto também, que possui pelo menos um outro objeto: carga, que representa seu estado*
  - *Os botões implementam comportamentos*

Objetos (2)

- Em uma linguagem orientada a objetos pura
  - Um número, uma letra, uma palavra, uma valor booleano, uma data, um registro, uma botão da GUI são objetos
- Em Java, objetos são armazenados na memória de heap e manipulados através de uma referência (variável), guardada na pilha.
  - Têm estado (seus atributos)
  - Têm comportamento (seus métodos)
  - Têm identidade (a referência)



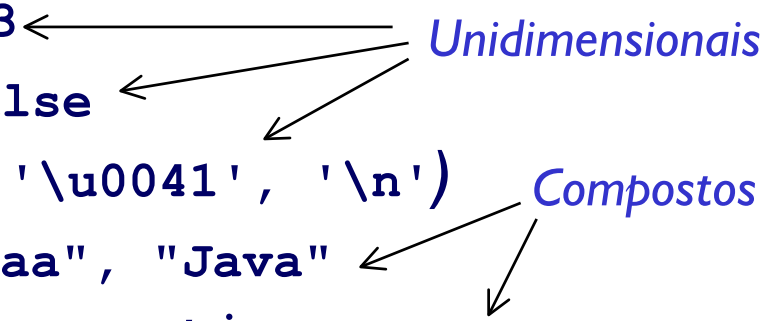
- Valores unidimensionais não são objetos em Java
  - Números, booleanos, caracteres são armazenados na pilha
  - Têm apenas identidade (nome da variável) e estado (valor literal armazenado na variável); - dinâmicos; + rápidos

Variáveis, valores e referências

- Variáveis são usadas em linguagens em geral para armazenar valores
- Em Java, variáveis podem armazenar endereços de memória do heap ou valores atômicos de tamanho fixo
  - Endereços de memória (referências) são inaccessíveis aos programadores (Java não suporta aritmética de ponteiros!)
  - Valores atômicos representam tipos de dados primitivos
- Valores são passados para variáveis através de operações de atribuição
  - Atribuição de valores é feita através de literais
  - Atribuição de referências (endereços para valores) é feita através de operações de construção de objetos e, em dois casos, pode ser feita através de literais

Literais e tipos

- *Tipos* representam um *valor*, uma *coleção* de valores ou coleção de outros tipos. Podem ser
  - *Tipos básicos, ou primitivos*, quando representam unidades indivisíveis de informação de tamanho fixo
  - *Tipos complexos*, quando representam informações que podem ser decompostas em tipos menores. Os tipos menores podem ser primitivos ou outros tipos complexos
- *Literais*: são *valores* representáveis literalmente.
  - Números: `1, 3.14, 1.6e-23`
  - Valores booleanos: `true` e `false`
  - Caracteres individuais: `'a', '\u0041', '\n'`)
  - Seqüências de caracteres: `"aaa", "Java"`
  - Vetores de números, booleanos ou strings: `{"a", "b"}`



Tipos primitivos e complexos

- Exemplos de tipos primitivos (atômicos)
  - Um inteiro ou um caractere,
  - Um literal booleano (true ou false)
- Exemplos de tipos complexos
  - Uma data: pode ser decomposta em três inteiros, representando dia, mês e ano
  - Um vetor de inteiros: pode ser decomposto em suas partes
  - Uma seqüência de caracteres: pode ser decomposta nos caracteres que a formam
- Em Java, tipos complexos são armazenados como objetos e tipos primitivos são guardados na pilha
  - Apesar de serem objetos, seqüências de caracteres (strings) em Java podem ser representadas literalmente.

Tipos primitivos em Java

- Têm tamanho fixo. Têm sempre valor default.
- Armazenados na pilha (acesso rápido)
- Não são objetos. Classe 'wrapper ' faz transformação, se necessário (encapsula valor em um objeto).

Tipo	Tamanho	Mínimo	Máximo	Default	'Wrapper'
boolean	—	—	—	false	Boolean
char	16-bit	Unicode 0	Unicode 2^{16} - 1	\u0000	Character
byte	8-bit	-128	+127	0	Byte
short	16-bit	-2^{15}	$+2^{15}$ —1	0	Short
int	32-bit	-2^{31}	$+2^{31}$ —1	0	Integer
long	64-bit	-2^{63}	$+2^{63}$ —1	0	Long
float	32-bit	IEEE754	IEEE754	0.0	Float
double	64-bit	IEEE754	IEEE754	0.0	Double
void	—	—	—	—	Void

Exemplos de tipos primitivos e literais

- *Literais de caracter:*

```
char c = 'a';
char z = '\u0041'; // em Unicode
```

- *Literais inteiros*

```
int i = 10;  short s = 15;  byte b = 1;
long hexa = 0x9af0L; int octal = 0633;
```

- *Literais de ponto-flutuante*

```
float f = 123.0f;
double d = 12.3;
double g = .1e-23;
```

- *Literais booleanos*

```
boolean v = true;
boolean f = false;
```

- *Literais de string (não é tipo primitivo - s é uma referência)*

```
String s = "abcde";
```

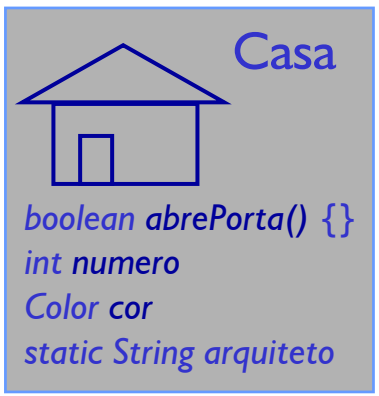
- *Literais de vetor (não é tipo primitivo - v é uma referência)*

```
int[] v = {5, 6};
```

O que é uma classe?

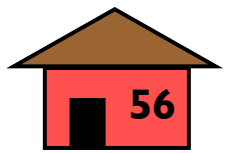
- Classes são uma especificação para objetos
- Uma classe representa um tipo de dados complexo
- Classes descrevem
  - Tipos dos dados que compõem o objeto (o que podem armazenar)
  - Procedimentos que o objeto pode executar (o que podem fazer)

Classe



Instâncias da classe Casa (objetos)


```
Casa c1 = new Casa();
c1.numero = 12;
c1.cor = Color.yellow;
```




```
Casa c2 = new Casa();
c2.numero = 56;
c2.cor = Color.red;
```


```
Casa c3 = new Casa();
c3.numero = 72;
c3.cor = Color.white;
c3.abrePorta();
```

Ou seja...

- Classes não são os objetos que representam
  - A planta de uma casa é um objeto, mas não é uma casa
- Classes definem lógica estática
  - Relacionamentos entre classes são definidos na programação e não mudam durante a execução
  - Relacionamentos entre objetos são dinâmicos e podem mudar. O funcionamento da aplicação reflete a lógica de relacionamento entre os objetos, e não entre as classes.
- Classes não existem no contexto da execução
  - Uma classe representa vários objetos que ocupam espaço na memória, mas ela não existe nesse domínio
  - A classe tem papel na criação dos objetos, mas não existe quando os objetos trocam mensagens entre si.

Resumo

- *Objetos são conceitos que têm estado (atributos), comportamento (métodos) e identidade (referência)*
- *Tipos representam valores*
  - *Primitivos: valores fixos e indivisíveis. São armazenados na pilha*
  - *Complexos: valores multidimensionais que podem ser decompostos em componentes menores. Descrevem objetos que são armazenados no heap*
- *Literais*
  - *Usados para definir tipos primitivos ou certos tipos complexos formados por componentes iguais (strings e vetores)*
- *Variáveis podem armazenar valores de tipos primitivos ou referências para objetos*
- *Classes são tipos complexos: descrevem objetos*
  - *Não são importantes no contexto da execução*

Membros: atributos e métodos

- Uma classe define uma estrutura de dados não-ordenada
  - Pode conter componentes em qualquer ordem
- Os componentes de uma classe são seus membros
- Uma classe pode conter três tipos de componentes
  - Membros estáticos ou de classe: não fazem parte do "tipo"
  - Membros de instância: definem o tipo de um objeto
  - Procedimentos de inicialização
- Membros estáticos ou de classe
  - Podem ser usados através da classe mesmo quando não há objetos
  - Não se replicam quando novos objetos são criados
- Membros de instância
  - Cada objeto, quando criado, aloca espaço para eles
  - Só podem ser usados através de objetos
- Procedimentos de inicialização
  - Usados para inicializar objetos ou classes

Exemplo

```
public class Casa {

    private Porta porta;
    private int numero;
    public java.awt.Color cor;

    public Casa() {
        porta = new Porta();
        numero = ++contagem * 10;
    }

    public void abrePorta() {
        porta.abre();
    }

    public static String arquiteto = "Zé";
    private static int contagem = 0;

    static {
        if ( condição ) {
            arquiteto = "Og";
        }
    }
}
```

Tipo

Atributos de instância: cada objeto poderá armazenar valores diferentes nessas variáveis.

Procedimento de inicialização de ***objetos*** *(Construtor)*: código é executado após a criação de cada novo objeto. Cada objeto terá um número diferente.

Método de instância: só é possível chamá-lo se for através de um objeto.

Atributos estáticos: não é preciso criar objetos para usá-los. Todos os objetos os compartilham.

Procedimento de inicialização ***estático***: código é executado uma única vez, quando a classe é carregada. O arquiteto será um só para todas as casas: ou Zé ou Og.

Boas práticas ao escrever classes

- Use, e abuse, dos espaços
  - Endente, com um tab ou 4 espaços, membros da classe,
  - Endente com 2 tabs, o conteúdo dos membros, ...
- A ordem dos membros não é importante, mas seguir convenções melhora a legibilidade do código
  - Mantenha os membros do mesmo tipo juntos (não misture métodos estáticos com métodos de instância)
  - Declare as variáveis antes ou depois dos métodos (não misture métodos, construtores e variáveis)
  - Mantenha os seus construtores juntos, de preferência bem no início
  - Se for necessário definir blocos static, defina apenas um, e coloque-o no início ou no final da classe.

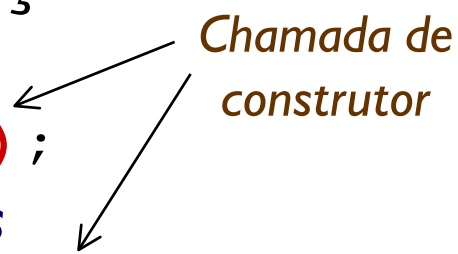
Construtores

- *Construtores são procedimentos realizados na construção de objetos*
  - *Parecem métodos, mas não têm tipo de retorno e têm nome idêntico ao nome da classe*
  - *Não fazem parte da definição do tipo do objeto (interface)*
  - *Nem sempre aparecem explícitos em uma classe: podem ser omitidos (o sistema oferece uma implementação default)*
- *Para cada objeto, o construtor é chamado exatamente uma vez: na sua criação*
  - *Exemplo:*

```
> Objeto obj = new Objeto();
```

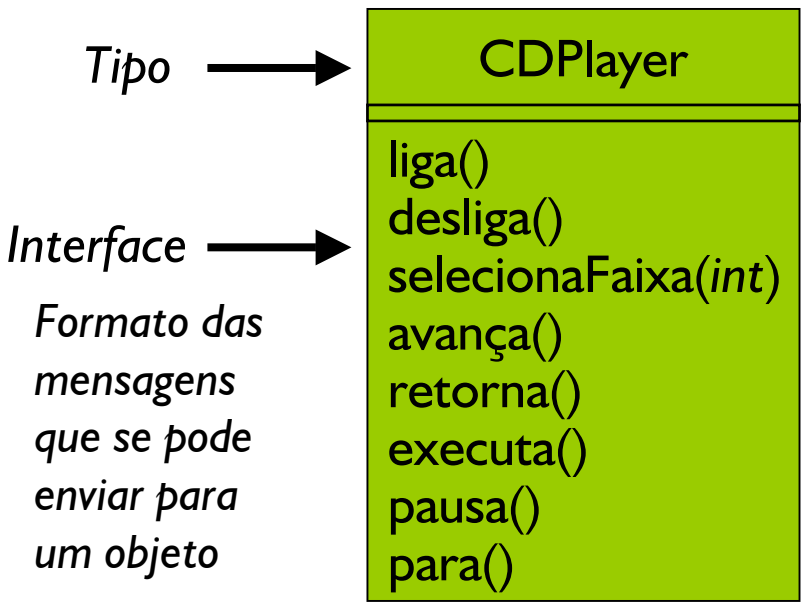
  - *Alguns podem requerer parâmetros*

```
> Objeto obj = new Objeto(35, "Nome");
```



Objetos possuem uma interface ...

- Através da *interface** é possível utilizá-lo
  - Não é preciso saber dos detalhes da implementação
- O tipo (Classe) de um objeto determina sua interface
  - O tipo determina quais mensagens podem ser enviadas



Em Java

```
(...)
CDPlayer cd1;
cd1 = new CDPlayer();
cd1.liga();
cd1.selecionaFaixa(3);
cd1.executa();
(...)
```

Classe Java (tipo)

Referência

Criação de objeto

Envio de mensagem

* interface aqui refere-se a um conceito e não a um tipo de classe Java

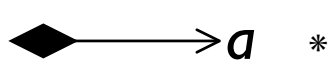
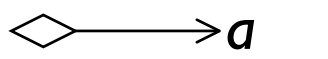
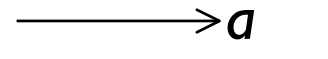
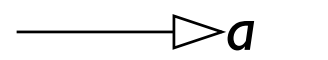
... e uma implementação (oculta)

- Implementação não interessa à quem **usa** objetos
- Papel do usuário de classes
  - não precisa saber como a classe foi escrita, apenas quais seus métodos, quais os parâmetros (quantidade, ordem e tipo) e valores que são retornados
  - usa apenas a interface (pública) da classe
- Papel do desenvolvedor de classes
  - define novos tipos de dados
  - expõe, através de métodos, todas as funções necessárias ao usuário de classes, e oculta o resto da implementação
  - tem a liberdade de mudar a implementação das classes que cria sem que isto comprometa as aplicações desenvolvidas pelo usuário de classes

Resumo

- Os componentes de uma classe, em Java, podem pertencer a dois domínios, que determinam como são usados
  - Domínio da classe: existem independentemente de existirem objetos ou não: métodos static, blocos static, atributos static e interface dos construtores de objetos
  - Domínio do objeto: métodos e atributos não declarados como static (definem o tipo ou interface que um objeto possui), e conteúdo dos construtores
- Construtores são usados apenas para construir objetos
  - Não são métodos (não declaram tipo de retorno)
  - "Ponte" entre dois domínios: são chamados uma vez antes do objeto existir (domínio da classe) e executados no domínio do objeto criado
- Separação de interface e implementação
  - Usuários de classes vêem apenas a interface.
  - Implementação é encapsulada dentro dos métodos, e pode variar sem afetar classes que usam os objetos

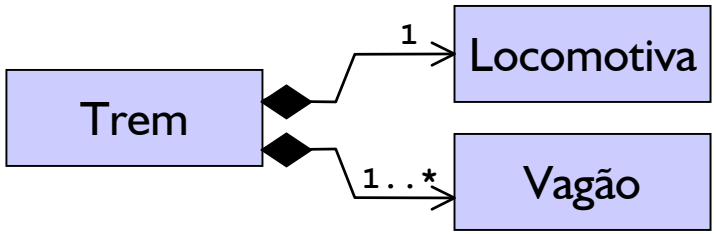
Reuso de implementação

- *Separação interface-implementação: maior reuso*
  - *Reuso depende de bom planejamento e design*
- *Uma vez criada uma classe, ela deve representar uma unidade de código útil para que seja reutilizável*
- *Formas de uso e reuso*
  - *Uso e reuso de **objetos** criados pela classe: mais flexível*
    - *Composição: a "é parte essencial de" b* 
    - *Agregação: a "é parte de" b* 
    - *Associação: a "é usado por" b* 
  - *Reuso da interface da **classe**: pouco flexível*
    - *Herança: b "é" a (substituição pura)* 
      ou b "é um tipo de" a (substituição útil, extensão)

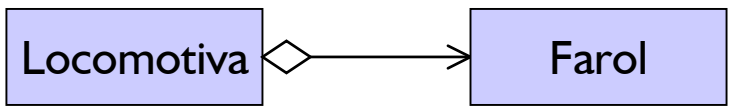
* Notação UML

Agregação, composição e associação

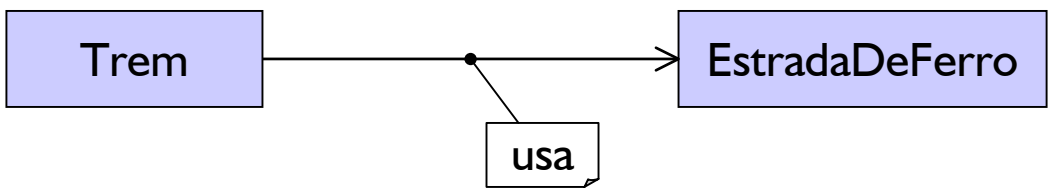
- ***Composição***: *um trem é formado por locomotiva e vagões*



- ***Agregação***: *uma locomotiva tem um farol (mas não vai deixar de ser uma locomotiva se não o tiver)*

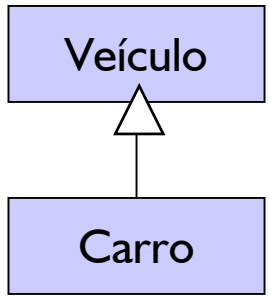


- ***Associação***: *um trem usa uma estrada de ferro (não faz parte do trem, mas ele depende dela)*



Herança (reuso de interface)

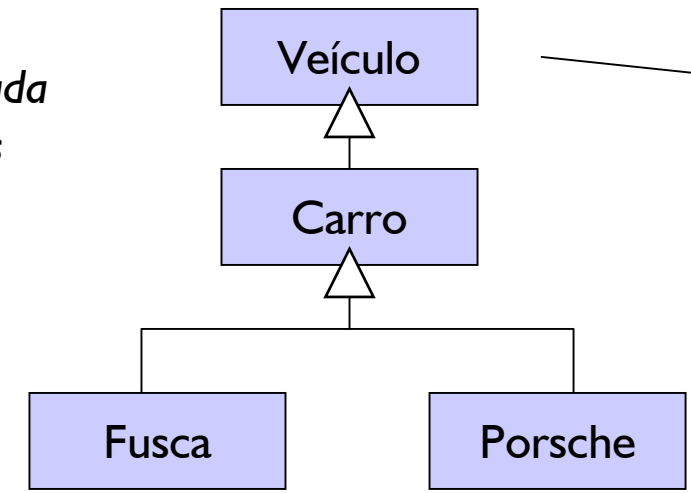
- *Um carro é um veículo*

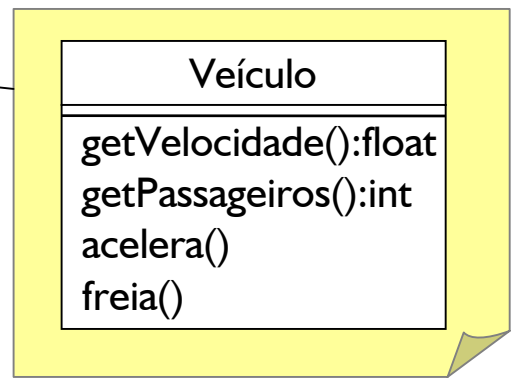


representação UML

- *Fuscas e Porsches são carros (e também veículos)*

representação UML simplificada (não mostra os métodos)



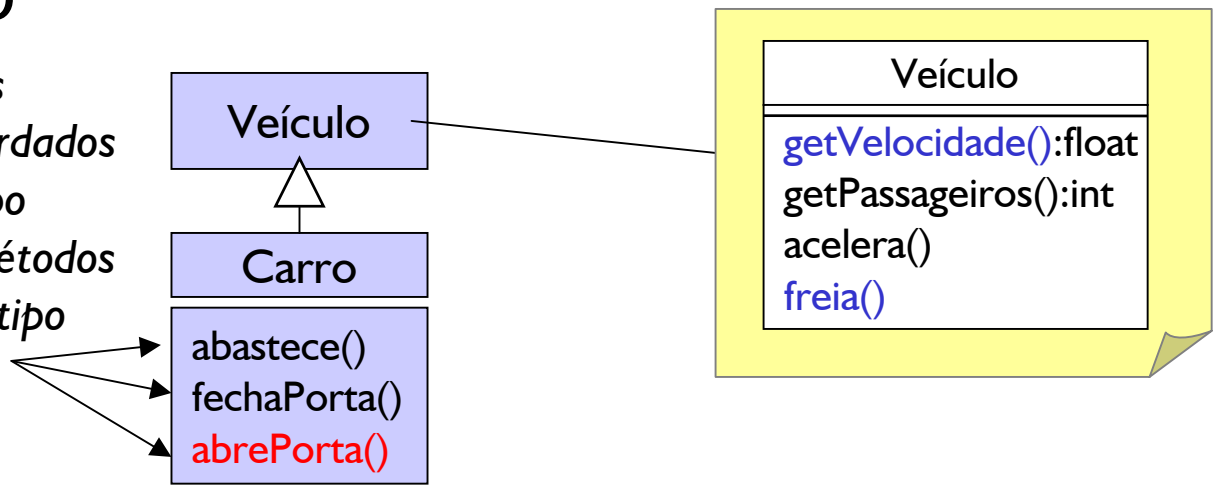


representação UML detalhada de 'Veículo'

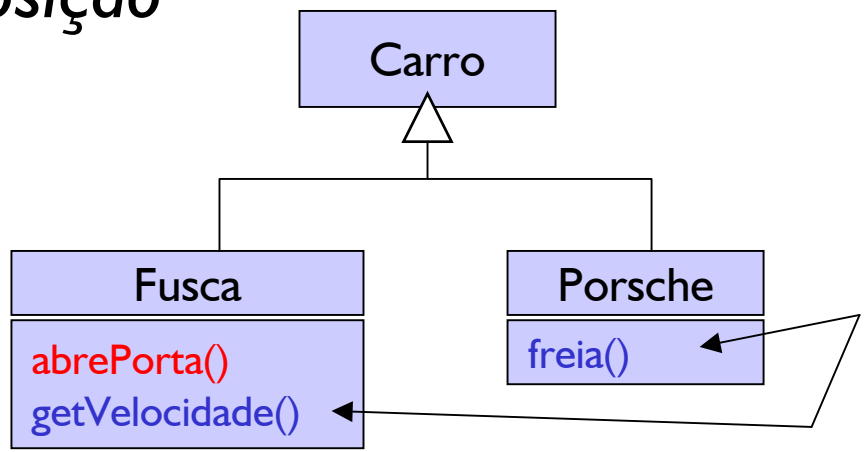
Extensão e sobreposição

- *Extensão*

***Acrescenta** novos métodos aos já herdados (Um **objeto** do tipo Carro tem mais métodos que um objeto do tipo Veiculo)*



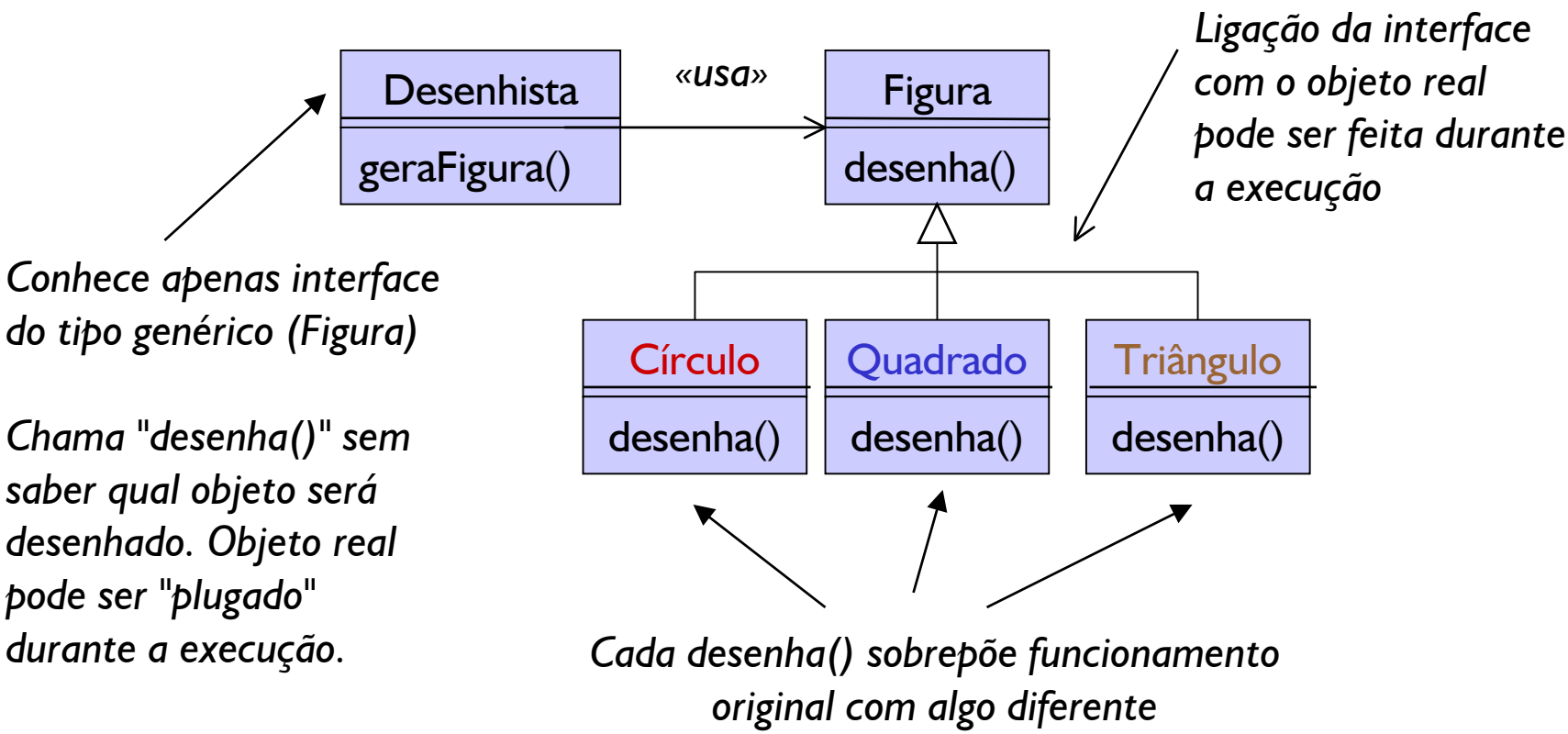
- *Sobreposição*



***Redefine** os métodos implementados previamente (Um **objeto** do tipo Fusca tem o mesmo número de métodos de um objeto do tipo Carro*

Polimorfismo

- *Uso de um objeto no lugar de outro*
  - *pode-se escrever código que não dependa da existência prévia de tipos específicos*



Encapsulamento

- Simplifica o objeto expondo apenas a sua interface essencial
- Código dentro de métodos é naturalmente encapsulado
  - Não é possível acessar interior de um método fora do objeto
- Métodos que não devem ser usados externamente e atributos podem ter seu nível de acesso controlado em Java, através de modificadores
  - **private**: apenas acesso dentro da classe
  - **package-private** (default): acesso dentro do pacote*
  - **protected**: acesso em subclasses
  - **public**: acesso global

---

* não existe um modificador com este nome. A ausência de um modificador de acesso deixa o membro com acesso package-private

Resumo de características OO

- Abstração de conceitos
  - Classes, definem um tipo separando interface de implementação
  - Objetos: instâncias utilizáveis de uma classe
- Herança: "é um"
  - Aproveitamento do código na formação de hierarquias de classes
  - Fixada na compilação (inflexível)
- Associação "tem um"
  - Consiste na delegação de operações a outros objetos
  - Pode ter comportamento e estrutura alterados durante execução
  - Vários níveis de acoplamento: associação, composição, agregação
- Encapsulamento
  - Separação de interface e implementação que permite que usuários de objetos possam utilizá-los sem conhecer detalhes de seu código
- Polimorfismo
  - Permite que objeto seja usado no lugar de outro

Exercício

- 1. Crie, e compile as seguintes classes
  - Uma Pessoa tem um nome (String)
  - Uma Porta tem um estado aberto, que pode ser true ou false, e pode ser aberta ou fechada
  - Uma Construcao tem uma finalidade
  - Uma Casa é uma Construcao com finalidade residencial que tem um proprietário Pessoa, um número e um conjunto (vetor) de Portas
- 2. Crie as seguintes classes
  - Um Ponto tem coordenadas x e y inteiras
  - Um Circulo tem um Ponto e um raio inteiro
  - Um Pixel é um tipo de Ponto que possui uma cor

Menor classe utilizável em Java

- Uma classe contém a representação de um objeto
  - define seus métodos (comportamento)
  - define os tipos de dados que o objeto pode armazenar (estado)
  - determina como o objeto deve ser criado (construtor)
- Uma classe Java também pode conter
  - procedimentos independentes (métodos 'static')
  - variáveis estáticas
  - rotinas de inicialização (blocos 'static')
- O programa abaixo é a menor unidade compilável em Java

```
class Menor {}
```

Símbolos essenciais

- Separadores
  - **{ ... }** chaves: contém as partes de uma classe e delimitam blocos de instruções (em métodos, inicializadores, estruturas de controle, etc.)
  - **;** ponto-e-vírgula: obrigatória no final de toda instrução simples ou declaração
- Identificadores
  - Nomes usados para representar classes, métodos, variáveis (por exemplo: desenha, Casa, abrePorta, Circulo, raio)
  - Podem conter letras (Unicode) e números, mas não podem começar com número
- Palavras reservadas
  - São 52 (assert foi incluída na versão 1.4.0) e consistem de 49 palavras-chave e literais true, false e null.
  - Exemplos de palavras-chave são public, int, class, for e void
  - A maior parte dos editores de código Java destaca as palavras reservadas

Para que serve uma classe

- Uma classe pode ser usada para
  - conter a rotina de execução principal de uma aplicação iniciada pelo sistema operacional (método main)
  - conter funções globais (métodos estáticos)
  - conter constantes e variáveis globais (campos de dados estáticos)
  - especificar e criar objetos (contém construtores, métodos e atributos de dados)

Uma unidade de compilação

Casa.java

```
package cidade; // classe faz parte do pacote cidade

import cidade.ruas.*;    // usa todas as classes de pacote
import pais.terrenos.LoteUrbano; // usa classe LoteUrbano
import Pessoa;           // ilegal desde Java 1.4.0
import java.util.*;      // usa classes de pacote Java

class Garagem {
  ...
}

interface Fachada {
...
}

/** Classe principal */
public class Casa {
...
}
```

Por causa da declaração 'package' o nome completo destas classes é cidade.Garagem cidade.Fachada e cidade.Casa

Este arquivo, ao ser compilado, irá gerar três arquivos .class

O que pode conter uma classe

- Um bloco '*class*' pode conter (entre as chaves `{ . . . }`), em qualquer ordem
  - (1) zero ou mais declarações de **atributos de dados**
  - (2) zero ou mais definições de **métodos**
  - (3) zero ou mais **construtores**
  - (4) zero ou mais **blocos de inicialização static**
  - (5) zero ou mais definições de **classes ou interfaces internas**
- Esses elementos só podem ocorrer **dentro** do bloco '`class NomeDaClasse { . . . }`'
  - tudo, em Java, 'pertence' a alguma classe
  - apenas '*import*' e '*package*' podem ocorrer fora de uma declaração '*class*' (ou '*interface*')

Métodos

- *Contém procedimentos - instruções simples ou compostas executadas em seqüência - entre chaves*
- *Podem conter argumentos*
  - *O tipo de cada argumento precisa ser declarado*
  - *Método é identificado pelo nome + número e tipo de argumentos*
- *Possuem um tipo de retorno ou a palavra void*
- *Podem ter modificadores (public, static, etc.) antes do tipo*

```
...
 public void paint (Graphics g){
   int x = 10;
   g.drawString(x, x*2, "Booo!");
 }
...
```

```
class T1 {
  private int a; private int b;
  public int soma () {
    return a + b;
  }
}
```

```
class T2 {
  int x, y;
  public int soma () {
    return x + y;
  }
  public static int soma(int a, int b){
    return a + b;
  }

  public static int soma(int a,
                         int b, int c){
    return soma(soma(a, b), c);
  }
}
```

Sintaxe de definição de métodos

- *Sintaxe básica*
  - [mod]* *tipo identificador* ([tipo arg]*) [throws exceção*] { ... }
- *Chave*
  - [mod]* – *zero ou mais modificadores separados por espaços*
  - *tipo – tipo de dados retornado pelo método*
  - *identificador – nome do método*
  - [arg]* – *zero ou mais argumentos, com tipo declarado, separados por vírgula*
  - [throws exceção*] – *declaração de exceções*
- *Exemplos*

```
public static void main ( String[] args ) { ... }
private final synchronized
          native int metodo (int i, int j, int k) ;
String abreArquivo ()
        throws IOException, Excecao2 { ... }
```

Atributos de dados

- *Contém dados*
- *Devem ser declaradas com tipo*
- *Podem ser pré-inicializadas (ou não)*
- *Podem conter modificadores*

```
public class Produto {
    public static int total = 0;
    public int serie = 0;
    public Produto() {
        serie = serie + 1;
        total = serie;
    }
}
```

```
class Data {
    int dia;
    int mes;
    int ano;
}
```

```
public class Livro {
    private String titulo;
    private int codigo = 815;
    ...
    public int mostraCodigo() {
        return codigo;
    }
}
```

```
class Casa {
    static Humano arquiteto;
    int numero;
    Humano proprietario;
    Doberman[] guardas;
}
```

Sintaxe de declaração de atributos

- *Sintaxe básica*
  - [modificador]* *tipo identificador* [= valor] ;
- *Chave*
  - [modificador]* – *zero ou mais modificadores (de acesso, de qualidade), separados por espaços: public, private, static, transient, final, etc.*
  - *tipo – tipo de dados que a variável (ou constante) pode conter*
  - *identificador – nome da variável ou constante*
  - [= valor] – *valor inicial da variável ou constante*
- *Exemplo*

```
protected static final double PI = 3.14159 ;
int numero;
```

Construtores

- *Têm sempre o mesmo nome que a classe*
- *Contém procedimentos entre chaves, como os métodos*
- *São chamados apenas uma vez: na criação do objeto*
- *Pode haver vários em uma mesma classe*
  - *São identificados pelo número e tipo de argumentos*
- *Nunca declaram tipo de retorno*

```
public class Produto {
    public static int total = 0;
    public int serie = 0;
    public Produto() {
        serie = total + 1;
        total = serie;
    }
}
```

```
public class Livro {
    private String titulo;
    public Livro() {
        titulo = "Sem título";
    }
    public Livro(String umTitulo) {
        titulo = umTitulo;
    }
}
```

Sintaxe de construtores

- Construtores são procedimentos especiais usados para construir novos objetos a partir de uma classe
  - A definição de construtores é opcional: Toda classe sem construtor declarado explicitamente possui um construtor fornecido pelo sistema (sem argumentos)
- Parecem métodos mas
  - não definem tipo de retorno
  - possuem, como identificador, o nome da classe: Uma classe pode ter vários construtores, com o mesmo nome, que se distinguem pelo número e tipo de argumentos
- Sintaxe
  - [mod]* nome_classe ([tipo arg]*) [throws exceção*] { ... }

Exemplo

- *Exemplo de classe com um atributo de dados (variável), um construtor e dois métodos*

```
public class UmaClasse {

      private String mensagem;

      public UmaClasse () {
            mensagem = "Mensagem inicial";
      }

      public void setMensagem (String m) {
            mensagem = m;
      }

      public String getMensagem() {
            return mensagem;
      }
}
```

variavel (referencia) do tipo String

construtor

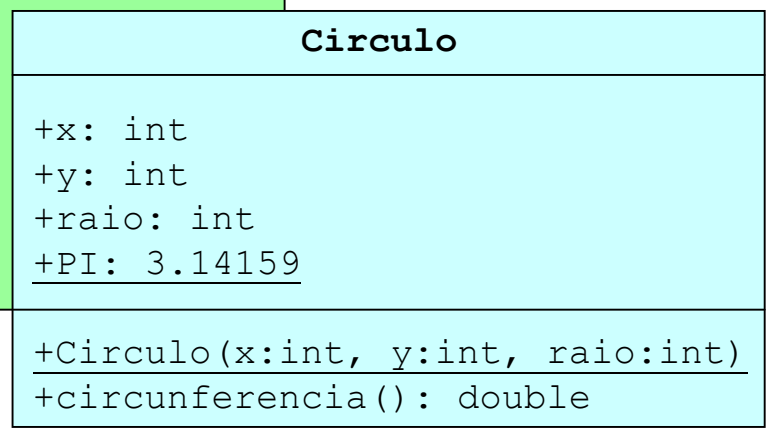
inicialização de variável ocorre quando objeto é construído

método que recebe parâmetro e altera variável

método que retorna variável

Exemplo: um círculo

```
public class Circulo {
   public int x;
   public int y;
   public int raio;
   public static final double PI = 3.14159;

   public Circulo (int x1, int y1, int r) {
      x = x1;
      y = y1;
      raio = r;
   }

   public double circunferencia() {
      return 2 * PI * raio;
   }
}
```



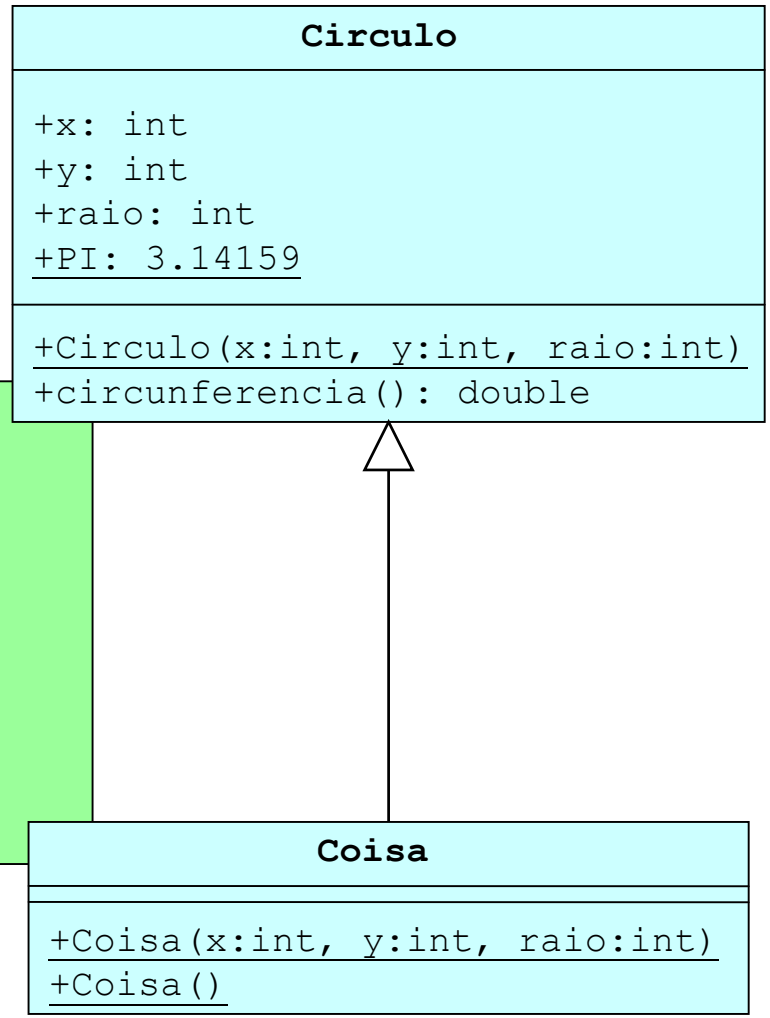
Usando círculos

- *Use dentro de um método ou construtor (blocos de procedimentos)*

```
Circulo c1, c2, c3;
c1 = new Circulo(3, 3, 1);
c2 = new Circulo(2, 1, 4);
c3 = c1; // mesmo objeto!
System.out.println("c1: (" + c1.x + ", "
                           + c1.y + ")");
int circ = (int) c1.circunferencia();
System.out.print("Raio de c1: " + c1.raio);
System.out.println("; Circunferência de c1: "
                   + circ);
```

Herança

```
class Coisa extends Circulo {
    Coisa() {
        this(1, 1, 0);
    }
    Coisa(int x, int y, int z) {
        super(x, y, z);
    }
}
```



*A Coisa **é um** Circulo!*

Exercício

- 1. Escreva uma classe Ponto
  - contém x e y que podem ser definidos em construtor
  - métodos getX() e getY() que retornam x e y
  - métodos setX(int) e setY(int) que mudam x e y
- 2. Escreva uma classe Circulo, que contenha
  - raio inteiro e origem Ponto
  - construtor que define origem e raio
  - método que retorna a área
  - método que retorna a circunferência
  - use java.lang.Math.PI (Math.PI)
- 3. Crie um segundo construtor para Circulo que aceite
  - um raio do tipo int e coordenadas x e y

Vetores

- *Vetores são coleções de objetos ou tipos primitivos*
  - *Os tipos devem ser conversíveis ao tipo em que foi declarado o vetor*

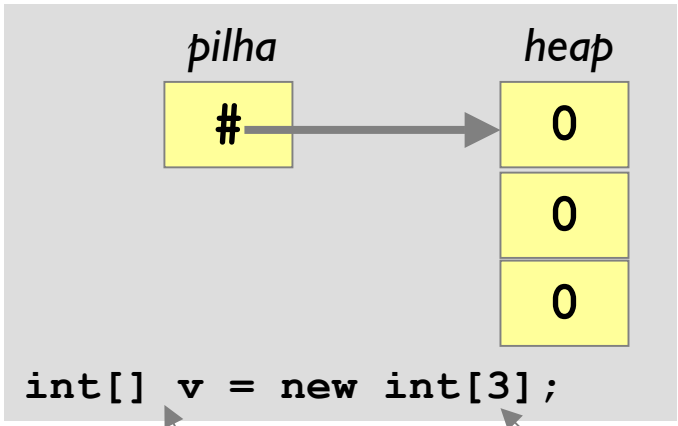
```
int[] vetor = new int[10];
```

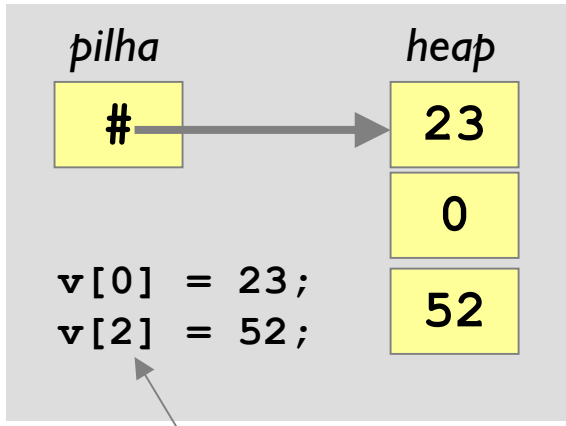
- *Cada elemento do vetor é inicializado a um valor default, dependendo do tipo de dados:*
  - *null, para objetos*
  - *0, para int, long, short, byte, float, double*
  - *Unicode 0, para char*
  - *false, para boolean*
- *Elementos podem ser recuperados a partir da posição 0:*

```
int elemento_1 = vetor[0];

int elemento_2 = vetor[1];
```

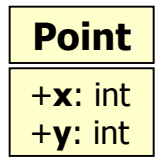
Vetores

- *De tipos primitivos*

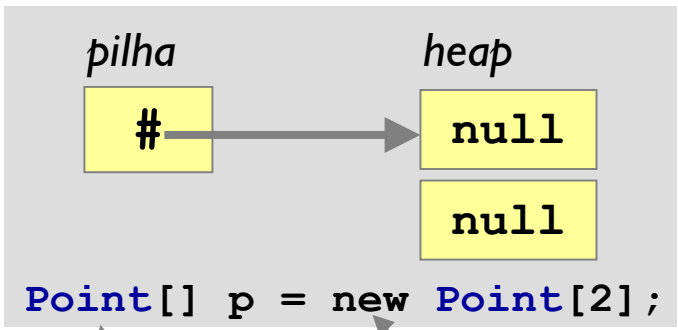


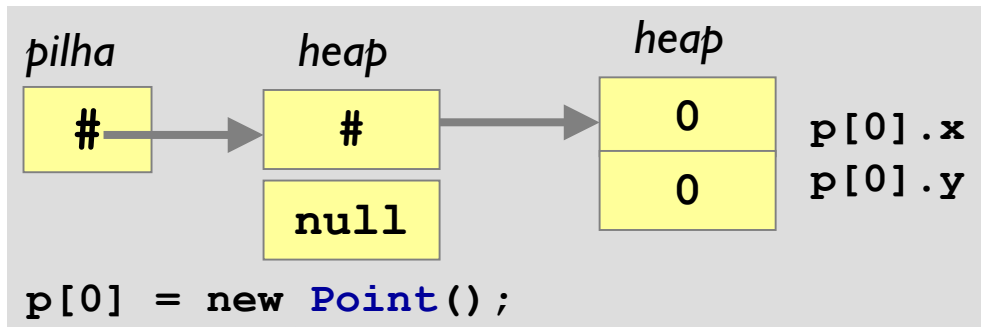



```
class Point {
   public int x;
   public int y;
}
```



- *De objetos (Point é uma classe, com membros x e y, inteiros)*





Inicialização de vetores

- *Vetores podem ser inicializados no momento em que são criados. Sintaxe:*

```
String[] semana = {"Dom", "Seg", "Ter",
                   "Qua", "Qui", "Sex","Sab"};
String[][] usuarios = {
     {"João", "Ninguém"},
     {"Maria", "D.", "Aparecida"},
     {"Fulano", "de", "Tal"}
  };
```

- *Essa inicialização não pode ser usada em outras situações (depois que o vetor já existe), exceto usando new, da forma:*

```
semana =
    new String[] {"Dom", "Seg", "Ter", "Qua",
                  "Qui", "Sex", "Sab"};
```

A propriedade length

- ***length***: *todo vetor em Java possui esta propriedade que informa o número de elementos que possui*
  - *length é uma propriedade read-only*
  - *extremamente útil em blocos de repetição*

```
for (int x = 0; x < vetor.length; x++) {
    vetor[x] = x*x;
}
```

- *Uma vez criados, vetores não podem ser redimensionados*
  - *Use **System.arraycopy()** para copiar um vetor para dentro de outro (alto desempenho)*
  - *Use **java.util.ArrayList** (ou Vector) para manipular com vetores de tamanho variável (baixo desempenho)*
  - *ArrayLists e Vectors são facilmente conversíveis em vetores quando necessário*

Vetores multidimensionais

- *Vetores multidimensionais em Java são vetores de vetores*
  - *É possível criar toda a hierarquia (vetor de vetor de vetor...), para fazer vetores retangulares ...*

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

  - *... ou criar apenas o primeiro nível (antes de usar, porém, é preciso criar os outros níveis)*

```
int [][][] prisma2 = new int [3][][];
prisma2[0] = new int[2][];
prisma2[1] = new int[3][2];
prisma2[2] = new int[4][4];
prisma2[0][0] = new int[5];
prisma2[0][1] = new int[3];
```

Exercícios

- *1. Crie uma classe TestaCirculos que*
  - *a) crie um vetor de 5 objetos Circulo*
  - *b) imprima os valores x, y, raio de cada objeto*
  - *c) declare outra referência do tipo Circulo[]*
  - *d) copie a referência do primeiro vetor para o segundo*
  - *e) imprima ambos os vetores*
  - *f) crie um terceiro vetor*
  - *g) copie os objetos do primeiro vetor para o terceiro*
  - *h) altere os valores de raio para os objetos do primeiro vetor*
  - *i) imprima os três vetores*

Escopo de variáveis

- *Atributos de dados (declarados no bloco da classe): podem ser usadas em qualquer lugar (qualquer bloco) da classe*
  - *Uso em outras classes depende de modificadores de acesso (public, private, etc.)*
  - *Existem enquanto o objeto existir (ou enquanto a classe existir, se declarados static)*
- *Variáveis locais (declaradas dentro de blocos de procedimentos)*
  - *Existem enquanto procedimento (método, bloco de controle de execução) estiver sendo executado*
  - *Não podem ser usadas fora do bloco*
  - *Não pode ter modificadores de acesso (private, public, etc.)*

Exemplo

variáveis visíveis dentro da classe, apenas

novoRaio *é variável local ao método* mudaRaio

maxRaio *é variável local ao método* mudaRaio

raio *é variável de instância*

inutil *é variável local ao bloco* if

```
public class Circulo {

    private int raio;
    private int x, y;

    public double area() {
        return Math.PI * raio * raio;
    }

    public void mudaRaio(int novoRaio) {
        int maxRaio = 50;
        if (novoRaio > maxRaio) {
            raio = maxRaio;
        }
        if (novoRaio > 0) {
            int inutil = 0;
            raio = novoRaio;
        }
    }
}
```

Membros de instância vs. componentes estáticos (de classe)

- Componentes estáticos
  - Os componentes de uma classe, quando declarados 'static', existem independente da criação de objetos
  - Só existe uma cópia de cada variável ou método
- Membros de instância
  - métodos e variáveis que **não tenham** modificador 'static' são membros do objeto
  - Para cada objeto, há uma cópia dos métodos e variáveis
- Escopo
  - Membros de instância não podem ser usados dentro de blocos estáticos: É preciso obter antes, uma referência para o objeto

Exemplos

- *Membros de instância só existem se houver um objeto*

main() não faz parte do objeto!

:Circulo
+x: 0 +y: 0 +raio: 0
area():double

Errado!

```
public class Circulo {

   public int raio;
   public int x, y;

   public double area() {
      return Math.PI * raio * raio;
   }

   public static void main(String[] a){
      raio = 3;
      double z = area();
   }
}
```

membros de instância

Pode. Porque area() faz parte do objeto!

qual *raio? existe?*

qual *area? existe?*

Não pode. Não existe objeto em main()!

Certo!

```
public class Circulo {

   public int raio;
   public int x, y;

   public double area() {
      return Math.PI * raio * raio;
   }

   public static void main(String[] a){
      Circulo c = new Circulo();
      c.raio = 3;
      double z = c.area();
   }
}
```

tem que criar pelo menos um objeto!

raio de `c`

area() de `c`

Variáveis locais vs. variáveis de instância

- Variáveis de instância ...
  - sempre são automaticamente inicializadas
  - são sempre disponíveis no interior dos métodos de instância e construtores
- Variáveis locais ...
  - sempre têm que ser inicializadas antes do uso
  - podem ter o mesmo identificador que variáveis de instância
  - neste caso, é preciso usar a palavra reservada **this** para fazer a distinção

```
class Circulo {
    private int raio;
    public void mudaRaio(int raio) {
        this.raio = raio;
    }
}
```

Comentários

- Há duas formas de incluir comentários em um arquivo Java
  - /* ... comentário de bloco ... */
  - // comentário de linha
- Antes de métodos, construtores, campos de dados e classes, o comentário de bloco iniciado com /** pode ser usado para gerar HTML em documentação
  - Há uma ferramenta (JavaDoc) que gera automaticamente documentação a partir dos arquivos .java
  - relaciona e descreve classes, métodos, etc e cria referências cruzadas
  - Descrições em HTML podem ser incluídas nos comentários especiais /** ... */

Geração de documentação

- *Para gerar documentação de um arquivo ou de uma coleção de arquivos .java use o javadoc:*

```
javadoc arquivo1.java arquivo2.java
```

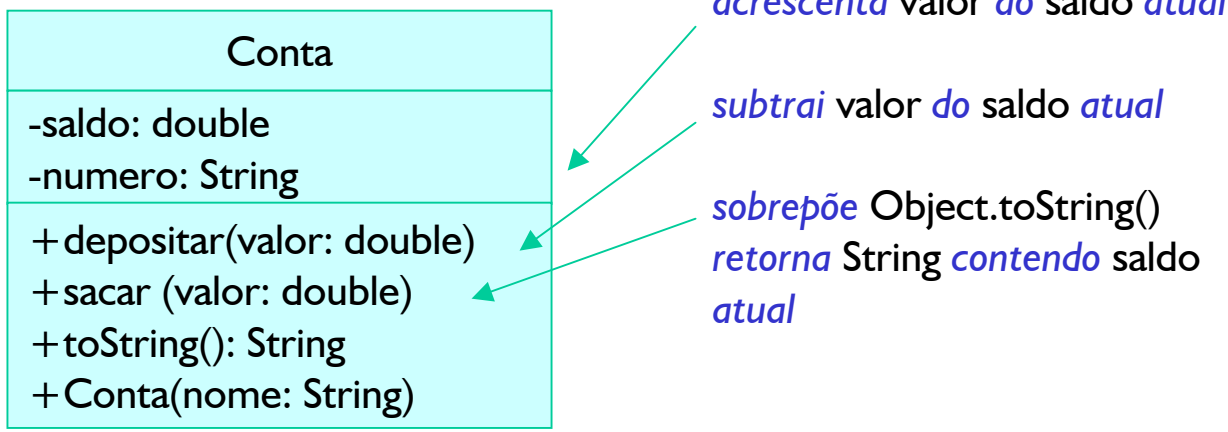
- *O programa criará uma coleção de arquivos HTML, interligados, entre eles estarão*
  - *índice de referências cruzadas*
  - *uma página para cada classe, com links para cada método, construtor e campo público, contendo descrições (se houver) de comentários /** .. */*
- *Consulte a documentação para maiores informações sobre a ferramenta javadoc.*

Convenções de código

- Toda a documentação Java usa uma convenção para nomes de classes, métodos e variáveis
  - Utilizá-la facilitará a manutenção do seu código!
- Classes, construtores e interfaces
  - use caixa-mista com primeira letra maiúscula, iniciando novas palavras com caixa-alta. Não use sublinhado.
  - ex: UmaClasse, Livro
- Métodos e variáveis
  - use caixa mista, com primeira letra minúscula
  - ex: umaVariavel, umMetodo()
- Constantes
  - use todas as letras maiúsculas. Use sublinhado para separar as palavras
  - ex: UMA_CONSTANTE

Exercícios

- *1. Classe Conta e TestaConta*
  - *a) Crie a classe **Conta**, de acordo com o diagrama UML abaixo*



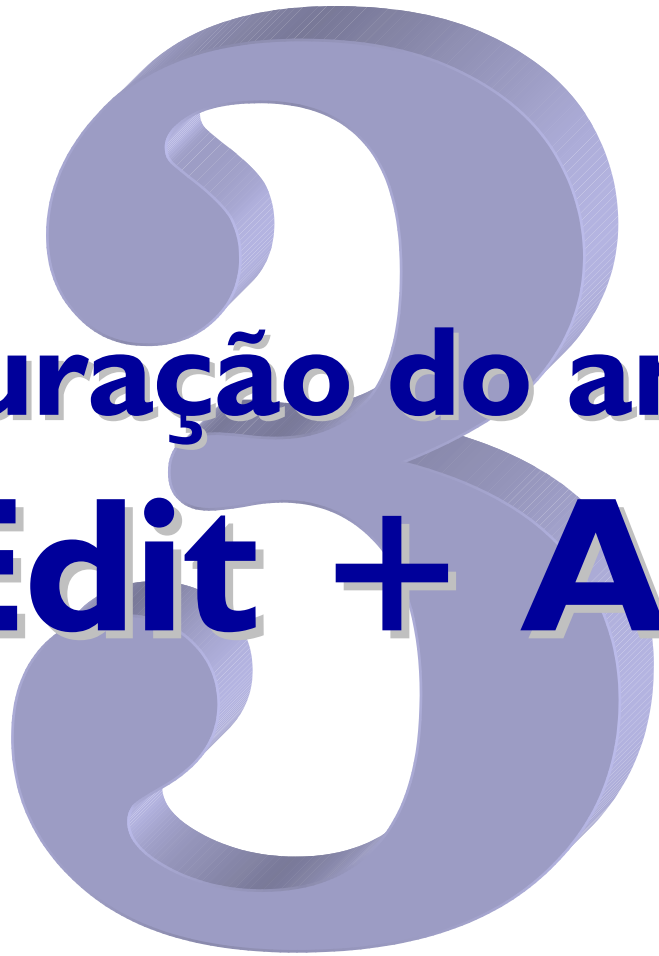
  - *b) Crie uma classe **TestaConta**, contendo um método main(), e simule a criação de objetos Conta, o uso dos métodos depositar() e sacar() e imprima, após cada operação, os valores disponíveis através do método toString()*
  - *c) Gere a documentação javadoc das duas classes*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br



Configuração do ambiente

JEdit + Ant

Helder da Rocha (helder@acm.org)

Java "IDE" configurável

- *Este módulo mostra como montar um ótimo ambiente de desenolvimento para aplicações Java e XML usando apenas ferramentas open-source*
  - ***JEdit** (editor de textos, código e ambiente integrado)*
  - ***Ant** (ferramenta de gerenciamento de projetos)*
  - ***JSDK 1.4.0** (kit de desenvolvimento da Sun)*
- *Onde conseguir o software*
  - *JEdit: www.jedit.org*
  - *Ant: ant.apache.org*
  - *JSDK: java.sun.com*

Use o artigo da Java Magazine anexo como referência atualizada para este módulo

Instalação

- *1. SDK*
  - *a) Instale o JSDK seguindo as instruções mostradas na tela de instalação*
  - *b) Configure uma variável de ambiente JAVA_HOME que aponte para o endereço onde o JSDK foi instalado*
  - *c) Acrescente o caminho $JAVA_HOME/bin no seu PATH*
- *2. Ant*
  - *a) Abra o ZIP da última versão binary do Ant em um diretório (ex: /usr/local/ant ou c:\ant)*
  - *b) Configure uma variável de ambiente ANT_HOME que aponte para o endereço onde o ANT foi instalado*
  - *c) Acrescente o caminho $ANT_HOME/bin no seu PATH*
- *3. JEdit*
  - *Instale o JEdit clicando duas vezes no JAR de distribuição ou executando java -jar jEdit4Install.jar*

Teste da instalação

- SDK e JRE
  - Digite java -version
    - O resultado deve ser o número da versão instalada
  - Digite javac
    - O resultado deve ser uma mensagem de erro com a lista de opções válidas para o javac.
- Ant
  - Digite ant
    - O resultado deve ser a mensagem de erro Buildfile: build.xml does no exist
- JEdit
  - Windows: clique duas vezes no ícone criado
  - Unix: rode o executável no diretório onde foi instalado

JEdit: plug-ins

- O JEdit, sem plug-ins, é um simples editor de textos
- Com plug-ins pode ter mais recursos e melhor integração que muitos IDEs comerciais
- Para instalar plug-ins (precisa estar conectado à Internet)
  - a) Inicie o JEdit
  - b) Selecione o menu "Plugins" / "Plugin Manager" (ou ícone )
  - c) Aperte o botão "Install Plugins". Após a conexão será mostrada uma lista dos plug-ins disponíveis. Clique em cada um para ver sua descrição.
  - d) Selecione o radio-button "Install in system plug-in directory"
  - e) Marque os plug-ins desejados e aperte "Install"
- Instale pelo menos...
  - AntFarm, Buffer Tabs, Console, Drag & Drop, Error List, Java Style, JBrowse, Project Viewer, Templates, XML

Personalização do JEdit (1)

- Reinicie o JEdit após a instalação dos plug-ins
  - Se algum plug-in instalado apresentar erro, uma mensagem irá ser exibida na abertura do JEdit. Você pode
    - corrigir o erro (a mensagem explica como ou a causa)
    - desinstalar o plug-in (no menu Plugins)
- Docking
  - Coloca plug-ins frequentemente usados nas laterais da área de trabalho para fácil utilização
  - Selecione o menu "Utilities" / "Global Options"
  - Na opção jEdit Options / Docking, selecione a segunda configuração de tela: 
  - Mude a posição default dos seguintes plug-ins:
    - AntFarm: left
    - Console: bottom
    - ErrorList: bottom
    - JBrowse: left
    - Project: left
    - XMLTree: left

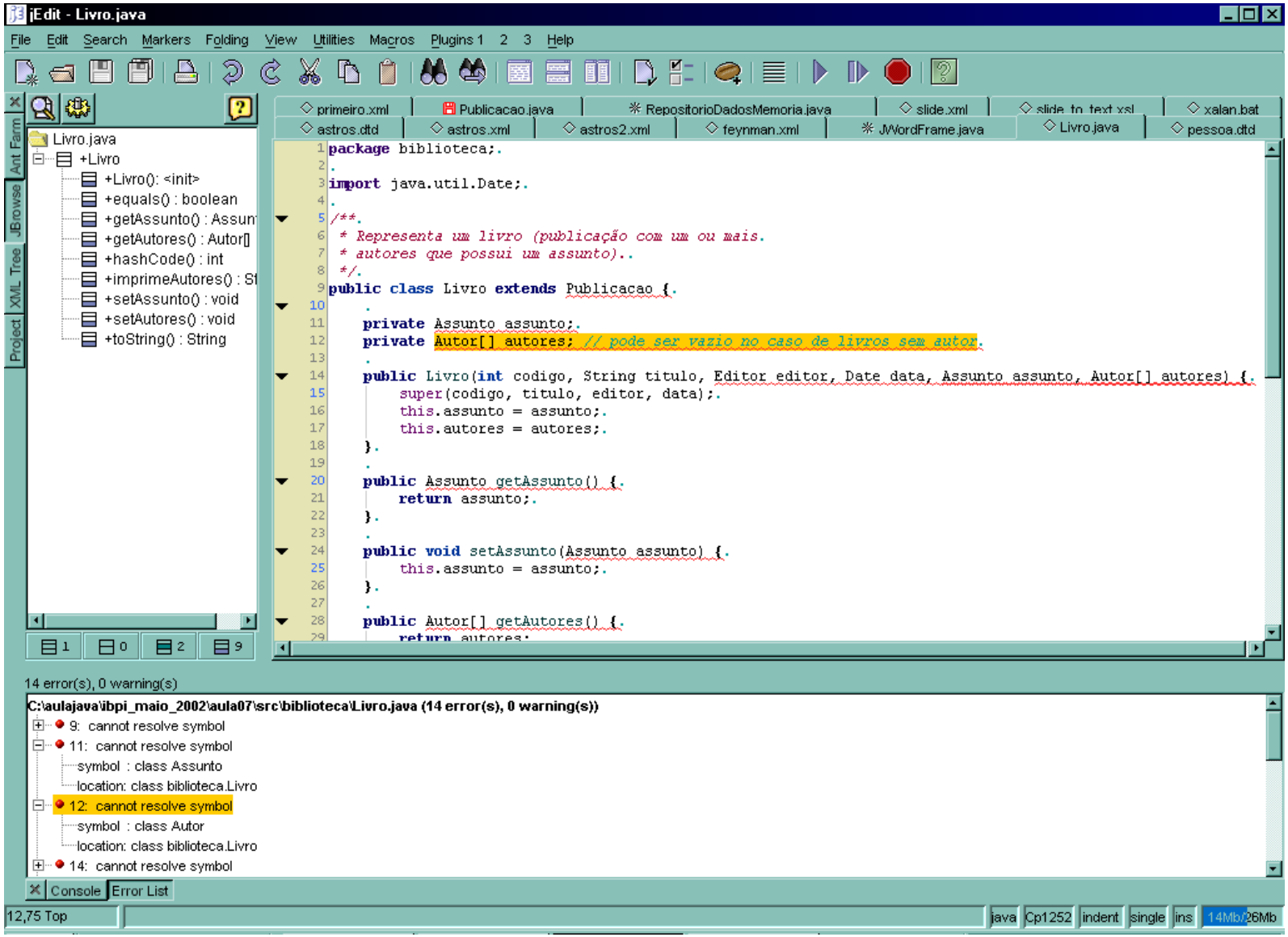
Personalização do JEdit (2)

- *Tabs para seleção de janelas de texto*
  - *Menu Global Options / jEdit Options / General*
    - *Desmarque "Show buffer switcher" e "Show search bar"*
  - *Menu Global Options / Plugin Options / Buffer Tabs*
    - *Marque "Enable buffer tabs by default"*
    - *Selecione "Location of buffer tabs:" para "top"*
- *AntFarm*
  - *Menu Global Options / Plugin Options / AntFarm*
    - *Selecione Build Options*
    - *Na seção "Build Execution Method", na segunda caixa de texto, informe o caminho até o executável do Ant na sua máquina (ex: c:\ant\bin\ant.bat)*
    - *Na seção "General Build Options" marque as opções "Load build files..." e "Save all buffers..."*

Personalização do JEdit (3)

- *ErrorList*
  - *Menu Global Options / Plugin Options / ErrorList*
    - *Marque "automatically display on error"*
- *JBrowse*
  - *Menu Global Options / Plugin Options / JBrowse*
    - *Marque "Display Status Bar", "Automatic parse" e "Sort"*
- *Numeração de linhas*
  - *Menu Global Options / jEdit Options / Gutter*
    - *Marque "Line Numbering"*
- *Tabulação*
  - *Menu Global Options / jEdit Options / Editing*
    - *Em "Tab Width" e "Indent Width" coloque "4"*

Resultado da personalização



Como usar o Console / ErrorList

- O **Console** do JEdit serve para
  - mostrar mensagens de erro
  - rodar o compilador
  - rodar outras aplicações do sistema
- Erros ocorridos durante a compilação, execução ou outro processo que produza erros são coletados no **ErrorList**, que
  - permite acesso rápido à fonte do erro através de double-click
  - oferece detalhamento dos erros
- Ajuste o Console e ErrorList na parte inferior do seu JEdit

Criando um projeto

- Para criar um novo projeto:
  - primeiro escolha um local no seu disco que será a raiz de seus projetos
  - depois crie um diretório para seu projeto
  - finalmente, usando o JProject, clique na opção "Create Project", informe um nome e o diretório
- Como adicionar arquivos ao projeto
  - Sempre que o JProject estiver aberto, ele perguntará se o arquivo salvo deve ser incluído no projeto
  - Arquivos também podem ser adicionados usando o ícone correspondente

Como criar um template

- Templates são úteis para se ganhar tempo com textos ou estruturas repetitivas, por exemplo:
  - Estrutura default de uma página HTML
  - Estrutura default de uma classe Java
- Para criar um novo template,
  - Edite-o no JEdit e, na hora de salvar, selecione o menu PlugIns/Templates/Save Template
  - Selecione Refresh Templates para que apareça na lista
  - Crie um arquivo novo e selecione o template da lista para que o seu conteúdo seja copiado
- Crie templates básicos
  - Classe Java
  - Buildfile do Ant

Outros plug-ins

- SpeedJava ("code insight") - não será instalado durante o curso
  - Acionado quando você digita um ponto "."
  - Lista métodos e variáveis da classe do objeto selecionado
  - Limitação (versão 0.2): classe tem que ter sido importada com import pacote.* (não funciona se classe foi importada nominalmente)
- Reformat Buffer e JavaStyle / Reformat Buffer
  - Rearruma código Java
  - Configure para refletir seu estilo de codificação
  - JavaStyle oferece mais opções de configuração
- XML e XSLT
  - Oferecem suporte para XML, XSLT e XPath
  - Validam XML com DTD, oferecem ajuda de contexto (quando há um DTD vinculado), montam árvore (plug-in XML Tree)

Problemas

- *Os plug-ins do JEdit são desenvolvidos por programadores independentes e podem não ter a mesma qualidade ou utilidade dos recursos nativos do JEdit*
  - *Alguns contêm bugs (falham ocasionalmente)*
  - *Alguns poderiam ser melhores*
  - *A integração entre os plug-ins é inferior à desejável*
  - *Ainda não há plugins para desenho de GUI, construção de EJBs, etc.*
- *Soluções*
  - *1) Esperar versões mais novas dos plug-ins (a atualização requer apenas apertar um botão (Update Plugins) no Plugin Manager*
  - *2) Enviar sugestões aos autores dos plug-ins*
  - *3) Escrever macros, scripts do Ant (EJB, arquivos WAR, etc.)*
  - *4) Participar do projeto open-source: baixar o código-fonte Java, fazer as alterações desejadas e enviar patches aos autores*
  - *5) Escrever novos plug-ins (integrando ou não com os existentes)*
  - *6) Usar outro IDE*

Alguns IDEs alternativos

- *Open source*
  - *Projeto Eclipse*
  - *NetBeans*
  - *Jext (Java Text Editor - similar ao JEdit) - não edita GUI*
- *Free*
  - *Borland JBuilder Personal Edition*
  - *Sun Forté for Java Community Edition (baseado no NetBeans)*
- *Leves*
  - *JCreator*
  - *Kawa*
  - *IDEA*
- *Comerciais*
  - *IBM Visual Age / WebSphere Studio*
  - *Borland JBuilder*
  - *Sun Forté for Java*

Servem para aumentar a produtividade: evite usá-los para aprender Java (prefira um editor de textos como oJEdit ou Jext)

Teste cada um com as ferramentas e recursos que você costuma usar e veja o que melhor se adapta às suas necessidades.

Exercícios

1. Organize os exercícios que você fez nos módulos anteriores em projetos no JEdit
   - a) Crie um diretório para cada um (coloque as fontes no subdiretório src)
   - b) Crie um projeto no Project Manager
   - c) Faça toda a compilação através do Ant (defina o build.xml disponível no diretório cap03/ como template, salve-o na raiz do seu projeto e inclua-o clicando no "+" do AntFarm). Remova quando fechar o projeto.
   - d) Mude sempre o nome do seu build file
   - e) Execute através do Console (use um target do Ant)

Dica: veja artigo da JavaMagazine (anexo) que mostra com detalhes como montar o ambiente deste módulo

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Como usar a documentação da API Java 2

Helder da Rocha
www.argonavis.com.br

Documentação

- *Aprender a usar a documentação é essencial para quem deseja desenvolver aplicações em Java*
- *A documentação da linguagem, ferramentas e API é um download separado do SDK*
- *Para instalar a documentação Java, abra o arquivo ZIP na raiz da sua instalação Java*
  - *A documentação será instalada abaixo do subdiretório docs/ da instalação Java ($JAVA_HOME)*
    - `%JAVA_HOME%\docs\` (ex: `c:\jdk1.4.0\docs/`)
    - `$JAVA_HOME/docs/` (ex: `/usr/java/j2sdk1.4.0/docs/`)
  - *A documentação da API Java está em*
    - `$JAVA_HOME/docs/api/index.html`
  - *Manuais, tutoriais sobre recursos da linguagem em*
    - `$JAVA_HOME/docs/index.html`

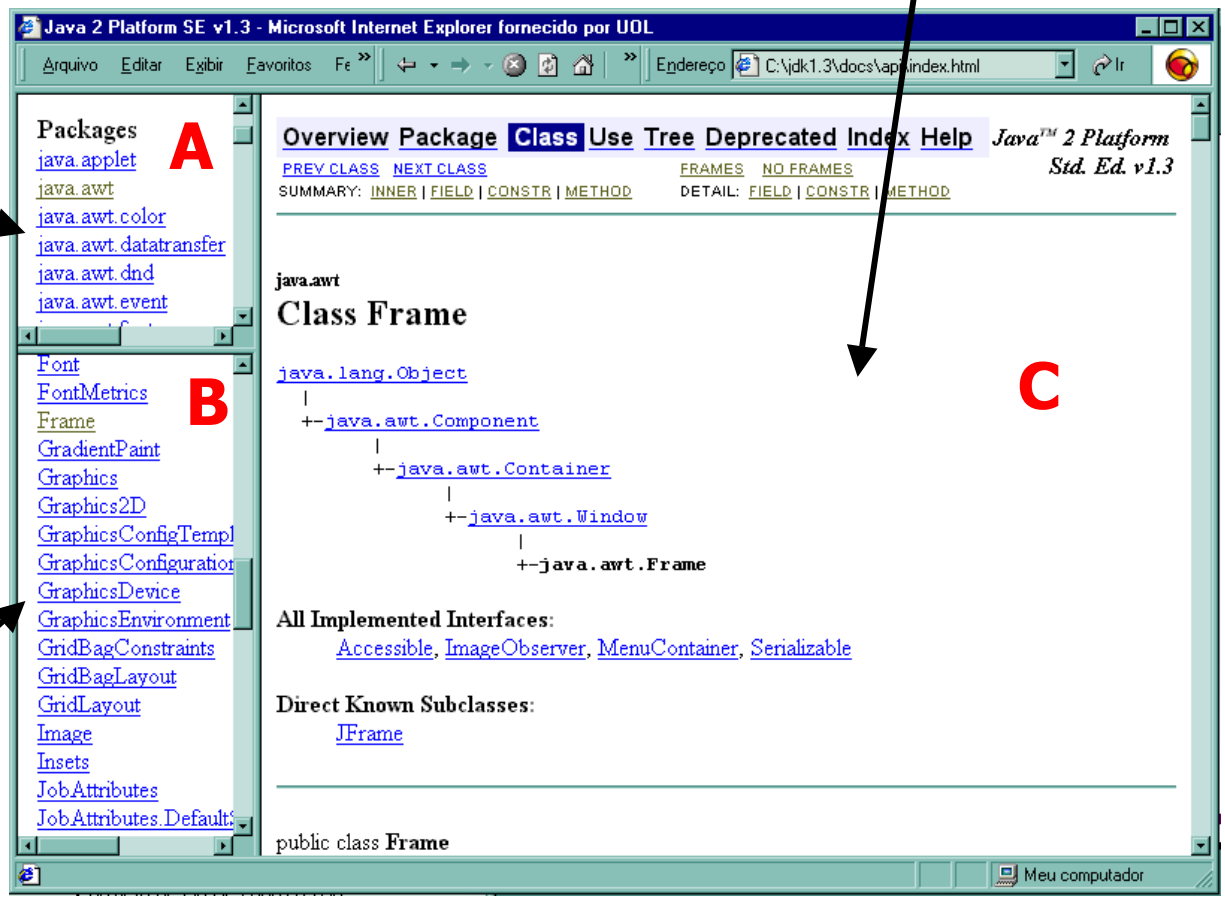
Documentação

Descrição da classe (escolhida na janela B)
- hierarquia
- documentação detalhada, métodos, variáveis, etc.

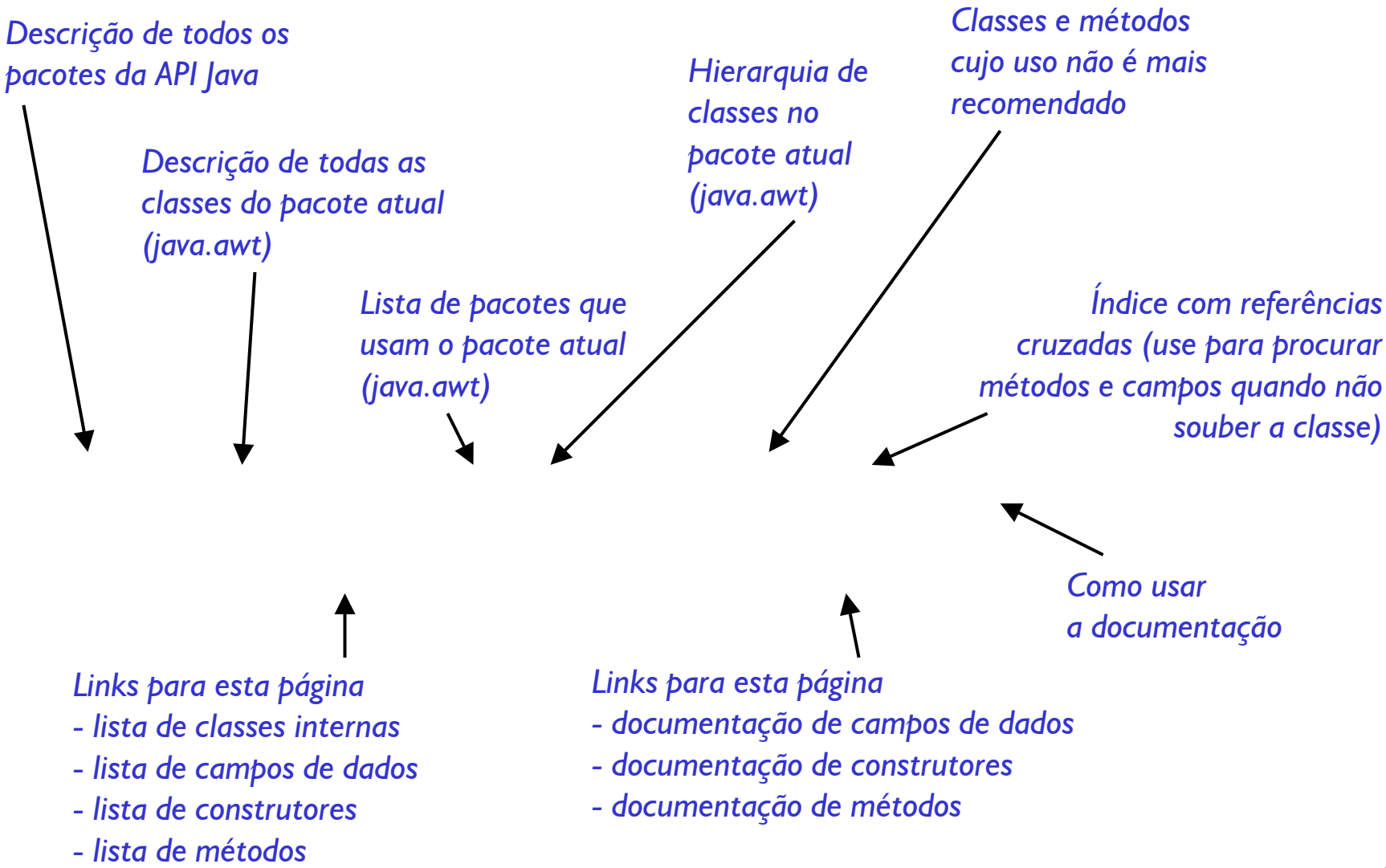
Se um método não for encontrado na classe mostrada, procure nas superclasses (use as referências cruzadas)

Lista de pacotes

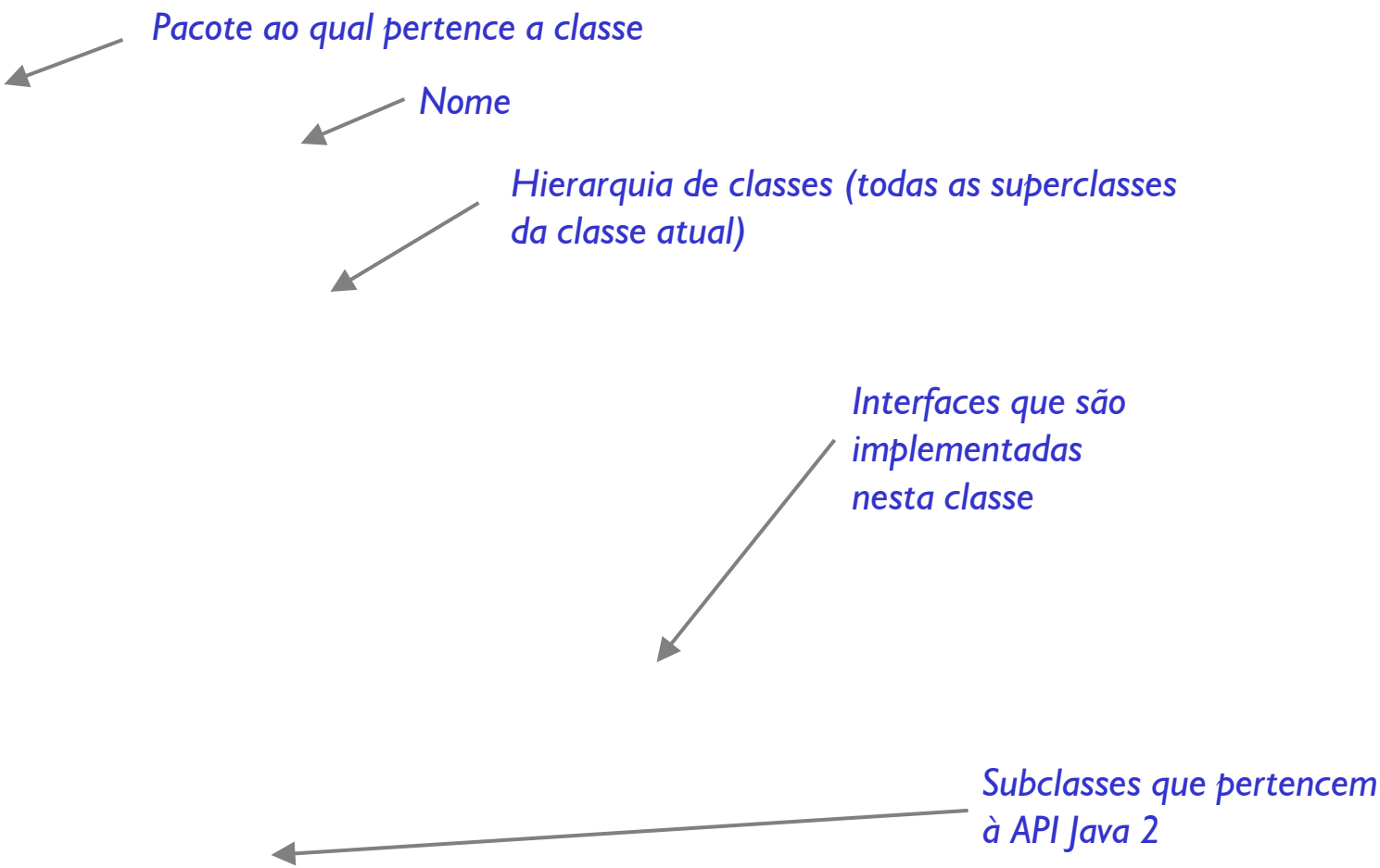
Lista de classes e interfaces do pacote escolhido na janela (A)



Como usar a documentação (2)

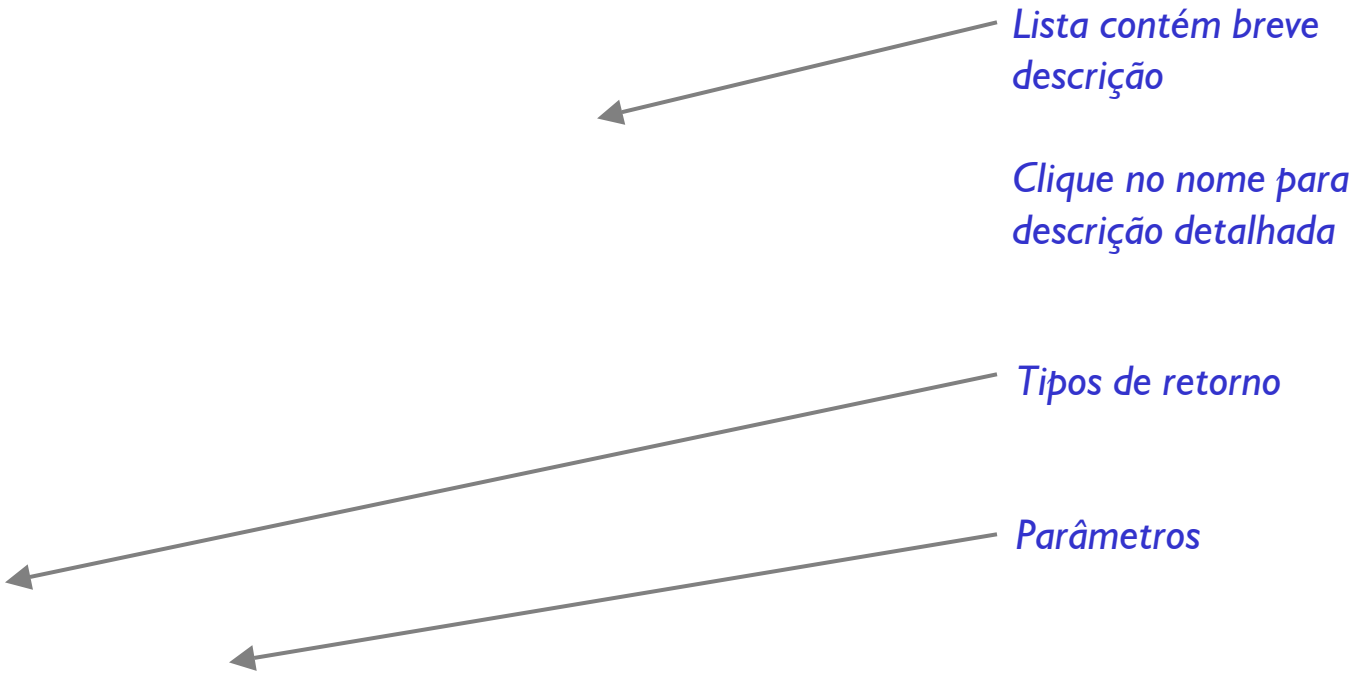


Como usar a documentação (3)

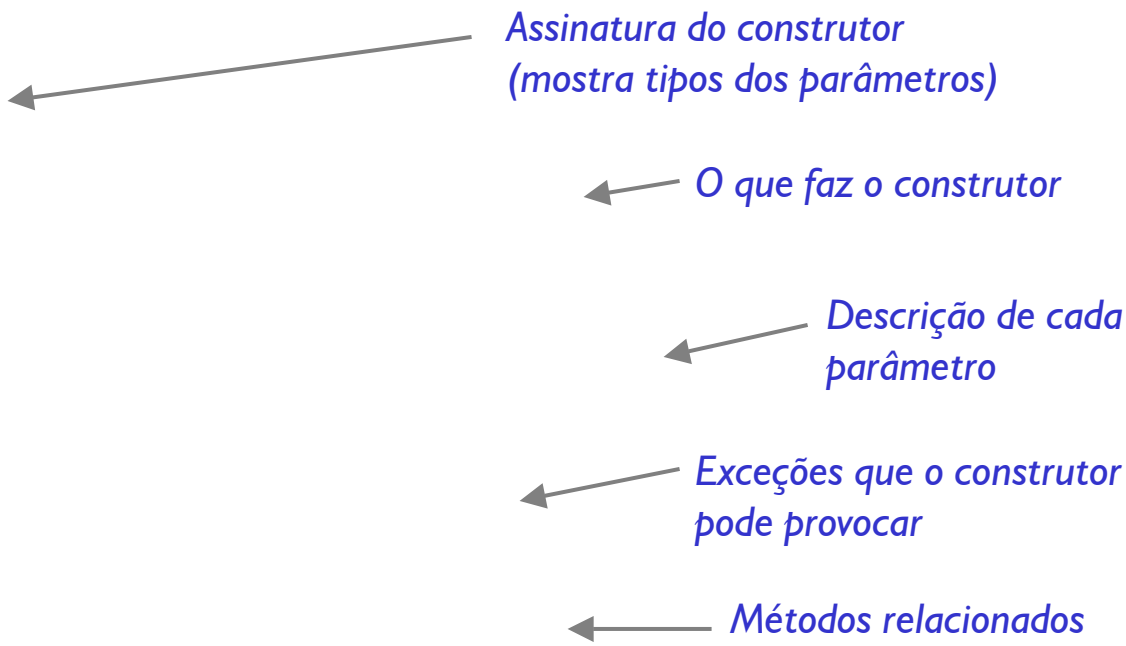


Como usar a documentação (4)

- *Listas de classes internas, campos de dados, métodos e construtores*



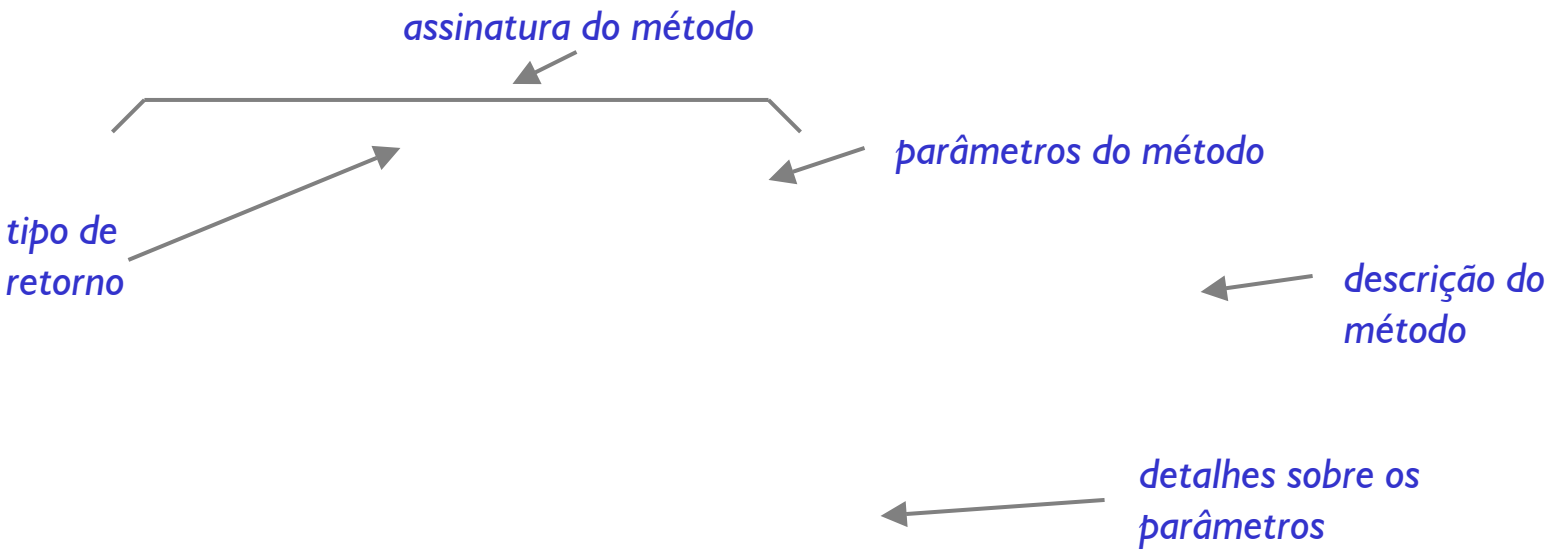
Documentação de um construtor



- *Alguns exemplos de como usar o construtor acima*

```
Frame f = new Frame("Título da Janela", null);

class MinhaJanela extends Frame {
   public MinhaJanela(String titulo) {
      super(titulo, null);
   ...
```

Documentação de um método



- *Para chamar o método acima (fillRect() da classe java.awt.Graphics)*

```
g.fillRect(25, 50, 100, 200);  // g: referência Graphics
```

  - *Desenha um retângulo preenchido com a cor atual do contexto gráfico, com seu canto superior esquerdo na posição x:25 e y:50, com 100 pixels de largura por 200 de altura*
- *Para sobrepor, repita a assinatura do método e forneça sua implementação*

Exercício 1: documentação

a) Crie duas classes
- Uma classe deve estender `javax.swing.JFrame` (esta classe irá fornecer a interface gráfica)
- Outra classe, executável (contendo `main`) para iniciar a classe gráfica.

b) Crie um construtor na classe derivada de JFrame
- O construtor deve definir o título da janela (use `super()` com os argumentos correspondentes ou descubra um método que faça isto)
- Deve definir o tamanho (`setSize`): 300x300, e tornar a janela visível (`setVisible`) - procure em `java.awt.omponent`

c) Sobreponha o método `paint(Graphics g)` herdado de `java.awt.Component`
- Veja a assinatura correta na documentação. `paint()` é chamada pelo sistema automaticamente para pintar o contexto gráfico da aplicação
- Método `paint` deve usar contexto gráfico da janela (objeto `g`) para mudar a cor atual de `g` (para vermelho, por exemplo) (`setColor`) e desenhar, em `g`, um círculo (`fillOval`), tendo a metade da largura da janela (150), e posicionado no centro.

Ex 2: geração de documentação

a) use comentários de documentação para descrever a classe que você criou, o construtor e o método paint()

- Use comentários /** ... */ ANTES dos métodos, construtores e classes
- Use `@param nome descrição` para descrever os parâmetros
- `@param` é um comando do javadoc. Pode vir no início da linha (o início da linha pode ter espaços ou asteriscos *). Exemplo:

```
/** @param mensagem Texto contendo descrição... */
```

b) Rode o javadoc passando a classe como argumento

```
> javadoc NomeDaClasse
```

- Navegue e explore os documentos HTML gerados

c) Crie uma target no Ant para gerar documentação de todo o projeto (veja no capítulo 8 um resumo dos tags do Ant).

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Tipos, literais, operadores e controle de fluxo

Helder da Rocha
www.argonavis.com.br

Operadores e controle de fluxo da execução

- Este módulo explora as estruturas procedurais da linguagem Java
- Operadores
  - Aritméticos, lógicos, binários, booleanos, de deslocamento, de concatenação, de conversão, ...
- Conversão de tipos
  - Promoção
  - Coerção (cast)
- Estruturas de controle de execução
  - if-else,
  - for, while, do-while
  - break, continue, rótulos
  - switch (case)

Operadores

- *Um operador produz um novo valor a partir de um ou mais argumentos*
- *Os operadores em Java são praticamente os mesmos encontrados em outras linguagens*
  - +, -, /, *, =, ==, <, >, >=, &&, *etc.*
- *A maior parte dos operadores só trabalha com valores de tipos primitivos.*
- *Exceções:*
  - + e += *são usados na concatenação de strings*
  - !=, = e == *são usados também com objetos (embora não funcionem da mesma forma quanto aos valores armazenados nos objetos)*

Lista de operadores do Java

OPERADOR	FUNÇÃO	OPERADOR	FUNÇÃO
`+`	Adição	`~`	Complemento
`-`	Subtração	`<<`	Deslocamento à esquerda
`*`	Multiplicação	`>>`	Deslocamento à direita
`/`	Divisão	`>>>`	Desloc. a direita com zeros
`%`	Resto	`=`	Atribuição
`++`	Incremento	`+=`	Atribuição com adição
`--`	Decremento	`-=`	Atribuição com subtração
`>`	Maior que	`*=`	Atribuição com multiplicação
`>=`	Maior ou igual	`/=`	Atribuição com divisão
`<`	Menor que	`%=`	Atribuição com resto
`<=`	Menor ou igual	`&=`	Atribuição com AND
`==`	Igual	`|=`	Atribuição com OR
`!=`	Não igual	`^=`	Atribuição com XOR
`!`	NÃO lógico	`<<=`	Atribuição com desl. esquerdo
`&&`	E lógico	`>>=`	Atribuição com desloc. direito
`||`	OU lógico	`>>>=`	Atrib. C/ desloc. a dir. c/ zeros
`&`	AND	`? :`	Operador ternário
`^`	XOR	`(`*tipo*`)`	Conversão de tipos (cast)
`|`	OR	`instanceof`	Comparação de tipos

Precedência

- A precedência determina em que ordem as operações em uma expressão serão realizadas.
  - Por exemplo, operações de multiplicação são realizadas antes de operações de soma:

```
int x = 2 + 2 * 3 - 9 / 3;  // 2+6-3 = 5
```

- Parênteses podem ser usados para sobrepor a precedência

```
int x = (2 + 2) * (3 - 9) / 3; // 4*(- 6)/3 = - 8
```

- A maior parte das expressões de mesma precedência é calculada da esquerda para a direita

```
int y = 13 + 2 + 4 + 6; // (((13 + 2) + 4) + 6)
```

  - Há exceções. Por exemplo, atribuição.

Tabela de precedência

ASSOC	TIPO DE OPERADOR	OPERADOR
D a E	separadores	`[] . ; , ()`
E a D	operadores unários	`new (cast) +expr -expr ~ !`
E a D	incr/decr pré-fixado	`++expr --expr`
E a D	multiplicativo	`* / %`
E a D	aditivo	`+ -`
E a D	deslocamento	`<< >> >>>`
E a D	relacional	`< > >= <= instanceof`
E a D	igualdade	`== !=`
E a D	AND	`&`
E a D	XOR	`^`
E a D	OR	`|`
E a D	E lógico	`&&`
E a D	OU lógico	`||`
D a E	condicional	`?:`
D a E	atribuição	`= += -= *= /= %= >>= <<= >>>= &= ^= !=`
E a D	incr/decr pós fixado	`expr++ expr--`

Literais de caracteres em Java

SEQÜÊNCIA	VALOR DO CARACTERE
`\b`	Retrocesso (backspace)
`\t`	Tabulação
`\n`	Nova Linha (new line)
`\f`	Alimentação de Formulário (form feed)
`\r`	Retorno de Carro (carriage return)
`\"`	Aspas
`\'`	Aspa
`\\`	Contra Barra
`\`*nnn*	O caractere correspondente ao valor octal *nnn*, onde *nnn* é um valor entre 000 e 0377.
`\u`*nnnn*	O caractere Unicode *nnnn*, onde *nnnn* é de um a quatro dígitos hexadecimais. Seqüências Unicode são processadas antes das demais seqüências.

Atribuição

- A atribuição é realizada com o operador '='
  - '=' serve apenas para atribuição – não pode ser usado em comparações (que usa '==')!
  - Copia o valor da variável ou constante do lado direito para a variável do lado esquerdo.

```
x = 13;  // copia a constante inteira 13 para x
y = x;   // copia o valor contido em x para y
```

- A atribuição copia valores
  - O valor armazenado em uma variável de tipo primitivo é o valor do número, caractere ou literal booleana (true ou false)
  - O valor armazenado em uma variável de tipo de classe (referência para objeto) é o ponteiro para o objeto ou null.
  - Conseqüentemente, copiar referências por atribuição não copia objetos mas apenas cria novas referências para o mesmo objeto!

Passagem de valores via atribuição

- *Variáveis de tipos primitivos*

Pilha após linha 2

letraPri	'a'
letraPri2	'a'

Pilha após linha 3

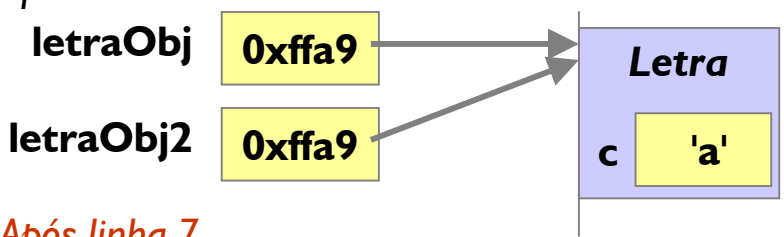
letraPri	'b'
letraPri2	'a'

```
(...)
1: char letraPri = 'a';
2: char letraPri2 = letraPri;
3: letraPri = 'b';
(...)
```

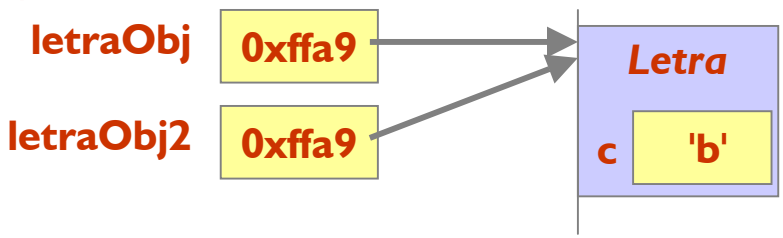
- *Referências de objetos*

```
public class Letra {
    public char c;
}
```

Após linha 6



Após linha 7




```
(...)
4: Letra letraObj = new Letra();
5: letraObj.c = 'a';
6: Letra letraObj2 = letraObj;
7: letraObj2.c = 'b';
(...)
```

Operadores matemáticos

- `+` *adição*
- `-` *subtração*
- `*` *multiplicação*
- `/` *divisão*
- `%` *módulo (resto)*

- *Operadores unários*
  - `-n` e `+n` *(ex: -23) (em uma expressão: 13 + -12)*
  - *Melhor usar parênteses: 13 + (-12)*
- *Atribuição com operação*
  - +=, -=, *=, /=, %=
  - `x = x + 1` *equivale a* `x += 1`

Incremento e decremento

- *Exemplo*

```
int a = 10;
int b = 5;
```

- *Incrementa ou decrementa antes de usar a variável*

```
int x = ++a; // a contém 11, x contém 11
int y = --b;   // b contém 4, y contém 4
```

  - *A atribuição foi feita DEPOIS!*

- *Incrementa ou decrementa depois de usar a variável*

```
int x = a++; // a contém 11, x contém 10
int y = b--;   // b contém 4, y contém 5
```

  - *A atribuição foi feita ANTES!*

Operadores relacionais

- `==` *igual*
- `!=` *diferente*
- `<` *menor*
- `<=` *menor ou igual*
- `>` *maior*
- `>=` *maior ou igual*

- *Sempre produzem um resultado booleano*
  - `true` *ou* `false`
  - *Comparam os valores de duas variáveis ou de uma variável e uma constante*
  - *Comparam as referências de objetos (apenas == e !=)*

Operadores lógicos

- **&&** *E (and)*
- **||** *Ou (or)*
- **!** *Negação (not)*

- *Produzem sempre um valor booleano*
  - *true ou false*
  - *Argumentos precisam ser valores booleanos ou expressões com resultado booleano*
  - *Por exemplo:* `(3 > x) && !(y <= 10)`
- *Expressão será realizada até que o resultado possa ser determinado de forma não ambígua*
  - *"short-circuit"*
  - *Exemplo: (*`false && <qualquer coisa>`*)*
  - *A expressão <qualquer coisa> não será calculada*

Operadores orientados a bit

- **&** *and*
- **|** *or*
- **^** *xor (ou exclusivo)*
- **~** *not*

- *Para operações em baixo nível (bit por bit)*
  - *Operam com inteiros e resultados são números inteiros*
  - *Se argumentos forem booleanos, resultado será igual ao obtido com operadores booleanos, mas sem 'curto-circuito'*
  - *Suportam atribuição conjunta: &=, |= ou ^=*

Operadores de deslocamento

- << deslocamento de bit à esquerda (multiplicação por dois)
- >> deslocamento de bit à direita (divisão truncada por dois)
- >>> deslocamento à direita sem considerar sinal (acrescenta zeros)
- Para operações em baixo nível (bit a bit)
  - Operam sobre inteiros e inteiros longos
  - Tipos menores (short e byte) são convertidos a int antes de realizar operação
  - Podem ser combinados com atribuição: <<=, >>= ou >>>=

Operador ternário (if-else)

- *Retorna um valor ou outro dependendo do resultado de uma expressão booleana*
  - `variavel = expressão ? valor, se true`
    `: valor, se false;`
- *Exemplo:*

```
int x = (y != 0) ? 50 : 500;
String tit = (sex == 'f') ? "Sra." : "Sr
num + " pagina" + (num != 1) ? "s" : ""
```

- *Use com cuidado*
  - *Pode levar a código difícil de entender*

Operador de concatenação

- *Em uma operação usando "+" com dois operandos, se um deles for String, o outro será convertido para String e ambos serão concatenados*
- *A operação de concatenação, assim como a de adição, ocorre da direita para a esquerda*

```
String s = 1 + 2 + 3 + "=" + 4 + 5 + 6;
```

- *Resultado: s contém a String* **"6=456"**

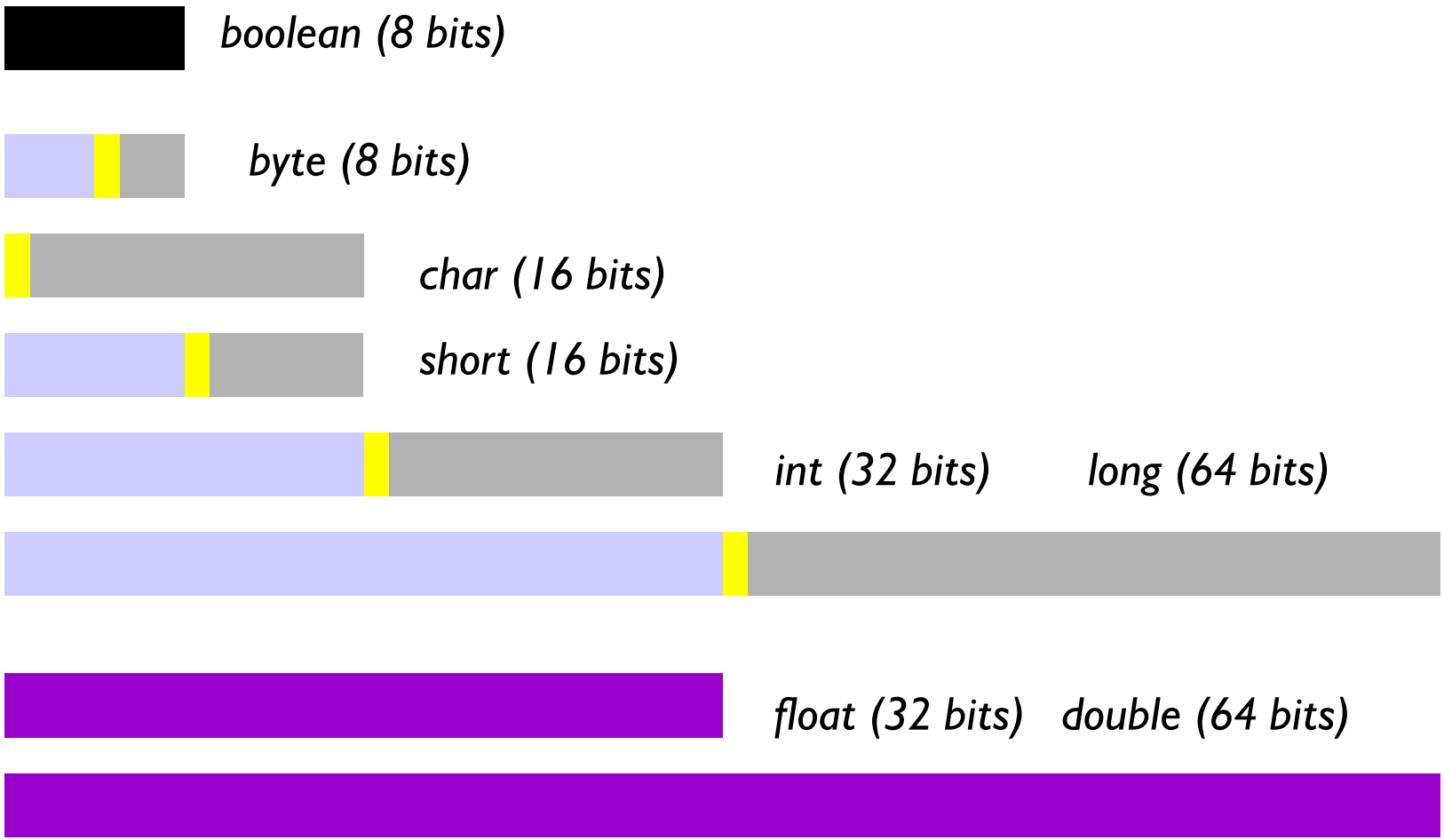
instanceof

- *instanceof é um operador usado para comparar uma referência com uma classe*
  - *A expressão será true se a referência for do tipo de uma classe ou subclasse testada e false, caso contrário*
  - *Sintaxe: referência instanceof Classe*
- *Exemplo:*

```
if (obj instanceof Point) {
  System.out.println("Descendente de Point");
}
```

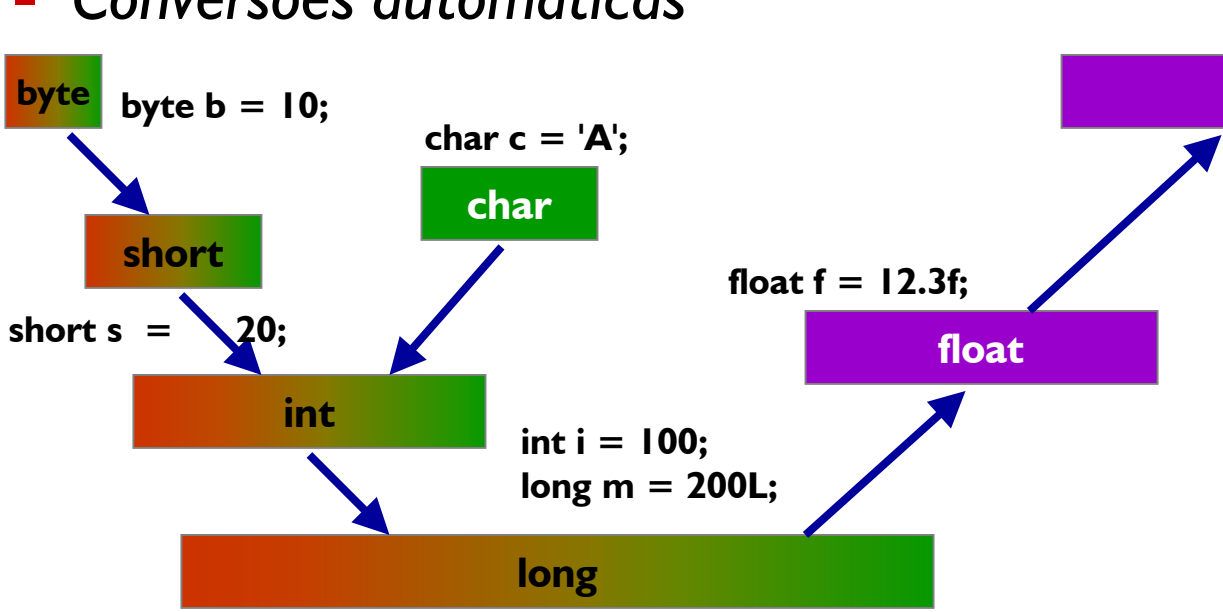
Mais detalhes sobre este assunto em capítulos posteriores!

Tipos de dados



Conversão de tipos primitivos

- *Java converterá um tipo de dados em outro sempre que isto for apropriado*
- *As conversões ocorrerão automaticamente quando houver garantia de não haver perda de informação*
  - *Tipos menores em tipos maiores*
  - *Tipos de menor precisão em tipos de maior precisão*
  - *Tnteiros em ponto-flutuante*
- *Conversões automáticas*



Algumas conversões legais

```
short s2 = b;
int i2 = c;
int i3 = s;
long m2 = i;
float f2 = m;
double d2 = f;
```

Conversão de referências

- *Pode-se atribuir uma referência A a uma outra referência B de um tipo diferente, desde que*
  - *B seja uma superclasse (direta ou indireta) de A: Qualquer referência pode ser atribuída a uma referência da classe Object*
  - *B seja uma interface implementada por A: mais detalhes sobre interfaces em aulas futuras*

```
class Carro extends Veiculo {...}

class Veiculo implements Dirigivel {}

class Porsche extends Carro {...}
```

Algumas conversões legais

```
Carro c = new Carro();
Veiculo v = new Carro();
Object o = new Carro();
Dirigivel d = new Carro();
Carro p = new Porsche();
```

Mais detalhes sobre este assunto em capítulos posteriores!

Operadores de coerção

- *Na coerção (cast), o programador assume os riscos da conversão de dados*
  - *No tipo byte cabem inteiros até 127*
  - *No tipo short cabem inteiros até 32767*
  - *Não há risco de perda de informação na atribuição a seguir*

    ```
    short s = 100; byte b = s;
    ```

    (pois 100 cabe em byte) mas o compilador acusará erro porque um short não pode ser atribuído a byte.
  - *Solução*

    operador de coerção (cast)

    ```
    byte b = (byte) s;
    ```
  - *O programador "assume o risco", declarando entre parênteses, que o conteúdo de* `s` *cabe em* `byte`.
  - *O operador de coerção tem maior precedência que os outros operadores!*

Promoção

- *Qualquer operação com dois ou mais operandos de tipos diferentes sofrerá promoção, isto é, conversão automática ao tipo mais abrangente, que pode ser*
  - *O maior ou mais preciso tipo da expressão (até double)*
  - *O tipo int (para tipos menores que int)*
  - *O tipo String (no caso de concatenações) - (na verdade isto não é uma promoção)*
- *Exemplos*
  - `String s = 13 - 9 * 16 + "4" + 9 + 2; // "-131492"`
  - `double d = 12 + 9L + 12.3; // tudo é promovido p/ double`
  - `byte b = 9; byte c = 10; byte d = 12;`
    `byte x = (byte) (b + c + d);`

a partir daqui só o sinal '+' é permitido!

cast é essencial aqui! Observe os parênteses!

promovidos para int!

Controle de execução

- O controle do fluxo da execução em Java utiliza os mesmos comandos existentes em outras linguagens
  - Repetição: for, while, do-while
  - Seleção: if-else, switch-case
  - Desvios (somente em estruturas de repetição): continue, break, rótulos
- Não existe comando goto
  - goto, porém, é palavra-reservada.

true e false

- *Todas as expressões condicionais usadas nas estruturas for, if-else, while e do-while são expressões booleanas*
  - *O resultado das expressões deve ser sempre true ou false*
  - *Não há conversões automáticas envolvendo booleanos em Java (evita erros de programação comuns em C/C++)*

Código errado.
Não compila em Java

```
int x = 10;
if (x = 5) {
    ...
}
```

código aceito em C/C++ (mas provavelmente errado) x, com valor 5, converte-se em 'true'.

Código correto.
x == 5 é expressão com resultado true ou false

```
int x = 10;
if (x == 5) {
    ...
}
```

if-else

- *Sintaxe*

```
if (expressão booleana)
   instrução_simples;

if (expressão booleana) {
   instruções
}
```

```
if (expressão booleana) {
   instruções
} else if (expressão booleana) {
   instruções
} else {
   instruções
}
```

- *Exemplo*

```
if ( ano < 0) {
   System.out.println("Não é um ano!");
} else if ( (ano%4==0 && ano%100!=0) || (ano%400==0) ) {
   System.out.println("É bissexto!");
} else {
   System.out.println("Não é bissexto!");
}
```

return

- *A palavra-chave return tem duas finalidades*
  - *Especifica o que um método irá retornar (se o método não tiver sido declarado com tipo de retorno void)*
  - *Causa o retorno imediato à linha de controle imediatamente posterior à chamada do método*
- *Exemplos de sintaxe:*

```
boolean método() {
    if (condição) {
        instrução;
        return true;
    }
    resto do método
    return false;
}
```

```
void método() {
    if (condição) {
        instrução;
        return;
    }
    mais coisas...
}
```

Este exemplo funciona como um if com else:

while e do-while

- Sintaxe

```
while (expresão booleana )
{
   instruções;
}
```

```
do
{
   instruções;
} while (expresão booleana );
```

- Exemplos

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

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

```
while ( true ) {
   if (obj.z == 0) {
     break;
   }
}
```

loop infinito!

for

- Sintaxe

```
for ( inicialização;
      expressões booleanas;
      passo da repetição )
{
   instruções;
}
```

```
for ( inicialização;
      expressões booleanas;
      passo da repetição )

  instrução_simples;
```

- Exemplos

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

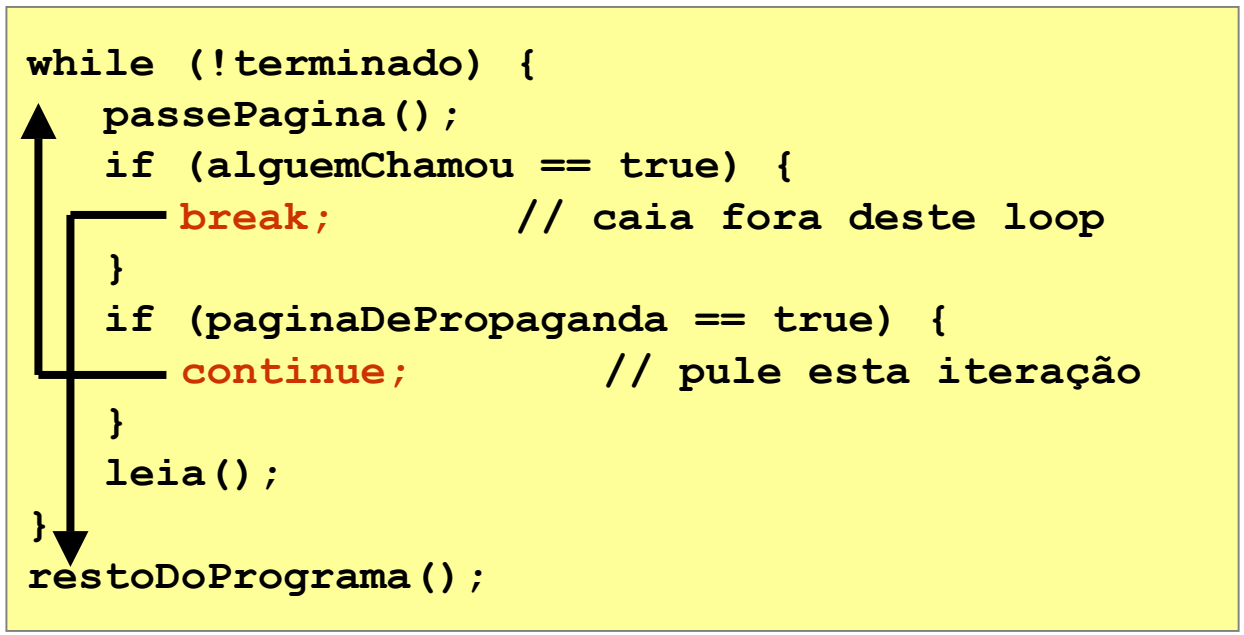
```
for ( int x = 0, int y = 25;
      x < 10 && (y % 2 == 0);
      x++, y = y - 1 ) {
   System.out.println (x + y);
}
```

```
for ( ; ; ) {
  if (obj.z == 0) {
    break;
  }
}
```

loop infinito!

break e continue

- **break**: interrompe a execução do bloco de repetição.
  - Continua com a próxima instrução, logo após o bloco.
- **continue**: interrompe a execução da iteração
  - Testa a condição e reinicia o bloco com a próxima iteração.



break e continue com rótulos

- ***break*** *e* ***continue*** *sempre atuam sobre o bloco de repetição onde são chamados*
- *Em blocos de repetição contidos em outros blocos, pode-se usar* ***rótulos*** *para fazer break e continue atuarem em blocos externos*
- *Os rótulos só podem ser usados antes de do, while e for*
- *As chamadas só podem ocorrer dentro de blocos de repetição. Exemplo:*

```
revista: while (!terminado) {
   for (int i = 10; i < 100; i += 10) {
      passePagina();
      if (textoChato) {
         break revista;
      }
   }
   maisInstrucoes();
}
restoDoPrograma();
```

break sem rótulo quebraria aqui!

- *Sintaxe:*

```
ident: do {...}
       ou
ident: while () {...}
       ou
ident: for () { ...}
```

switch (case)

- *Sintaxe*

qualquer expressão que resulte em valor inteiro (incl. char)

```
switch(seletor_inteiro) {
  case valor_inteiro_1 :
    instruções;
    break;
  case valor_inteiro_2 :
    instruções;
    break;
   ...
  default:
    instruções;
}
```

uma constante inteira (inclui char)

- *Exemplo*

char letra;

```
switch(letra) {
  case 'A' :
    System.out.println("A");
    break;
  case 'B' :
    System.out.println("B");
    break;
   ...
  default:
    System.out.println("?");
}
```

Exercícios

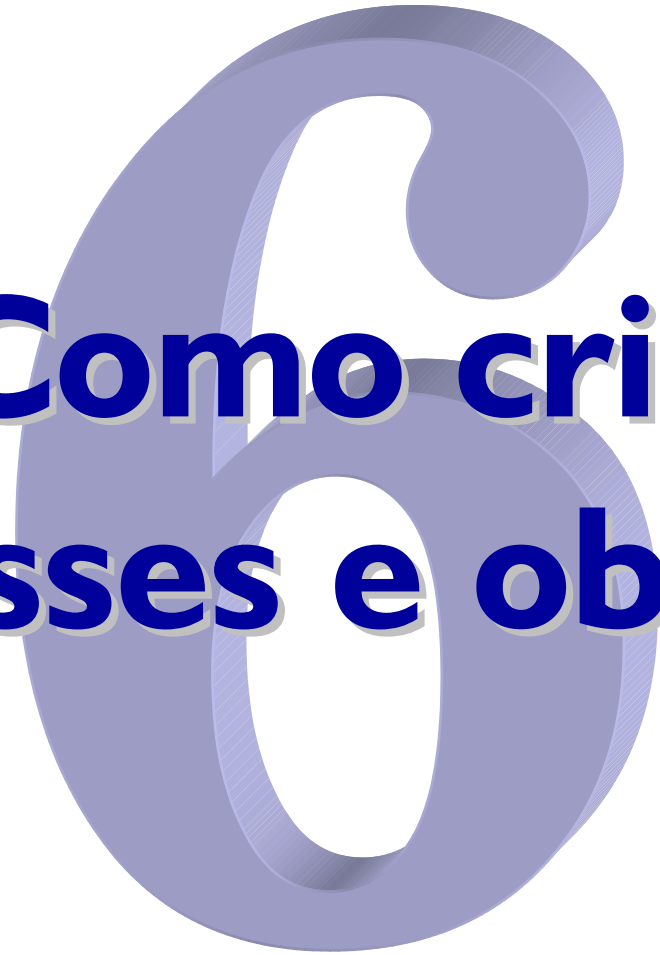
- *1. Escreva um programa Quadrados que leia um número da linha de comando e imprima o quadrado de todos os números entre 1 e o número passado.*
  - *Para converter de String para int, use:*
    ```
    int numero = Integer.parseInt("10");
    ```
- *2. Use o JOptionPane (veja documentação) e repita o exercício anterior recebendo os dados através da janela de entrada de dados (programa WinQuadrados)*
  - *Use* `JOptionPane.showInputDialog(string)` *de* `javax.swing` *para ler entrada de dados*
  - *Use* `JOptionPane.showMessageDialog(null, msg)` *para exibir a saída de dados*
- *3. Se desejar, use o build.xml do Ant disponível que executa Quadrados e WinQuadrados*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br



Como criar classes e objetos

Helder da Rocha
www.argonavis.com.br

Assuntos abordados

- Este módulo explora detalhes da construção de classes e objetos
  - Construtores
  - Implicações da herança
  - Palavras super e this, usadas como referências para o objeto corrente e a super classe
  - Instruções super() e this() usadas para chamar construtores durante a criação de objetos
  - Detalhes sobre a inicialização de objetos e possíveis problemas

Criação e destruição de objetos

- Para a criação de novos objetos, Java garante que cada classe tenha um construtor
  - O construtor default recebe zero argumentos
  - Faz apenas inicialização da superclasse
- Programador pode criar um construtor explicitamente e determinar suas operações de inicialização
  - Inicialização pela superclasse continua garantida
  - Construtor default deixa de existir
- Objetos são destruídos automaticamente pelo sistema, porém, sistema não faz finalização
  - Método finalize(), herdado de Object, teoricamente permite ao programador controlar a finalização de qualquer objeto
  - finalize() não funciona 95% das vezes - não use! Se precisar de finalização, coloque seu código em um bloco try {...} finally {...}

Construtores e sobrecarga

- Construtores default (sem argumentos) só existem quando não há construtores definidos explicitamente no código
  - A criação de um construtor explícito substitui o construtor fornecido implicitamente
- Uma classe pode ter vários construtores (isto se chama sobrecarga de nomes)
  - Distinção é feita pelo número e tipo de argumentos (ou seja, pela assinatura do construtor)
- A assinatura é a identidade do método. É pela assinatura que ele se distingue dos outros métodos. Consiste de
  - Tipo de retorno
  - Nome
  - Tipo de argumentos
  - Quantidade de argumentos

Sobrecarga de métodos

- Uma classe também pode ter vários métodos com o mesmo nome (sobrecarga de nomes de métodos)
  - Distinção é feita pela assinatura: tipo e número de argumentos, assim como construtores
  - Apesar de fazer parte da assinatura, o tipo de retorno não pode ser usado para distinguir métodos sobrecarregados
- Na chamada de um método, seus parâmetros são passados da mesma forma que em uma atribuição
  - Valores são passados em tipos primitivos
  - Referências são passadas em objetos
  - Há promoção de tipos de acordo com as regras de conversão de primitivos e objetos
  - Em casos onde a conversão direta não é permitida, é preciso usar operadores de coerção (cast)

Distinção de métodos na sobrecarga

- *Métodos sobrecarregados devem ser diferentes o suficiente para evitar ambigüidade na chamada*
- *Qual dos métodos abaixo ...*

```
int metodo (long x, int y) {...}
int metodo (int x, long y) {...}
```

- *... será chamado pela instrução abaixo?*

```
int z = metodo (5, 6);
```

- *O compilador detecta essas situações*

this()

- *Em classes com múltiplos construtores, que realizam tarefas semelhantes, this() pode ser usado para chamar outro construtor local, identificado pela sua assinatura (número e tipo de argumentos)*

```
public class Livro {
   private String titulo;
   public Livro() {
      titulo = "Sem titulo";
   }

   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

```
public class Livro {
   private String titulo;
   public Livro() {
      this("Sem titulo");
   }

   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

super()

- Todo construtor chama algum construtor de sua superclasse
  - Por default, chama-se o construtor sem argumentos, através do comando super() (implícito)
  - Pode-se chamar outro construtor, identificando-o através dos seus argumentos (número e tipo) na instrução super()
  - super(), se presente, deve sempre ser a primeira instrução do construtor (substitui o super() implícito)
- Se a classe tiver um construtor explícito, com argumentos, subclasses precisam chamá-lo diretamente
  - Não existe mais construtor default na classe

this e super

- A palavra this é usada para referenciar membros de um objeto
  - Não pode ser usada dentro de blocos estáticos (não existe objeto atual 'this' em métodos estáticos)
  - É obrigatória quando há ambiguidade entre variáveis locais e variáveis de instância
- super é usada para referenciar os valores originais de variáveis ou as implementações originais de métodos sobrepostos

```
class Numero {
    public int x = 10;
}
```

```
class OutroNumero extends Numero {
    public int x = 20;
    public int total() {
        return this.x + super.x;
    }
}
```

20

10

- Não confunda this e super com this() e super()
  - Os últimos são usados apenas em construtores!

Inicialização de instâncias

- O que acontece quando um objeto é criado usando *new NomeDaClasse()* ?
  - 1. Inicialização default de campos de dados (0, null, false)
  - 2. Chamada recursiva ao construtor da superclasse (até Object)

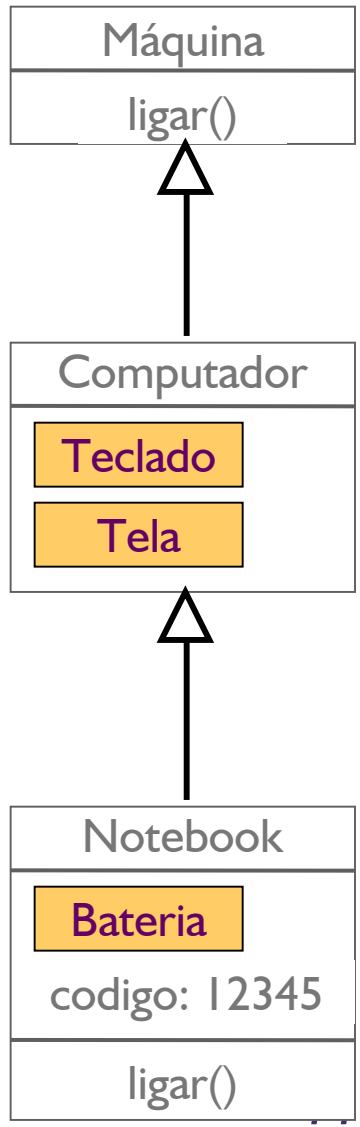
    2.1 Inicialização default dos campos de dados da superclasse (recursivo, subindo a hierarquia)

    2.2 Inicialização explicita dos campos de dados

    2.3 Execução do conteúdo do construtor (a partir de Object, descendo a hierarquia)
  - 3. Inicialização explícita dos campos de dados
  - 4. Execução do conteúdo do construtor

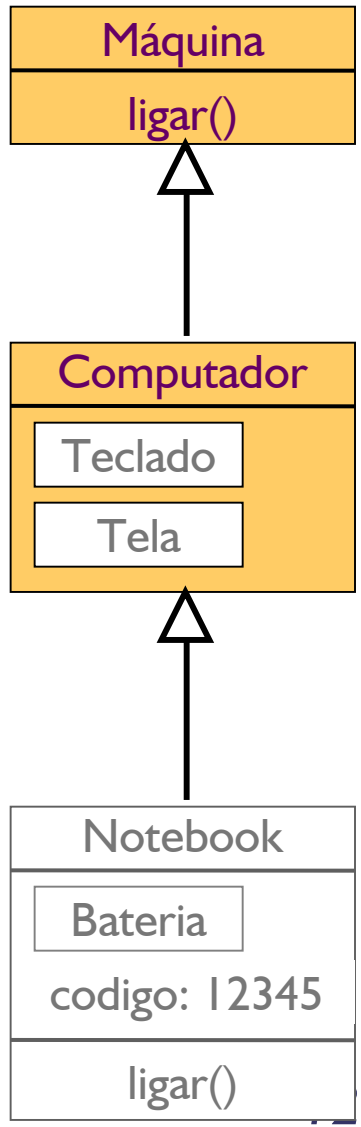
Exemplo (1)

```
class Bateria {
  public Bateria() {
    System.out.println("Bateria()");
  }
}

class Tela {
  public Tela() {
    System.out.println("Tela()");
  }
}

class Teclado {
  public Teclado() {
    System.out.println("Teclado()");
  }
}
```



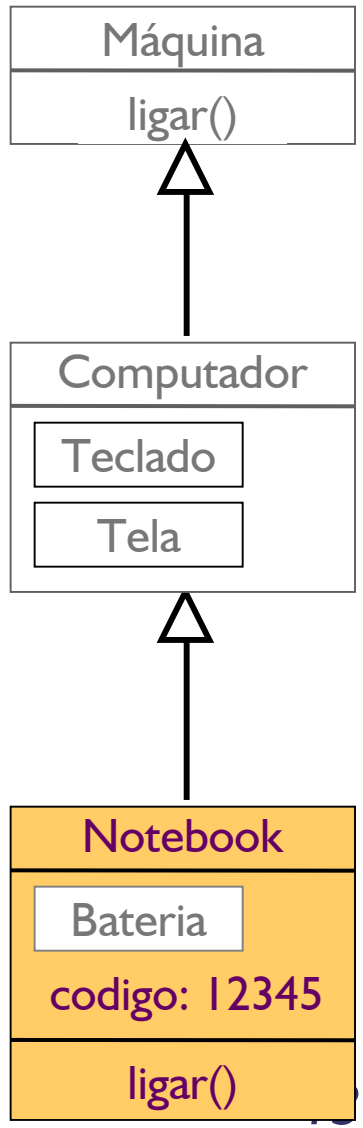
Exemplo (2)

```
class Maquina {
  public Maquina() {
     System.out.println("Maquina()");
     this.ligar();
  }
  public void ligar() {
    System.out.println("Maquina.ligar()");
  }
}

class Computador extends Maquina {
  public Tela tela = new Tela();
  public Teclado teclado = new Teclado();
  public Computador() {
     System.out.println("Computador()");
  }
}
```

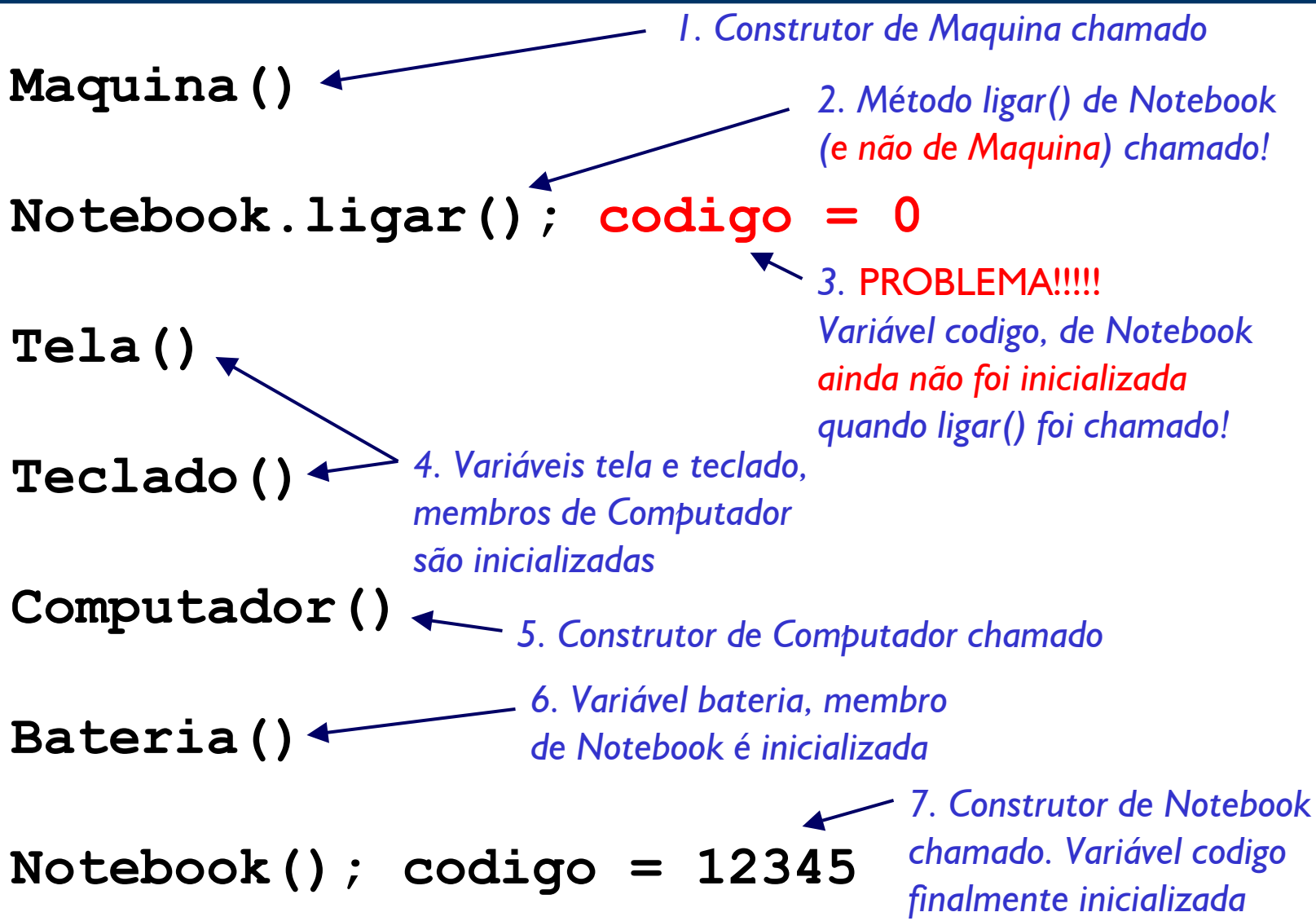


Exemplo (3)

```
class Notebook extends Computador {
  int codigo = 12345;
  public Bateria bateria = new Bateria();
  public Notebook() {
    System.out.print("Notebook(); " +
                     "codigo = "+codigo);
  }
  public void ligar() {
    System.out.println("Notebook.ligar();" +
                       " codigo = "+ codigo);
  }
}

public class Run {
  public static void main (String[] args) {
  new Notebook();
  }
}
```



Resultado de `new Notebook()`



Detalhes

N1. new Notebook() chamado
N2. variável código iniciaizada: 0
N3. variável bateria iniciaizada: null
N4. super() chamado (Computador)

C1. variável teclado iniciaizada: null
C2. variável tela iniclaizada: null
C3. super() chamado (Maquina)

M2. super() chamado (Object)

M2. Corpo de Maquina() executado:
println() e this.ligar()

C4: Construtor de Teclado chamado

Tk1: super() chamado (Object)

C5. referência teclado inicializada
C6: Construtor de Tela chamado

Te1: super() chamado (Object)

C7: referência tela inicializada
C8: Corpo de Computador()
executado: println()

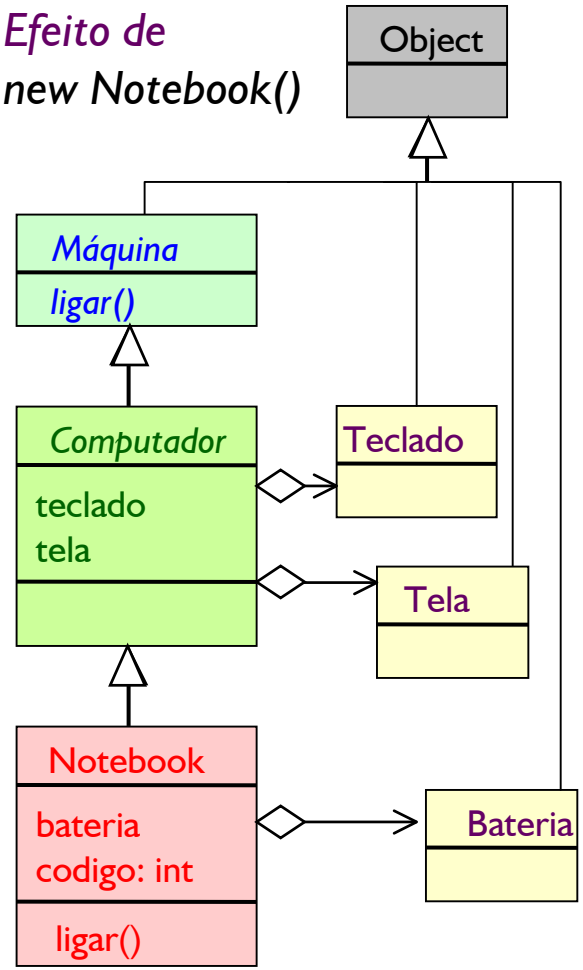
N5. Construtor de Bateria chamado

B1: super() chamado (Object)

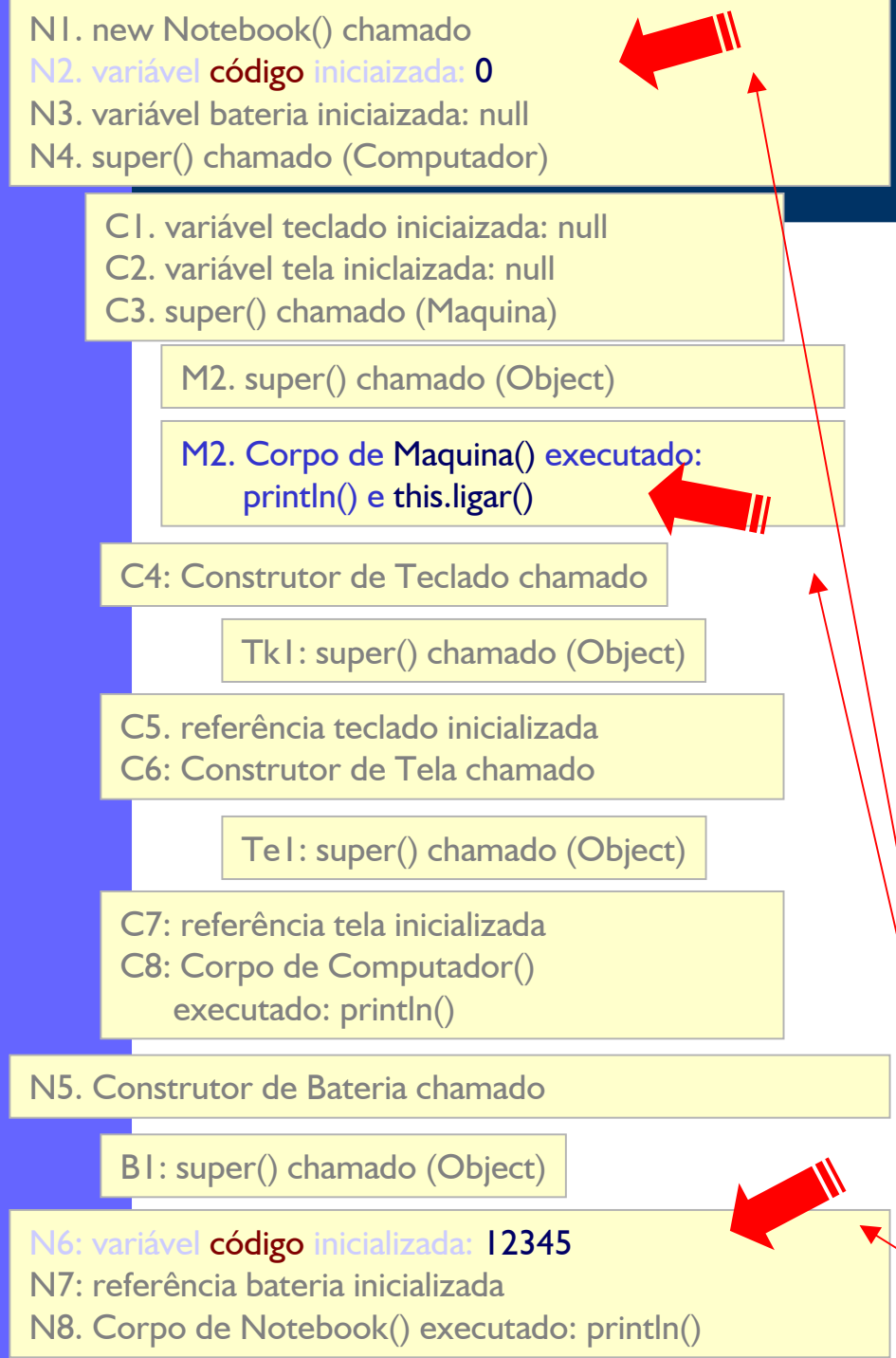
N6: variável código inicializada: 12345
N7: referência bateria inicializada
N8. Corpo de Notebook() executado: println()

O1. Campos inicializados
O2. Corpo de Object() executado

Efeito de
new Notebook()



Problemas com inicialização



- *método ligar() é chamado no construtor de Maquina, mas ...*
- *... a versão usada é a implementação em Notebook, que imprime o valor de código (e não a versão de Maquina como aparenta)*
- *Como código ainda não foi inicializado, valor impresso é 0!*

Preste atenção nos pontos críticos!

Como evitar o problema?

- Evite chamar métodos locais dentro de construtores
  - Construtor (qualquer um da hierarquia) sempre usa versão sobreposta do método
- Isto pode trazer resultados inesperados se alguém estender a sua classe com uma nova implementação do método que
  - Dependa de variáveis da classe estendida
  - Chame métodos em objetos que ainda serão criados (provoca NullPointerException)
  - Dependa de outros métodos sobrepostos
- Use apenas métodos finais em construtores
  - Métodos declarados com modificador final não podem ser sobrepostos em subclasses

Inicialização estática

- *Para inicializar valores estáticos, é preciso atuar logo após a carga da classe*
  - *O bloco 'static' tem essa finalidade*
  - *Pode estar em qualquer lugar da classe, mas será chamado antes de qualquer outro método ou variável*

```
class UmaClasse {
    private static Point[] p = new Point[10];
    static {
       for (int i = 0; i < 10; i++) {
          p[i] = new Point(i, i);
       }
    }
}
```

- *Não é possível prever em que ordem os blocos static serão executados, portanto: só tenha um!*

Exercício

- 1. Preparação
  - a) Crie um novo projeto para este módulo
  - b) Copie os arquivos Ponto e Circulo dos exercícios feitos no Capítulo 2 (se você não os fez, use os da solução)
- 2. Crie mais um construtor em Círculo
  - a) Crie um construtor default, que represente um círculo na origem (0,0) com raio unitário
  - b) Use this() em dois dos três construtores de Circulo
- 3. Crie uma classe Ponto3D que estenda Ponto.
  - Use, se necessário, a chamada correta de super() para permitir que a classe seja compilada
- 4. Altere TestaCirculo para testar as novas classes

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

7 Pacotes e Encapsulamento

Helder da Rocha
www.argonavis.com.br

Assuntos abordados neste módulo

- Este módulo explora pacotes em Java, iniciando pelos pacotes da própria API Java e terminando por mostrar como construir pacotes e guardá-los em um arquivo JAR
- Assuntos
  - API do Java 2 - visão geral dos principais pacotes
  - Classes do java.lang: visão geral das principais classes
  - Métodos de java.lang.Object que devem ser implementados
  - Pacotes em Java: como criar e como usar
  - Arquivos JAR para bibliotecas e executáveis

API Java 2

- A API do Java 2 consiste de classes distribuídas e organizadas em pacotes e subpacotes
- Pacotes básicos
  - java.lang: classes fundamentais - importado automaticamente
  - java.util: classes utilitárias
  - java.io: classes para entrada e saída
  - java.net: classes para uso em rede (TCP/IP)
  - java.sql: classes para acesso via JDBC
  - java.awt: interface gráfica universal nativa
  - java.text: internacionalização, transformação e formatação de texto
- Veja a documentação!

java.lang

- *É importante conhecer bem as classes deste pacote*
- *Interfaces*
  - *Cloneable*
  - *Runnable*
- *Classes*
  - *Boolean, Number (e subclasses), Character, Void*
  - *Class*
  - *Math*
  - *Object*
  - *Process, Thread, System e Runtime*
  - *String e StringBuffer*
  - *Throwable e Exception (e subclasses)*

java.lang.Object

- Raiz da hierarquia de classes da API Java
- Toda classe estende Object, direta ou indiretamente
  - Classes que não declaram estender ninguém, estendem Object diretamente

```
class Ponto {}
```

é o mesmo que

```
class Ponto extends Object {}
```

  - Classes que declaram estender outra classe, herdam de Object pela outra classe cuja hierarquia começa em Object
- Todos os métodos de Object estão automaticamente disponíveis para qualquer objeto
  - Porém, as implementações são default, e geralmente inúteis para objetos específicos

Exercício

- *1. Consulte no JavaDoc a página sobre java.lang.Object: veja os métodos.*
- *2. Utilize a classe TestaCirculo (exercícios anteriores) e teste os métodos equals() e toString() de Object*
  - *a) Crie dois Pontos e dois Circulos iguais (com mesmos raio e coordenadas)*
  - *b) Teste se os círculos e pontos são iguais usando "=="*
  - *c) Teste se são iguais usando* `equals()`*:*

```
if (p1.equals(p2)) {...}
```

  - *d) Imprima o valor retornado pelo método* `toString()` *de cada um dos objetos*

Classe java.lang.Object e interface Cloneable

- *Principais métodos de Object (todos os objetos têm)*
  - `public boolean equals(Object obj)`
  - `public String toString()`
  - `public int hashCode()`
  - `protected Object clone()`
    `throws CloneNotSupportedException`
  - `public void wait() throws InterruptedException`
  - `public void notify()`

por ser protected
TEM que ser sobreposto
para que possa ser usado
em qualquer classe

- *Cloneable*
  - *Usada para permitir que um objeto seja clonado. Não possui declaração de métodos*
  - *Como fazer:*

```
class SuaClasse implements Cloneable
class SuaClasse extends SupClasse implements Cloneable
```

Como estender Object

- *Há vários métodos em Object que devem ser sobrepostos pelas subclasses*
  - *A subclasse que você está estendendo talvez já tenha sobreposto esses métodos mas, alguns deles, talvez precisem ser redefinidos para que sua classe possa ser usada de forma correta*
- *Métodos que devem ser sobrepostos*
  - `boolean equals(Object o)`: *Defina o critério de igualdade para seu objeto*
  - `int hashCode()`: *Para que seu objeto possa ser localizado em Hashtables*
  - `String toString()`: *Sobreponha com informações específicas do seu objeto*
  - `Object clone()`: *se você desejar permitir cópias do seu objeto*

Como sobrepor equals()

- Determine quais os critérios (que propriedades do objeto) que podem ser usados para dizer que um objeto é igual a outro
  - O raio, em um objeto Círculo
  - O número de série, em um objeto genérico
  - O nome, sobrenome e departamento, para um empregado
  - A chave primária, para um objeto de negócio
- Implemente o equals(), testando essas condições e retornando true apenas se forem verdadeiras (false, caso contrário)
  - Verifique que a assinatura seja igual à definida em Object

instanceof e exemplo com equals()

- *instanceof* é um operador usado para comparar uma referência com uma classe
  - A expressão será true se a referência for do tipo de uma classe ou subclasse testada e false, caso contrário
- Exemplo: sobreposição de equals()

```
class Point {
  private int x, y;
  public boolean equals(Object obj) {
    if (obj instanceof Point) {
      Point ponto = (Point) obj;
      if (ponto.x == this.x && ponto.y == this.y) {
        return true;
      }
    }
    return false;
  }
}
```

Como sobrepor toString()

- *toString() deve devolver um String que possa representar o objeto quando este for chamado em uma concatenação ou representado como texto*
  - *Decida o que o toString() deve retornar*
  - *Faça chamadas super.toString() se achar conveniente*
  - *Prefira retornar informações que possam identificar o objeto (e não apenas a classe)*
  - *toString() é chamado automaticamente em concatenções usando a referência do objeto*

Exercício

- 1. Sobreponha toString() em Ponto e Circulo
  - Faça toString retornar informações que representem o objeto quando ele for impresso
  - Se desejar, faça toString retornar o conteúdo de imprime() (se você implementou este método)
  - Teste sua implementação imprimindo os objetos diretamente no System.out.println() sem chamar toString() explicitamente
- 2. Sobreponha equals() em Circulo
  - Um circulo é igual a outro se estiver no mesmo ponto na origem e se tiver o mesmo raio.
  - Rode a classe de teste e veja se o resultado é o esperado
  - Implemente, se necessário, equals() em Ponto também!

Como sobrepor hashCode()

- *hashCode()* deve devolver um número inteiro que represente o objeto
  - Use uma combinação de variáveis, uma chave primária ou os critérios usados no equals()
  - Número não precisa ser único para cada objeto mas dois objetos iguais devem ter o mesmo número.
  - O método *hashCode()* é chamado automaticamente quando referências do objeto forem usadas em coleções do tipo hash (Hashtable, HashMap)
  - *equals()* é usado como critério de desempate, portanto, se implementar hashCode(), *implemente equals() também*.

Como sobrepor clone()

- *clone() é chamado para fazer cópias de um objeto*

```
Circulo c = new Circulo(4, 5, 6);
Circulo copia = (Circulo) c.clone();
```

cast é necessário porque clone() *retorna Object*

- *Se o objeto apenas contiver tipos primitivos como seus campos de dados, é preciso*
  1. *Declarar que a classe implementa Cloneable*
  2. *Sobrepor clone() da seguinte forma:*

é preciso sobrepor clone() *porque ele é definido como protected*

```
public Object clone() {
   try {
     return super.clone();
   } catch (CloneNotSupportedException e) {
     return null;
   }
}
```

Como sobrepor clone() (2)

- *Se o objeto contiver campos de dados que são referências a objetos, é preciso fazer cópias desses objetos também*

```
public class Circulo {
 private Point origem;
 private double raio;
 public Object clone() {
  try {
    Circulo c = (Circulo)super.clone();
    c.origem = (Point)origem.clone(); // Point clonável!
    return c;
  } catch (CloneNotSupportedException e) {return null;}
 }
}
```

clone() é implementação do padrão de projeto Prototype

A classe java.lang.Math

- A classe Math é uma classe final (não pode ser estendida) com construtor private (não permite a criação de objetos)
- Serve como repositório de funções e constantes matemáticas
- Para usar, chame a constante ou função precedida do nome da classe:

```
double distancia = Math.sin(0.566);
int sorte = (int) (Math.random() * 1000);
double area = 4 * Math.PI * raio;
```

- Consulte a documentação sobre a classe Math e explore as funções disponíveis

Funções matemáticas

- *Algumas funções úteis de java.lang.Math (consulte a documentação para mais detalhes)*
  - `double random()`: *retorna número aleatório entre 0 e 1*
  - `int floor(double valor)`: *trunca o valor pelo decimal (despreza as casas decimais)*
  - `int ceil(double valor)`: *retorna o próximo valor inteiro (arredonda para cima)*
  - `int round(double valor)`: *arredonda valor*
  - `double pow(double valor, double valor)`: *expoente*
  - `double sqrt(double valor)`: *raiz quadrada*
  - `double sin(double valor)`, `double cos(double valor)`, `double tan(double valor)`: *calculam seno, cosseno e tangente respectivamente*
  - *Veja ainda:* `exp()`, `log()`, `ln()`, `E`, `PI`, *etc.*

As classes empacotadoras (wrappers)

- Classes que servem para embutir tipos primitivos para que possam ser manipulados como objetos
- Exemplo:
  - Um vetor de tipos Object pode conter qualquer objeto mas não tipos primitivos

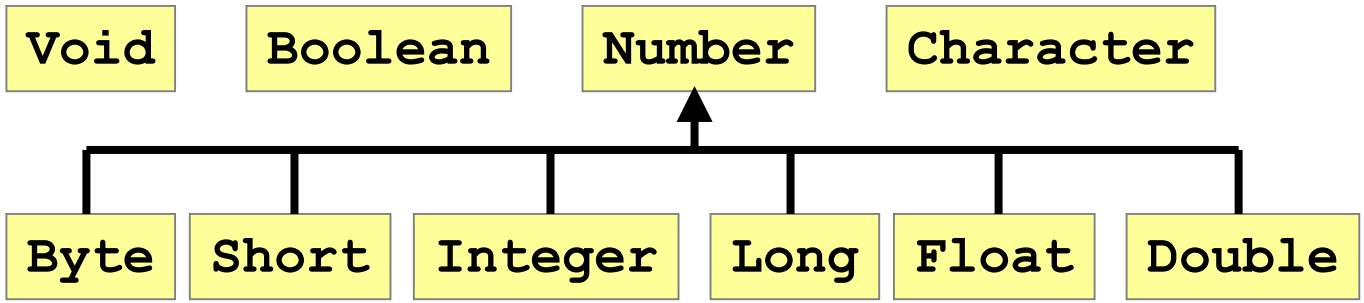
```
Object[] vetor = new Object[5];
vetor[0] = new Ponto(); // OK
vetor[1] = 15; // Errado!!
```

  - Solução: colocar os tipos dentro de wrappers

```
vetor[1] = new Integer(15);
```

- Construtores aceitam literais ou Strings
- Há métodos para realizar conversões entre tipos e de tipos para Strings e vice-versa
- Padrão de projeto: Adapter pattern

Classes empacotadoras



- *Criação e conversão*

```
Integer n = new Integer(15);
Integer m = Integer.valueOf(15);
int q = n.intValue();
```

Mais eficiente

- *Para conversão de String em tipos primitivos*

```
int i = Integer.parseInt("15");
double i = Double.parseDouble("15.9");
boolean b =
        (new Boolean("true")).booleanValue();
```

Controle e acesso ao sistema

- *Classe System dá acesso a objetos do sistema operacional*
  - `System.out` - *saída padrão (java.io.PrintStream)*
  - `System.err` - *saída padrão de erro (java.io.PrintStream)*
  - `System.in` - *entrada padrão (java.io.InputStream)*
- *Runtime e Process permitem controlar processos externos (processos do S.O.)*

```
Runtime r = System.getRuntime();
Process p =
  r.exec("c:\program~1\micros~1\msie.exe");
```

- *Thread e Runnable lidam com processos internos (processos da aplicação - threads)*

Exceções

- Classes descendentes de java.lang.Throwable representam situações de erro
  - Erros graves irrecuperáveis (descendentes da classe java.lang.Error)
  - Exceções de tempo de execução (descendentes da classe java.lang.Exception)
- São usadas em blocos try-catch
  - Serão tratadas em capítulo a parte
- São classes, portanto...
  - ... podem ser estendidas
  - ... podem ser usadas para criar objetos via construtor
  - ... podem ter métodos chamados

Identificação em tempo de execução

- A classe *Class* contém métodos que permitem enxergar a interface pública de uma classe
  - Saber quais métodos existem para serem chamados
  - Saber quais os parâmetros e tipos dos parâmetros dos métodos
  - Saber quais os construtores e campos públicos
- Como obter um objeto class (ex: classe MinhaClasse):

```
Class cobj = instancia.getClass();// Quando não se sabe a classe
Class cobj = MinhaClasse.class; // Quando não se tem instância
```

- Com essas informações, é possível carregar uma classe compilada, em tempo de execução, e usá-la
  - Há métodos para carregar uma classe pelo nome, passado como String (`Class cobj = Class.forName();`)
  - Há métodos (`cobj.newInstance();`) para criar objetos a partir de objeto Class obtido a partir do nome da classe
- Reflection API!

Outros pacotes

- Outros pacotes mais importantes da API Java 2 serão explorados em aulas futuras
- Use a documentação e experimente utilizar classes de algum pacote de seu interesse
- Sugestões
  - java.text: descubra como formatar texto para impressão (por exemplo, 2 decimais e 3 dígitos fracionários)
  - java.util: descubra como usar datas com Calendar, GregorianCalendar e Date. Use as classes de java.text para formatar as datas corretamente
  - java.io: descubra como abrir um arquivo
  - java.sql: descubra como conectar-se a um banco

java.util.Date

- Date representa uma data genérica representada pela contagem de milissegundos desde 1/1/1970.
  - Não representa uma data de calendário (para isto existe java.util.Calendar - pesquise da documentação!)
  - Não representa data ou hora do calendário ocidental (para isto existe java.util.GregorianCalendar)
- Use Date para obter o momento atual

```
Date agora = new Date();
```

- Ou para criar momentos com base na contagem de milissegundos

```
Date ontem =
     new Date(agora.getTime() - 86400000);
Date intervalo =
     agora.getTime() - inicio.getTime();
```

Exercícios

- 1. Implemente os seguintes métodos para Circulo e Point:
  - hashCode()
  - clone()
- 2. Crie um vetor de Object com 5 elementos e inclua dentro dele um inteiro, um char, um double, um Ponto e uma Data
  - Depois escreva código recuperando cada valor original
- 3. Teste os objetos criando cópias com clone() e imprimindo o hashCode()
  - hashCode() na verdade, terá maior utilidade quando usado dentro de HashMaps (capítulo sobre coleções)

Pacotes: import

- Um pacote é uma coleção de classes e interfaces Java, agrupadas
  - O pacote faz parte do nome da classe: Uma classe NovaYork do pacote simulador.cidades pode ser usada por outra classe (de pacote diferente) apenas se for usado o nome completo simulador.cidades.NovaYork
  - Toda classe pertence a um pacote: Se a classe não tiver declaração package, ela pertence ao pacote default, que, durante a execução, corresponde à raiz do Classpath.
- import, pode ser usado, para compartilhar os espaços de nomes de pacotes diferentes (assim, não será preciso usar nomes completos)

Classpath

- O **Classpath** é uma propriedade do sistema que contém as localidades onde o JRE irá procurar classes. Consiste de
  - 1. JARs nativos do JRE (API Java 2)
  - 2. Extensões do JRE (subdiretórios $JAVA_HOME/jre/lib/classes e $JAVA_HOME/jre/lib/ext)
  - 3. Lista de caminhos definidos na variável de ambiente CLASSPATH e/ou na opção de linha de comando -classpath (-cp) da aplicação java.
- A ordem acima é importante
  - Havendo mais de uma classe com mesmo pacote/Nome somente a primeira classe encontrada é usada. Outras são ignoradas
  - Há risco de conflitos. API nova sendo carregada depois de antiga pode resultar em classes novas chamando classes antigas!
  - A ordem dos caminhos na variável CLASSPATH (ou opção -cp) também é significativa.

Variável CLASSPATH e -cp

- Em uma instalação típica, **CLASSPATH** contém apenas "."
  - Pacotes iniciados no diretório atual (onde o interpretador java é executado) são encontrados (podem ter suas classes importadas)
  - Classes localizadas no diretório atual são encontradas.
- Geralmente usada para definir caminhos para **uma** aplicação
  - Os caminhos podem ser diretórios, arquivos ZIP ou JARs
  - Pode acrescentar novos caminhos mas não pode remover caminhos do Classpath do JRE (básico e extensões)
- A opção **-cp** (-classpath) substitui as definições em CLASSPATH
- Exemplo de definição de CLASSPATH
  - no DOS/Windows

```
set CLASSPATH=extras.jar;.;c:\progs\java
java -cp %CLASSPATH%;c:\util\lib\jsw.zip gui.Programa
```

  - no Unix (sh, bash)

```
CLASSPATH=extras.jar:.:/home/mydir/java
export CLASSPATH
java -classpath importante.jar:$CLASSPATH Programa
```

Classpath do JRE

- *Colocar JARs no subdiretório **ext** ou **classes** e pacotes no diretório classes automaticamente os inclui no Classpath para todas as aplicações da JVM*
  - *Carregados antes das variáveis CLASSPATH e -cp*
  - *Evite usar: pode provocar conflitos. Coloque nesses diretórios apenas os JARs e classes usados em todas suas aplicações*
- *Exemplo: suponha que o Classpath seja*
  - *Classpath JRE:* `%JAVA_HOME%\jre\lib\rt.jar;`
  - *Classpath Extensão JRE:* `%JAVA_HOME%\jre\lib\ext\z.jar`
  - *Variável de ambiente CLASSPATH:* `.;c:\programas;`

  e que uma classe, localizada em c:\exercicio seja executada. Se esta classe usar a classe arte.fr.Monet o sistema irá procurá-la em

1. `%JAVA_HOME%\jre\lib\rt.jar\arte\fr\Monet.class`
2. `%JAVA_HOME%\jre\lib\ext\z.jar\arte\fr\Monet.class`
3. `c:\exercício\arte\fr\Monet.class`
4. `c:\programas\arte\fr\Monet.class`

Como criar e usar um pacote

- Para **criar** um pacote é preciso
  1. Declarar o nome do pacote em cada unidade de compilação
  2. Guardar a classe compilada em uma localidade (caminho) compatível com o pacote declarado
     - O "caminho de pontos" de um pacote, por exemplo, simulador.cidade.aeroporto corresponde a um caminho de diretórios simulador/cidade/aeroporto
- Para **usar** um pacote (usar suas classes) é preciso
  1. Colocar a raiz do pacote (pasta "simulador", por exemplo) no Classpath
  2. Importar as classes do pacote (ou usar suas classes pelo nome completo): A instrução import é opcional

Arquivos JAR

- Geralmente, aplicações Java são distribuídas em arquivos **JAR**
  - São extensões do formato ZIP
  - armazenam pacotes e preservam a hierarquia de diretórios
- Para **usar** um JAR, é preciso incluí-lo no Classpath
  - via CLASSPATH no contexto de execução da aplicação, ou
  - via parâmetro -classpath (-cp) do interpretador Java, ou
  - copiando-o para $JAVA_HOME/jre/lib/ext
- Para **criar** um JAR

```
jar cvf classes.jar C1.class C2.class xyz.gif abc.html
jar cf mais.jar -C raiz_onde_estao_pacotes/ .
```

- Para **abrir** um JAR

```
jar xvf classes.jar
```

- Para **listar o conteúdo** de um JAR

```
jar tvf classes.jar
```

Criação de bibliotecas

- *Uma **biblioteca** ou **toolkit** é um conjunto de classes base (para criar objetos ou extensão) ou classes utilitárias (contendo métodos úteis)*
- *Para **distribuir** uma biblioteca*
  - *Projete, crie e compila as classes (use pacotes)*
  - *Guarde em arquivo JAR*
- *Para **usar** a biblioteca*
  - *Importe as classes da sua biblioteca em seus programas*
  - *Rode o interpretador tendo o JAR da sua biblioteca no Classpath*

Uma biblioteca (simples)

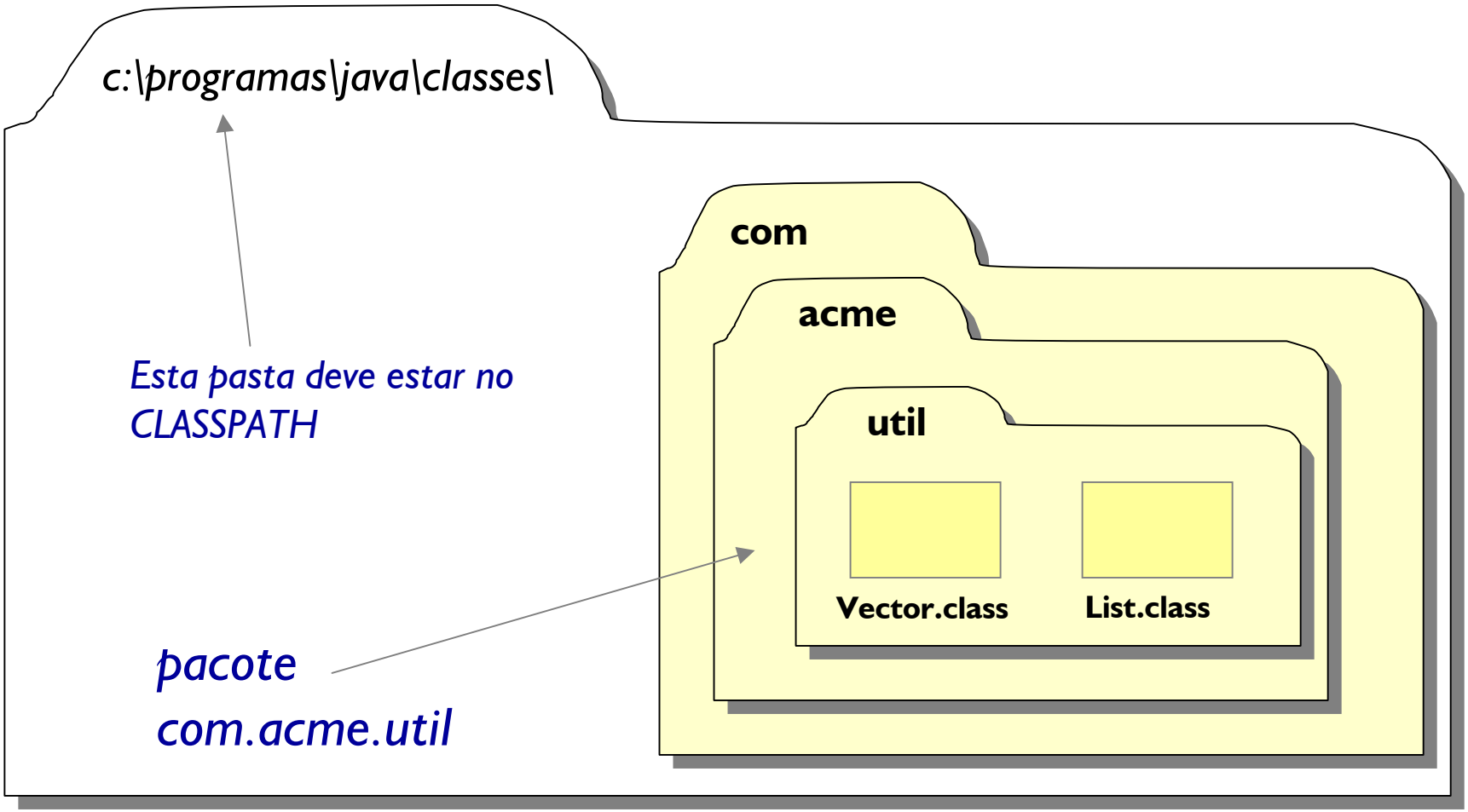
List.java

```
package com.acme.util;
public class List {
  public List() {
    System.out.println("com.acme.util.List");
  }
}
```

Vector.java

```
package com.acme.util;
public class Vector {
  public Vector() {
    System.out.println("com.acme.util.Vector");
  }
}
```

Onde armazenar

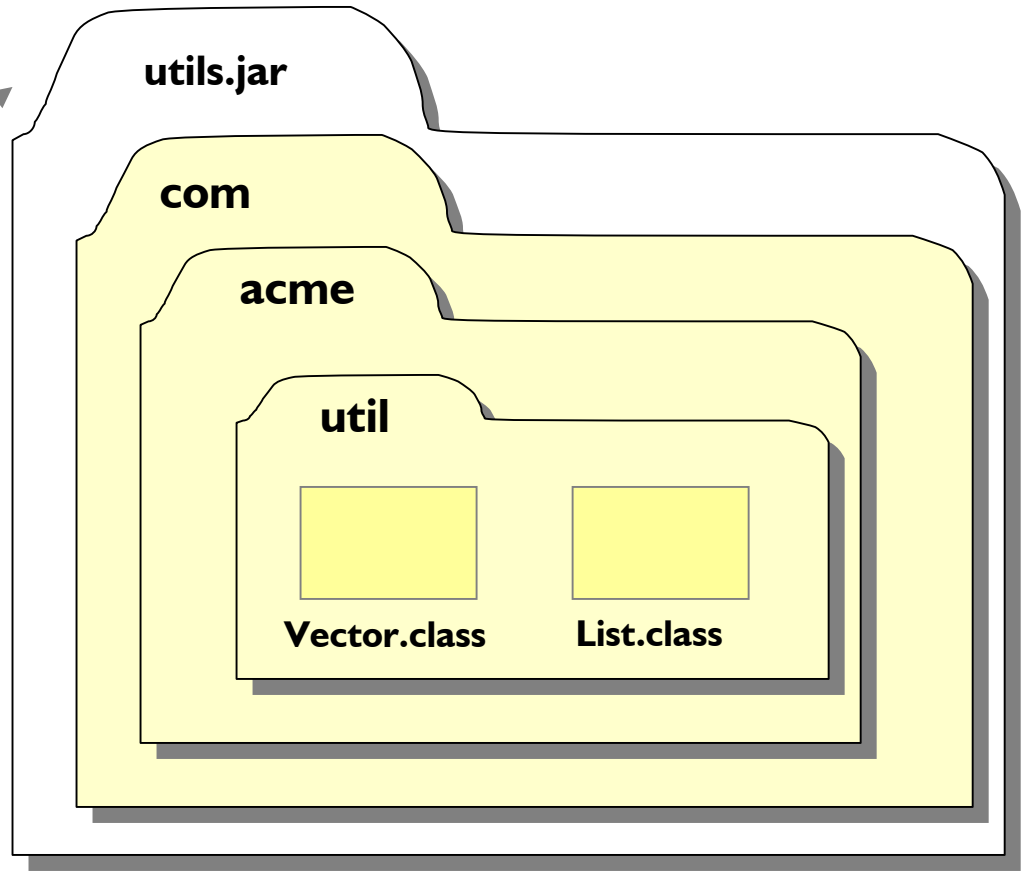


Ou em um JAR

O caminho até a localização do JAR deve estar no CLASSPATH (pode ser especificado no momento da execução:

```
java -classpath
  %CLASSPATH%;utils.jar
  NomeProgExecutavel
```

JAR também pode ser jogado no diretório ext do JRE



Como usar a biblioteca

- *Programa que usa biblioteca pode estar em qualquer lugar*
  - *É preciso que caminho até a biblioteca (pasta onde começa pacote ou caminho até JAR) esteja definido no Classpath*

```
CLASSPATH=%CLASSPATH%;c:\programas\java\classes
CLASSPATH=%CLASSPATH%;c:\jars\utils.jar
```

```
// Programa que usa a biblioteca
import com.acme.util.*;
public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List m = new List();
  }
}
```

  - *Pode também usar*

```
java -cp %CLASSPATH%;c:\...\java\classes LibTest
java -cp %CLASSPATH%;c:\jars\utils.jar LibTest
```

Como criar um executável

- *Todo JAR possui um arquivo chamado **Manifest.mf** no subdiretório /META-INF.*
  - *Lista de pares Nome: atributo*
  - *Serve para incluir informações sobre os arquivos do JAR, CLASSPATH, classe Main, etc.*
  - *Se não for indicado um arquivo específico, o sistema gerará um Manifest.mf default (vazio)*
- *Para tornar um JAR executável, o Manifest.mf deve conter a linha:*
  - `Main-Class: nome.da.Classe`
- *Crie um arquivo de texto qualquer com a linha acima e monte o JAR usando a opção -manifest:*

```
jar cvfm arq.jar arquivo.txt -C raiz .
```

- *Para executar o JAR em linha de comando:*

```
java -jar arq.jar
```

Colisões entre classes

- *Se houver, no mesmo espaço de nomes (Classpath + pacotes), duas classes com o mesmo nome, não será possível usar os nomes das classes de maneira abreviada*
- *Exemplo: classes com.acme.util.List e java.util.List*

```
import com.acme.util.*;
import java.util.*;
class Xyz {
  // List itens; // Ñ COMPILA!
  // java.util.List lista;
}
```

```
import com.acme.util.*;

class Xyz {
  List itens; //com.acme.util.List
  java.util.List lista;
}
```

Exercício

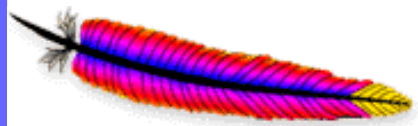
- 1. Copie a aplicação contendo os Pontos e Circulos para um novo projeto
  - a) Declare o Ponto e o Circulo como pertencentes a um pacote chamado graficos
  - b) Importe esses pacotes de TestaCirculos
  - c) Execute a aplicação
  - d) Empacote tudo em um JAR
  - e) Torne o JAR executável (para usar java -jar arquivo.jar)
- 2. Acrescente o alvo jar no seu build.xml para que seja possível gerar o JAR e Manifest automaticamente:
  - Veja exemplo no capítulo 8!

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br



The Jakarta Project
http://jakarta.apache.org

Gerenciamento de projetos com o Apache Ant

Helder da Rocha (helder@acm.org)

Sobre este módulo

- Este módulo apresenta o Jakarta Ant - ferramenta importante para gerenciar projetos
  - Qualquer aplicação Java com mais que meia dúzia de classes ou organizada em pacotes deve ser organizada como um projeto
  - É uma boa prática manter scripts para automatizar procedimentos de desenvolvimento (compilar, testar, criar documentação, gerar JARs, etc.)
- O material disponível é muito extenso para tratamento detalhado neste curso
  - Abordagem será superficial, mas use-o como referência durante o curso

Ant: o que é?

- Ferramenta para construção de aplicações
  - Implementada em Java
  - Baseada em roteiros XML
  - Extensível (via scripts ou classes)
  - 'padrão' do mercado
  - Open Source (Grupo Apache, Projeto Jakarta)
- Semelhante a make, porém
  - Mais simples e estruturada (XML)
  - Mais adequada a tarefas comuns em projetos Java
  - Independente de plataforma

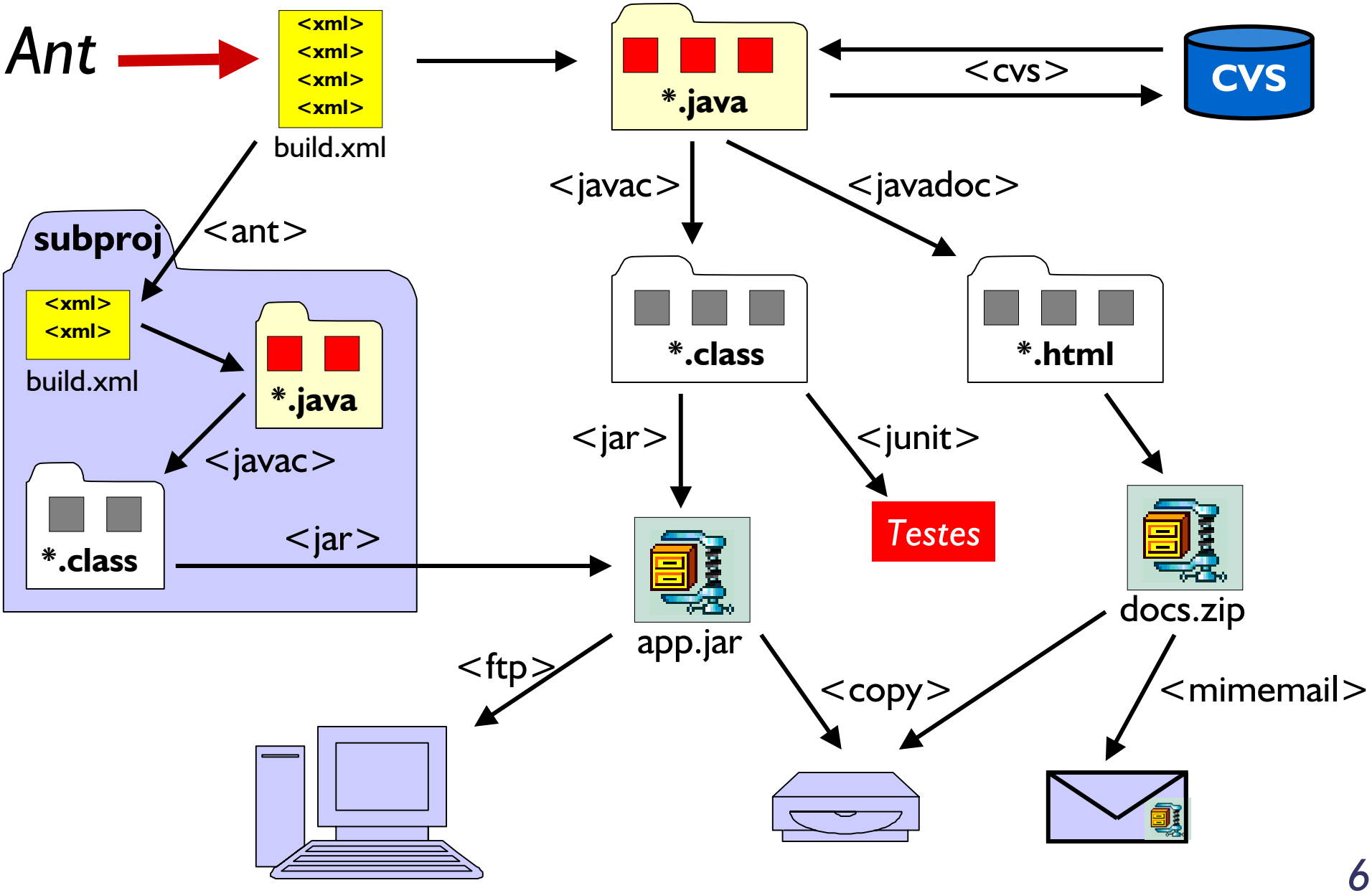
Para que serve?

- Para montar praticamente qualquer aplicação Java que consista de mais que meia dúzia de classes; Aplicações
  - distribuídas em pacotes
  - que requerem a definição de classpaths locais, e precisam vincular código a bibliotecas (JARs)
  - cuja criação/instalação depende de mais que uma simples chamada ao javac. Ex: RMI, CORBA, EJB, servlets, JSP,...
- Para automatizar processos frequentes
  - Javadoc, XSLT, implantação de serviços Web e J2EE (deployment), CVS, criação de JARs, testes, FTP, email

Como funciona?

- Ant executa roteiros escritos em XML: 'buildfiles'
- Cada projeto do Ant possui um buildfile
  - subprojetos podem ter, opcionalmente, buildfiles adicionais chamados durante a execução do primeiro
- Cada projeto possui uma coleção de alvos
- Cada alvo consiste de uma seqüência de tarefas
- Exemplos de execução
  - `ant`
    - procura build.xml no diretório atual e roda alvo default
  - `ant -buildfile outro.xml`
    - executa alvo default de arquivo outro.xml
  - `ant compilar`
    - roda alvo 'compilar' e possíveis dependências em build.xml

Como funciona (2)



Buildfile

- *O buildfile é um arquivo XML:* **build.xml** *(default)*
- *Principais elementos*
  - `<project default="alvo_default">`
    - *Elemento raiz (obrigatório): define o projeto.*
  - `<target name="nome_do_alvo">`
    - *Coleção de tarefas a serem executadas em seqüência*
    - *Deve haver pelo menos um <target>*
  - `<property name="nome" value="valor">`
    - *pares nome/valor usados em atributos dos elementos do* build.xml *da forma* `${nome}`
    - *propriedades também podem ser definidas em linha de comando (*`-Dnome=valor`*) ou lidas de arquivos externos (atributo* `file`*)*
  - *tarefas (mais de 130) - dentro dos alvos.*
    - `<javac>`, `<jar>`, `<java>`, `<copy>`, `<mkdir>`, ...

Buildfile (2)

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- Compila diversos arquivos .java -->
<project default="compile" basedir=".">
  <property name="src.dir" value="${basedir}/src" />
  <property name="build.dir" value="build" />
  <target name="init">
    <echo> Criando diretório </echo>
    <mkdir dir="${build.dir}" />
  </target>
  <target name="compile" depends="init"
          description="Compila os arquivos-fonte">
    <javac srcdir="${src.dir}" destdir="${build.dir}">
      <classpath>
        <pathelement location="${build.dir}" />
      </classpath>
    </javac>
  </target>
</project>
```

Propriedades

Alvos

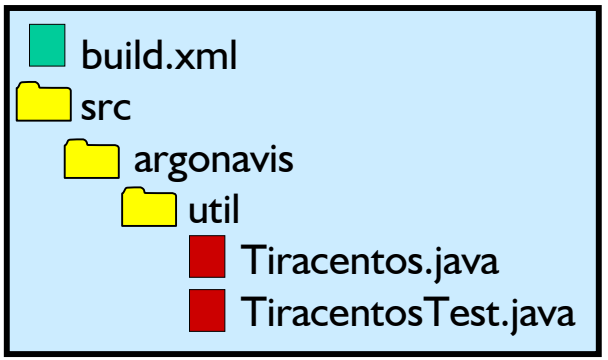
Tarefas

Exemplo

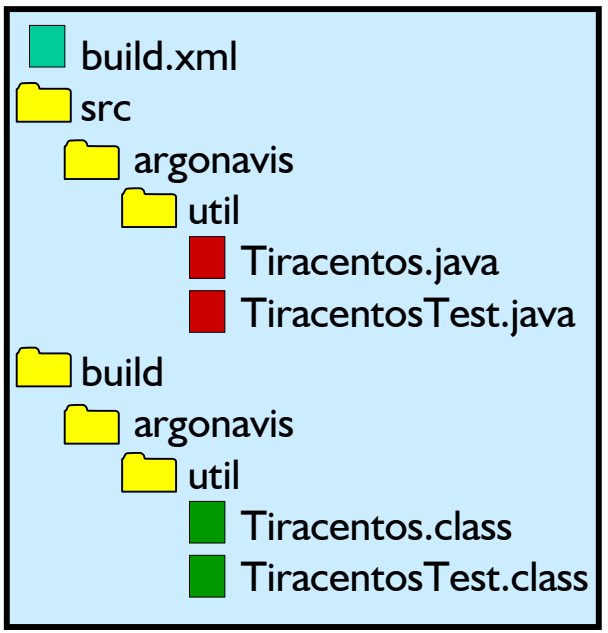
- *Executando buildfile da página anterior*

```
C:\usr\palestra\antdemo> ant
Buildfile: build.xml

init:
     [echo]  Criando diretório
     [mkdir] Created dir:
     C:\usr\palestra\antdemo\build

compile:
     [javac] Compiling 2 source files to
     C:\usr\palestra\antdemo\build

BUILD SUCCESSFUL
Total time: 4 seconds
C:\usr\palestra\antdemo>
```

ANTES



DEPOIS



Propriedades

- *Podem ser definidas com* **<property>**

```
<property name="app.nome" value="jmovie" />
```

- *Podem ser carregadas de um arquivo*

```
<property file="c:/conf/arquivo.conf" />
```

```
app.ver=1.0
docs.dir=c:\docs\
codigo=15323
```

arquivo.conf

- *Podem ser passadas na linha de comando*

```
c:\> ant -Dautor=Wilde
```

- *Para recuperar o valor, usa-se* ${nome}

```
<jar destfile="${app.nome}-${app.ver}.jar"/>
<echo message="O autor é ${autor}" />
<mkdir dir="build${codigo}" />
```

Propriedades especiais

- **<tstamp>**: Grava um instante
  - A hora e data podem ser recuperados como propriedades
    - `${TSTAMP}` *hhmm* *1345*
    - `${DSTAMP}` *aaaammdd* *20020525*
    - `${TODAY}` *dia mes ano* *25 May 2002*
  - Novas propriedades podem ser definidas, locale, etc.
  - Uso típico: `<tstamp/>`
- **<property** environment="env"**>**: Propriedade de onde se pode ler variáveis de ambiente do sistema
  - Dependende de plataforma

```
<target name="init">
  <property environment="env"/>
  <property name="j2ee.home"
            value="env.J2EE_HOME" />
</target>
```

O que se pode fazer com Ant?

- Compilar.
  <javac>, <csc>
- Gerar documentação
  <javadoc>, <junitreport>, <style>, <stylebook>
- Executar programas
  <java>, <apply>, <exec> <ant>, <sql>
- Testar unidades de código
  <junit>
- Empacotar e comprimir
  <jar>, <zip>, <tar>, <war>, <ear>, <cab>
- Expandir, copiar, instalar
  <copy>, <delete>, <mkdir>, <unjar>, <unwar>, <untar>, <unzip>
- Acesso remoto
  <ftp>, <telnet>, <cvs>, <mail>, <mimemail>
- Montar componentes
  <ejbc>, <ejb-jar>, <rmic>
- Criar novas tarefas
  <taskdef>
- Executar roteiros e sons
  <script>, <sound>

Compilação e JAR

- **<javac>**: Chama o compilador Java

```
<javac srcdir="dirfontes" destdir="dirbuild" >
    <classpath>
        <pathelement path="arquivo.jar" />
        <pathelement path="/arquivos" />
    </classpath>
    <classpath idref="extra" />
</javac>
```

- **<jar>**: Monta um JAR

```
<jar destfile="bin/programa.jar">
   <manifest>
      <attribute name="Main-class"
                 value="exemplo.main.Exec"/>
   </manifest>
   <fileset dir="${build.dir}"/>
</jar>
```

Tarefas do sistema de arquivos

- **<mkdir>**: *cria diretórios*

```
<mkdir dir="diretorio" />
```

- **<copy>**: *copia arquivos*

```
<copy todir="dir" file="arquivo" />
<copy todir="dir">
    <fileset dir="fonte"
             includes="*.txt" />
</copy>
```

- **<delete>**: *apaga arquivos*

```
<delete file="arquivo />
<delete dir="diretorio"/>
```

Geração de documentação

- **<javadoc>**: Gera documentação do código-fonte.
  - *Alvo abaixo gera documentação e exclui classes que contém 'Test.java'*

```
<target name="generate-docs">
  <mkdir dir="docs/api"/>
  <copy todir="tmp">
    <fileset dir="${src.dir}">
      <include name="**/*.java" />
      <exclude name="**/**Test.java" />
    </fileset>
  </copy>
  <javadoc destdir="docs/api"
           packagenames="argonavis.*"
           sourcepath="tmp" />
  <delete dir="tmp" />
</target>
```

Tipos de dados: arquivos e diretórios

- **<fileset>**: árvore de arquivos e diretórios
  - Conteúdo do conjunto pode ser reduzido utilizando elementos <include> e <exclude>
  - Usando dentro de tarefas que manipulam com arquivos e diretórios como <copy>, <zip>, etc.

```
<copy todir="${build.dir}/META-INF">
   <fileset dir="${xml.dir}" includes="ejb-jar.xml"/>
   <fileset dir="${xml.dir}/jboss">
      <include name="*.xml" />
      <exclude name="*-orig.xml" />
   </fileset>
</copy>
```

- **<dirset>**: árvore de diretórios
  - Não inclui arquivos individuais

Tipos de dados: coleções

- **<patternset>**: representa coleção de padrões

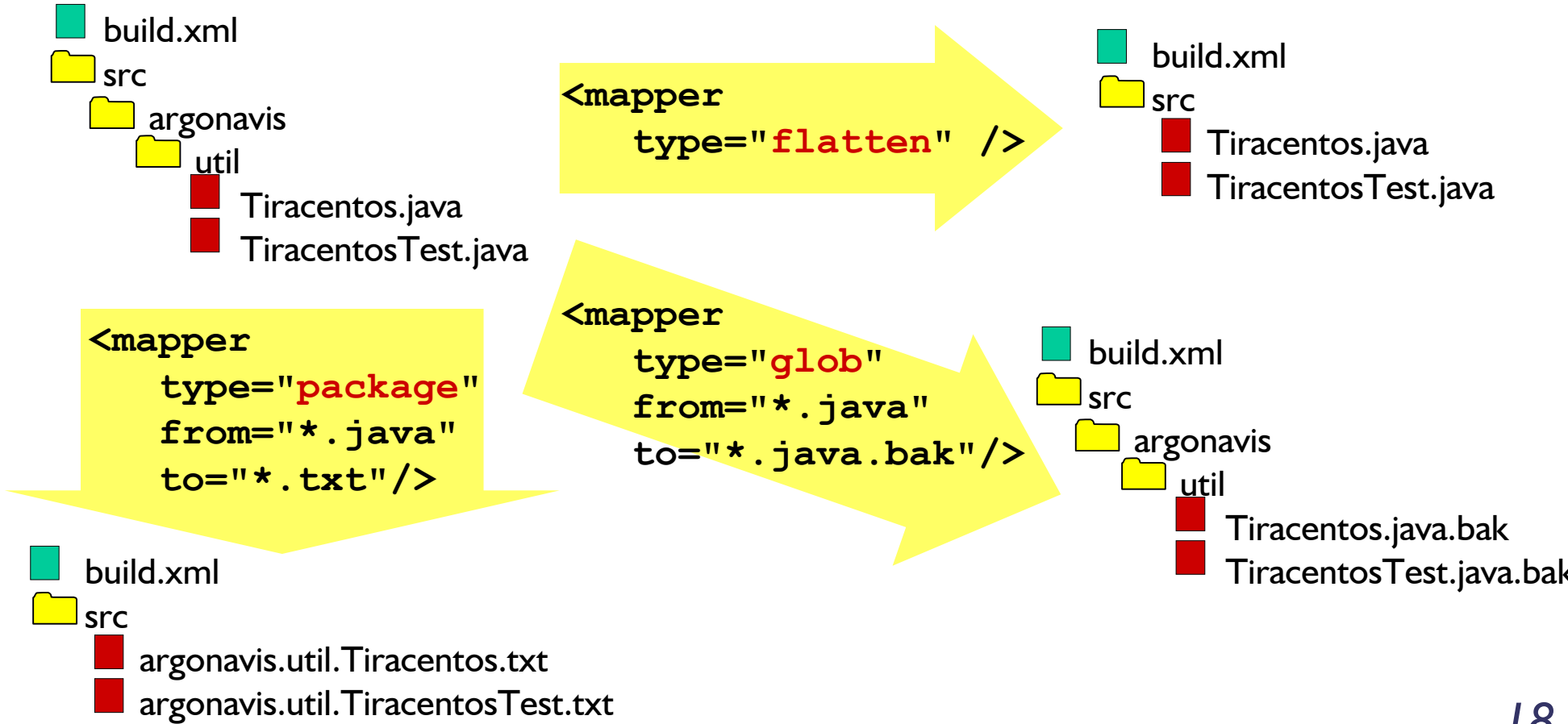
```
<patternset id="project.jars" >
  <include name="**/*.jar"/>
  <exclude name="**/*-test.jar"/>
</patternset>
```

- **<path>**: representa uma coleção de caminhos
  - Associa um ID a grupo de arquivos ou caminhos

```
<path id="server.path">
  <pathelement path="${j2ee.home}/lib/locale" />
  <fileset dir="${j2ee.home}/lib">
    <patternset refid="project.jars" />
  </fileset>
</path>

<target name="compile" depends="init">
   <javac destdir="${build.dir}" srcdir="${src.dir}">
      <classpath refid="server.path" />
   </javac>
</target>
```

Tipos de dados: File Mapper

- **<mapper>**: *altera nomes de arquivos durante cópias ou transformações*
  - *Seis tipos: identity, flatten, merge, regexp, glob, package*



Tipos de dados: seletores

- *Permitem a seleção dos elementos de um fileset usando critérios além dos definidos por <include> e <exclude>*
- *Sete seletores básicos (pode-se criar novos)*
  - ***<contains>*** *- Seleciona arquivos que contém determinado texto*
  - ***<date>*** *- Arquivos modificados antes ou depois de certa data*
  - ***<depend>*** *- Seleciona arquivos cuja data de modificação seja posterior a arquivos localizados em outro lugar*
  - ***<depth>*** *- Seleciona arquivos encontrados até certa profundidade de uma árvore de diretórios*
  - ***<filename>*** *- Equivalente ao include e exclude*
  - ***<present>*** *- Seleciona arquivo com base na sua (in)existência*
  - ***<size>*** *- Seleciona com base no tamanho em bytes*

Exemplo*: Seleciona arquivos do diretório "fonte" que também estão presentes em "destino"*

```
<fileset dir="fonte">
  <present targetdir="destino"/>
</fileset>
```

Tipos de dados: filtros

- **<filter>** e **<filterset>**: Permite a substituição de padrões em arquivos durante a execução de uma tarefa
  - Caractere default: @
  - Exemplo: a tarefa abaixo irá substituir todas as ocorrências de `@javahome@` por `c:\j2sdk1.4.0` nos arquivos copiados

```
<copy todir="${dest.dir}">
   <fileset dir="${src.dir}"/>
   <filterset>
      <filter token="javahome" value="c:\j2sdk1.4.0"/>
   </filterset>
</copy>
```

- Pares *token=valor* podem ser carregados de arquivo:

```
<filterset>
   <filtersfile file="build.properties" />
</filterset>
```

Substituição pode ser feita também sem tokens com <replace>

Execução de aplicações

- **<java>**: *roda o interpretador Java*

```
<target name="runrmiclient">
  <java classname="hello.rmi.HelloClient" fork="true">
    <jvmarg value="-Djava.security.policy=rmi.policy"/>
    <arg name="host" value="${remote.host}" />
    <classpath refid="app.path" />
  </java>
</target>
```

- **<exec>**: *executa um comando do sistema*

```
<target name="orbd">
  <exec executable="${java.home}\bin\orbd">
    <arg line="-ORBInitialHost ${nameserver.host}"/>
  </exec>
</target>
```

- **<apply>**: *semelhante a <exec> mas usado em executáveis que operam sobre outros arquivos*

Tarefas de rede

- ***<ftp>***: *Realiza a comunicação com um servidor FTP remoto para upload ou download de arquivos*
  - *Tarefa opcional que requer NetComponents.jar (http://www.savarese.org)*

```
<target name="remote.jboss.deploy" depends="dist">
    <ftp server="${ftp.host}" port="${ftp.port}"
         remotedir="/jboss/server/default/deploy"
         userid="admin" password="jboss"
         depends="yes" binary="yes">
        <fileset dir="${basedir}">
            <include name="*.war"/>
            <include name="*.ear"/>
            <include name="*.jar"/>
        </fileset>
    </ftp>
</target>
```

Efeitos sonoros

- **<sound>**: define um par de arquivos de som para soar no sucesso ou falha de um projeto
  - Tarefa opcional que requer Java Media Framework
- Exemplo:
  - No exemplo abaixo, o som frog.wav será tocado quando o build terminar sem erros fatais. Bark.wav tocará se houver algum erro que interrompa o processo:

```
<target name="init">
  <sound>
    <success source="C:/Media/frog.wav"/>
    <fail source="C:/Media/Bark.wav"/>
  </sound>
</target>
```

Ant programável

- Há duas formas de estender o Ant com novas funções
  - Implementar roteiros usando JavaScript
  - Criar novas tarefas reutilizáveis
- A tarefa <script> permite embutir JavaScript em um buildfile. Pode-se
  - realizar operações aritméticas e booleanas
  - utilizar estruturas como if/else, for, foreach e while
  - manipular com os elementos do buildfile usando DOM
- A tarefa <taskdef> permite definir novas tarefas
  - tarefa deve ser implementada em Java e estender Task
  - método execute() contém código de ação da tarefa
  - cada atributo corresponde a um método setXXX()

Integração com outras aplicações

- Ant provoca vários eventos que podem ser capturados por outras aplicações
  - Útil para implementar integração, enviar notificações por email, gravar logs, etc.
- Eventos
  - Build iniciou/terminou
  - Alvo iniciou/terminou
  - Tarefa iniciou/terminou
  - Mensagens logadas
- Vários listeners e loggers pré-definidos
  - Pode-se usar ou estender classe existente.
  - Para gravar processo (build) em XML:

```
ant -listener org.apache.tools.ant.XmlLogger
```

Integração com editores e IDEs

- Produtos que integram com Ant e oferecem interface gráfica e eventos para buildfiles:
  - Antidote: GUI para Ant (do projeto Jakarta)
    - http://cvs.apache.org/viewcvs/jakarta-ant-antidote/
  - JBuilder (AntRunner plug-in)
    - http://www.dieter-bogdoll.de/java/AntRunner/
  - NetBeans e Forté for Java
    - http://ant.netbeans.org/
  - Visual Age for Java (integração direta)
  - JEdit (AntFarm plug-in)
    - http://www.jedit.org
  - Jext (AntWork plug-in)
    - ftp://jext.sourceforge.net/pub/jext/plugins/AntWork.zip

Como gerenciar projetos com o Ant

- Crie um diretório para armazenar seu projeto. Nele guarde o seu build.xml
  - Use um arquivo build.properties para definir propriedades exclusivas do seu projeto (assim você consegue reutilizar o mesmo build.xml em outros projetos)
- Dentro desse diretório, crie alguns subdiretórios
  - src/ Para armazenar o código-fonte
  - lib/ Opcional. Para guardar os JARs de APIs usadas
  - doc/ Opcional. Para guardar a documentação gerada
- O seu Ant script deve ainda criar
  - build/ Ou classes/. Onde estará o código compilado
  - dist/ Ou jars/ ou release/. Onde estarão os JARs criados

Alvos básicos do build.xml

- Você também deve padronizar os nomes dos alvos dos seus build.xml. Alguns alvos típicos são
  - *init* Para criar dietórios, inicializar o ambiente, etc.
  - *clean* Para fazer a faxina, remover diretórios gerados, etc.
  - *compile* Para compilar
  - *build* Para construir a aplicação, integrar, criar JARs
  - *run* Para executar um cliente da aplicação
  - *test* Para executar os testes da aplicação
- Você pode usar outros nomes, mas mantenha um padrão
- Também pode criar uma nomenclatura que destaque alvos principais, usando maiúsculas. Ex:
  - *CLEAN, que chama clean-this, clean-that, undeploy, etc.*
  - *BUILD, que chama build-depend, build-client, build-server*

Exemplo de projeto

build.xml

```
<project default="compile" name="MiniEd">
  <property file="build.properties"/>
  <target name="init">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${dist.dir}"/>
  </target>
  <target name="clean">   ...  </target>
  <target name="compile"
          depends="init"> ...  </target>
  <target name="build"
          depends="compile">...</target>
  <target name="javadoc"
          depends="build"> ... </target>
  <target name="run"
          depends="build"> ... </target>
</project>
```

Estrutura dos arquivos (antes de executar o Ant)

```
# Nome da aplicação
app.name=minied
# Nomes dos diretórios
src.dir=src
docs.dir=docs
build.dir=classes
dist.dir=jars
# Nome da classe executável
app.main.class=com.javamagazine.minied.MiniEditor
root.package=com
```

build.properties

Conclusões

- O Ant é uma ferramenta indispensável em qualquer projeto de desenvolvimento Java
  - Permite automatizar todo o processo de desenvolvimento
  - Facilita a montagem da aplicação por outras pessoas
  - Ajuda em diversas tarefas essenciais do desenvolvimento como compilar, rodar, testar, gerar JavaDocs, etc.
  - Independe de um IDE comercial (mas pode ser facilmente integrado a um)
- Use o Ant em todos os seus projetos
  - Crie sempre um projeto e um buildfile, por mais simples que seja a sua aplicação
  - Escreva buildfiles que possam ser reutilizados

Exercícios

- 1. Crie um buildfile para cada projeto que você montar nos próximos módulos
  - Use o template básico de build.xml mostrado neste capítulo
  - Configure-o e personalize-o para incluir os alvos e tarefas específicas do seu projeto
  - Inclua alvos com tarefas de javadoc, jar e execução
- 2.Pratique com os exemplos apresentados
  - Execute os buildfiles e use o código como exemplo

Dica: consulte a documentação do Ant

  - Muito bem estruturada
  - Contém exemplos de todos os tags

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

9 Reuso com Herança e Composição

Helder da Rocha
www.argonavis.com.br

Como aumentar as chances de reuso

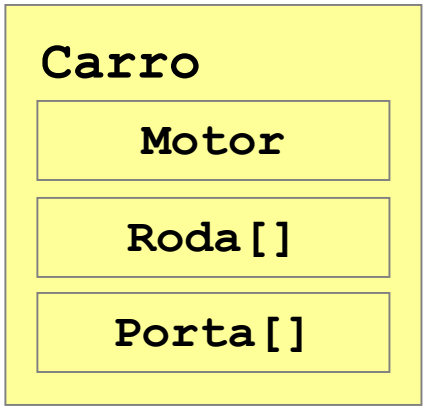
- Separar as partes que podem mudar das partes que não mudam. Exemplo: bibliotecas
  - Programador cliente deve poder usar o código sem a preocupação de ter que reescrever seu código caso surjam versões futuras
  - Programador de biblioteca deve ter a liberdade de fazer melhoramentos sabendo que o cliente não terá que modificar o seu código
- Em Java: esconder do cliente
  - Métodos que não fazem parte da interface de uso
  - Métodos que não fazem parte da interface de herança
  - Todos os atributos de dados

Reuso

- *Quando você precisa de uma classe, você pode*
  - *Usar uma classe que faz exatamente o que você deseja fazer*
  - *Escrever uma classe do zero*
  - *Reutilizar uma classe existente com composição*
  - *Reutilizar uma classe existente ou estrutura de classes com herança*

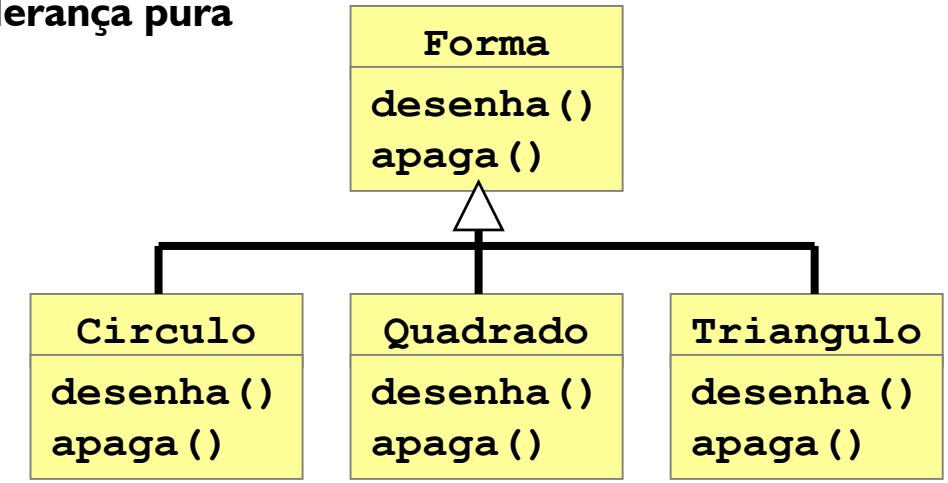
Composição vs. Herança

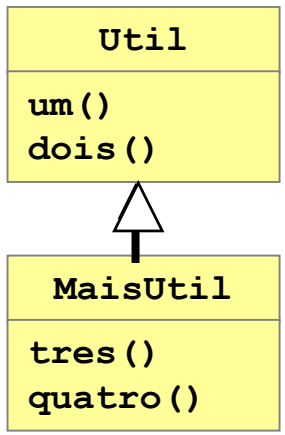
Composição pura



Ao reutilizar
uma classe
a composição
deve ser sua
escolha preferencial

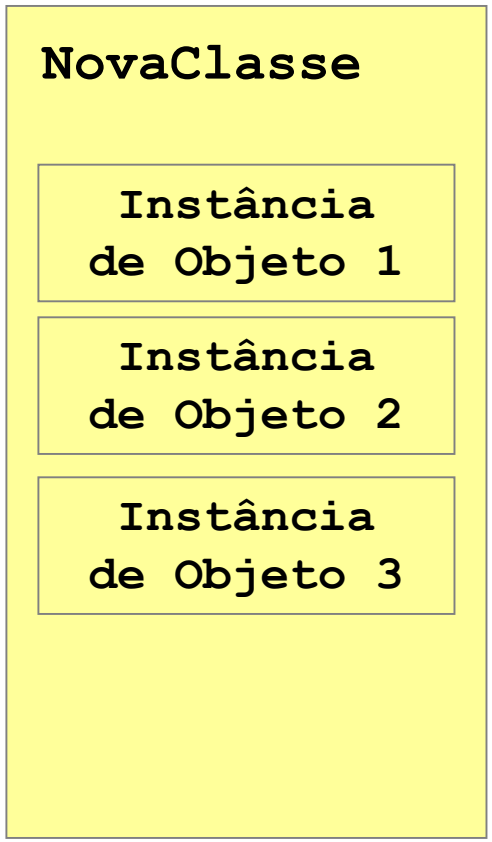
Herança pura





Extensão

Composição em Java

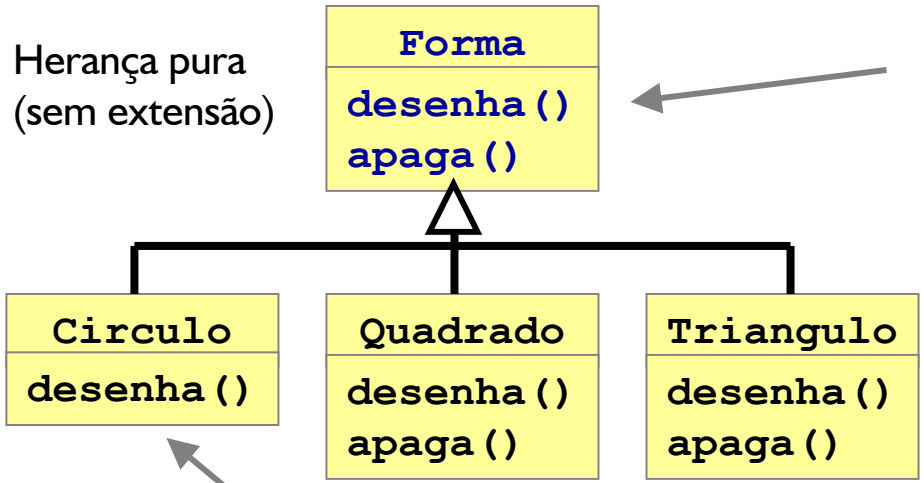



```
class NovaClasse {
    Um um = new Um();
    Dois dois = new Dois();
    Tres tres = new Tres();
}
```

- *Objetos podem ser inicializados no construtor*
- *Flexibilidade*
  - *Pode trocar objetos durante a execução!*
- *Relacionamento*
  - *"TEM UM"*

Herança em Java

Herança pura
(sem extensão)



interface original é automaticamente duplicada nas classes derivadas

Campos de dados da classe base também são duplicados

Se membro derivado não for redefinido, implementação original é usada

```
class Forma {
  public void desenha() {
    /*...*/
  }
  public void apaga() {
    /*...*/
  }
}
```

```
class Circulo extends Forma {
  public void desenha() {
    /* nova implementação */
  }
}
```

Assinatura do método tem que ser igual ou sobreposição não ocorrerá (poderá ocorrer sobrecarga não desejada)

Composição e Herança

- Composição e herança não são mutuamente exclusivas
  - As técnicas podem ser usadas em conjunto para obter os melhores resultados de cada uma
  - No desenvolvimento, composição é a técnica predominante
  - Herança geralmente ocorre mais no design de tipos

Quando usar? Composição ou herança?

- 1. Identifique os componentes do objeto, suas partes
  - Essas partes devem ser agregadas ao objeto via composição (é parte de)
- 2. Classifique seu objeto e tente encontrar uma semelhança de identidade com classes existentes
  - Herança só deve ser usada se você puder comparar seu objeto A com outro B dizendo, que A "É UM tipo de..." B.
  - Tipicamente, herança só deve ser usada quando você estiver construindo uma família de tipos (relacionados entre si)

Modificadores relacionados

- No projeto de uma classe, é preciso definir duas interfaces
  - Interface para uso de classes via composição
  - Interface para uso de classes via herança
- A palavra **protected** deve ser usada para declarar os métodos, construtores e variáveis que destinam-se à interface de herança
- Elementos usados em interface para composição devem ser declarados **public**
- A palavra **final** é usada para limitar o uso das classes, variáveis e métodos quando existe a possibilidade de haver herança: impede que implementações ou valores sejam alterados

Constantes

- *Para declarar uma **constante**, defina-a com um modificador **final***

```
final XIS = 0;
public static final IPSILON = 12;
```

- *Qualquer variável declarada como final tem que ser inicializada no momento da declaração*
  - *Exceção: argumentos constantes em métodos - valores não mudam dentro do método (uso em classes internas)*
- *Uma constante de tipo primitivo não pode receber outro valor*
- *Uma constante de referência não pode ser atribuída a um novo objeto*
  - *O objeto, porém, não é constante (apenas a referência o é). Os atributos do objeto podem ser alterados*

Métodos finais

- Método declarado como **final** não pode ser sobreposto
- Motivos para declarar um método **final**
  - Design: é a versão final (o método está "pronto")
  - Eficiência: compilador pode embutir o código do método no lugar da chamada e evitar realizar chamadas em tempo de execução: pode limitar o uso de sua classe
  - Procedimentos chamados de dentro de construtores: quaisquer métodos chamados de dentro de construtores devem ser final.
- Métodos declarados como **private** são implicitamente **final**

Classes finais

- A classe também pode ser declarada **final**

```
public final class Definitiva { ... }
```

  - Se uma classe não-final tiver todos os seus métodos declarados como final, é possível herdar os métodos, acrescentar novos, mas não sobrepor
- Se uma classe for declarada como final
  - Não é possível estender a classe (a classe nunca poderá aparecer após a cláusula extends de outra classe)
  - Todos os métodos da classe são finais
- Útil em classes que contém funções utilitárias e constantes apenas (ex: classe Math)

Modificadores de acesso

- Em ordem crescente de acesso
  - private
  - "package-private"
    - modificador ausente
  - protected
  - public

public

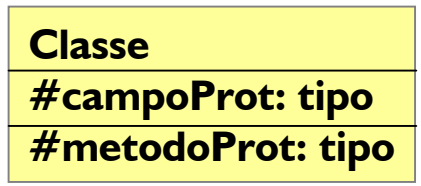
- *Acessível*
  - *na própria classe*
  - *nas subclasses*
  - *nas classes do mesmo pacote*
  - *em todas as outras classes*

Classe
+campoPublico: tipo
+metodoPublico: tipo

- *Use para*
  - *construtores e métodos que fazem parte da interface do objeto*
  - *métodos estáticos utilitários*
  - *constantes (estáticas) utilitárias*
- *Evite usar em*
  - *construtores e métodos de uso restrito*
  - *campos de dados de objetos*

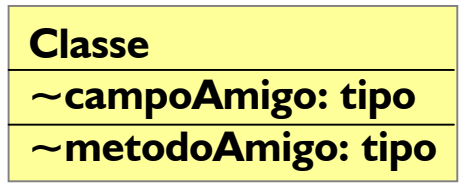
protected

- *Acessível*
  - *na própria classe*
  - *nas subclasses*
  - *nas classes do mesmo pacote*
- *Use para*
  - *construtores que só devem ser chamados pelas subclasses (através de super())*
  - *métodos que só devem ser usados se sobrepostos*
- *Evite usar em*
  - *construtores em classes que não criam objetos*
  - *métodos com restrições à sobreposição*
  - *campos de dados de objetos*



package-private

- *Modificador ausente*
  - *se não houver outro modificador de acesso, o acesso é "package-private".*
- *Acessível*
  - *na própria classe*
  - *nas classes e subclasses do mesmo pacote*
- *Use para*
  - *construtores e métodos que só devem ser chamados pelas classes e subclasses do pacote*
  - *constantes estáticas úteis apenas dentro do pacote*
- *Evite usar em*
  - *construtores em classes que não criam objetos*
  - *métodos cujo uso externo seja limitado ou indesejável*
  - *campos de dados de objetos*



private

- *Acessível*
  - *na própria classe (nos métodos, funções estáticas, blocos estáticos e construtores)*
- *Use para*
  - *construtores de classes que só devem criar um número limitado de objetos*
  - *métodos que não fazem parte da interface do objeto*
  - *funções estáticas que só têm utilidade dentro da classe*
  - *variáveis e constantes estáticas que não têm utilidade ou não podem ser modificadas fora da classe*
  - *campos de dados de objetos*

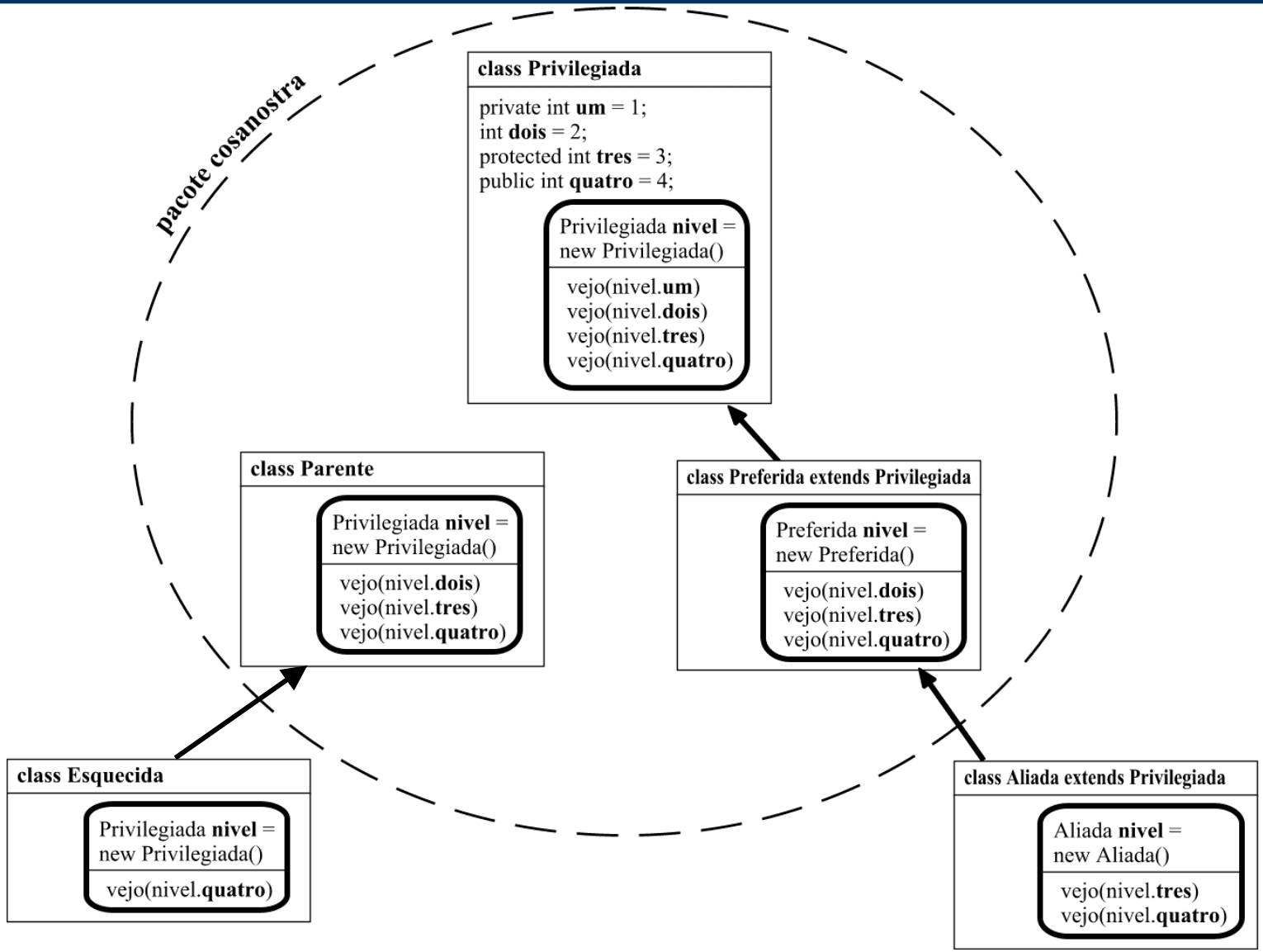
Observações sobre acesso

- Classes e interfaces (exceto classes internas)
  - Só podem ser package-private ou public
- Construtores
  - Se private, criação de objetos depende da classe
  - Se protected, apenas subclasses (além da própria classe e classes do pacote) podem criar objetos
- Váriáveis e constantes
  - O acesso afeta sempre a leitura e alteração. Efeitos "read-only" e "write-only" só podem ser obtidos por meio de métodos
- Variáveis locais
  - Usar modificadores de acesso dentro dos métodos é ilegal e não faz sentido pois variáveis locais só têm escopo local

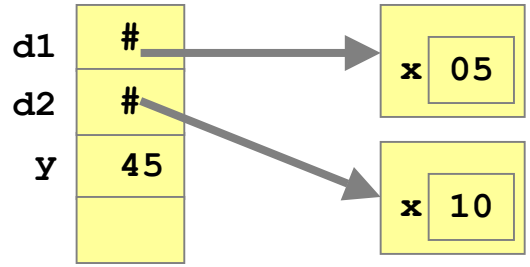
Sobreposição

- *Métodos sobrepostos nunca podem ter* ***menos*** *acesso que os métodos originais*
  - *Se método original for* ***public****, novas versões têm que ser public*
  - *Se método original for* ***protected****, novas versões podem ser protected ou public*
  - *Se método original* ***não tiver modificador de acesso*** *(é "package-private"), novas versões podem ser declaradas sem modificador de acesso, com modificador protected ou public*
  - *Se método original for* ***private****, ele não será visível da subclasse e portanto, jamais poderá ser estendido.*

Exemplo


pacote cosanostra
class Privilegiada
private int um = 1;
int dois = 2;
protected int tres = 3;
public int quatro = 4;
Privilegiada nivel = new Privilegiada()
vejo(nivel.um)
vejo(nivel.dois)
vejo(nivel.tres)
vejo(nivel.quatro)
class Parente
Privilegiada nivel = new Privilegiada()
vejo(nivel.dois)
vejo(nivel.tres)
vejo(nivel.quatro)
class Preferida extends Privilegiada
Preferida nivel = new Preferida()
vejo(nivel.dois)
vejo(nivel.tres)
vejo(nivel.quatro)
class Esquecida
Privilegiada nivel = new Privilegiada()
vejo(nivel.quatro)
class Aliada extends Privilegiada
Aliada nivel = new Aliada()
vejo(nivel.tres)
vejo(nivel.quatro)

Mais sobre 'static'



- *Variáveis declaradas como 'static' existem antes de existir qualquer objeto da classe*
  - *Só existe uma variável static, independente do número de objetos criado com a classe*
  - *Podem ser chamadas externamente pelo nome da classe*

    `Color.red, System.out, BorderLayout.NORTH`
  - *Podem também ser chamadas através da referência de um objeto (evite usar este método)*

```
class Duas {
    int x;
    static int y;
}
```

```
(...)
Duas d1 = new Duas();
Duas d2 = new Duas();
d1.x = 5;
d2.x = 10;
//Duas.x = 60; // ilegal!
d1.y = 15;
d2.y = 30;
Duas.y = 45;
(...)
```

mesma variável!

Use esta notação apenas

Classe
campoStatic: tipo
metodoStatic: tipo

Métodos static

- *Métodos static **nunca** são sobrepostos*
  - *Método static de assinatura igual na subclasse apenas "**oculta**" original*
  - *Não há polimorfismo: método está sempre associado ao tipo da **classe** (e não à instância)*
- *Exemplo: considere as classes abaixo*

```
class Alfa {
  static void metodo() {
    System.out.println("Alfa!");
  }
}
```

```
class Beta extends Alfa {
  static void metodo() {
    System.out.println("Beta!");
  }
}
```

- *O código a seguir*

```
Alfa pai = new Alfa ();
pai.metodo();

Beta filho1 = new Beta ();
filho1.metodo();

Alfa filho2 = new Beta ();
filho2.metodo();
```

irá imprimir:

Alfa!
Beta!
Alfa!

e não...

Alfa!
Beta!
Beta!

*como ocorreria **se** os métodos fossem de instância*

- *Não chame métodos static via referências! Use sempre:*

```
Classe.metodo()
```

Classe que só permite um objeto (Singleton pattern)

```
public class Highlander {
  private Highlander() {}
  private static Highlander instancia;
  public static Highlander criarInstancia() {
    if (instancia == null)
      instancia = new Highlander();
    }
    return instancia;
  }
}
```

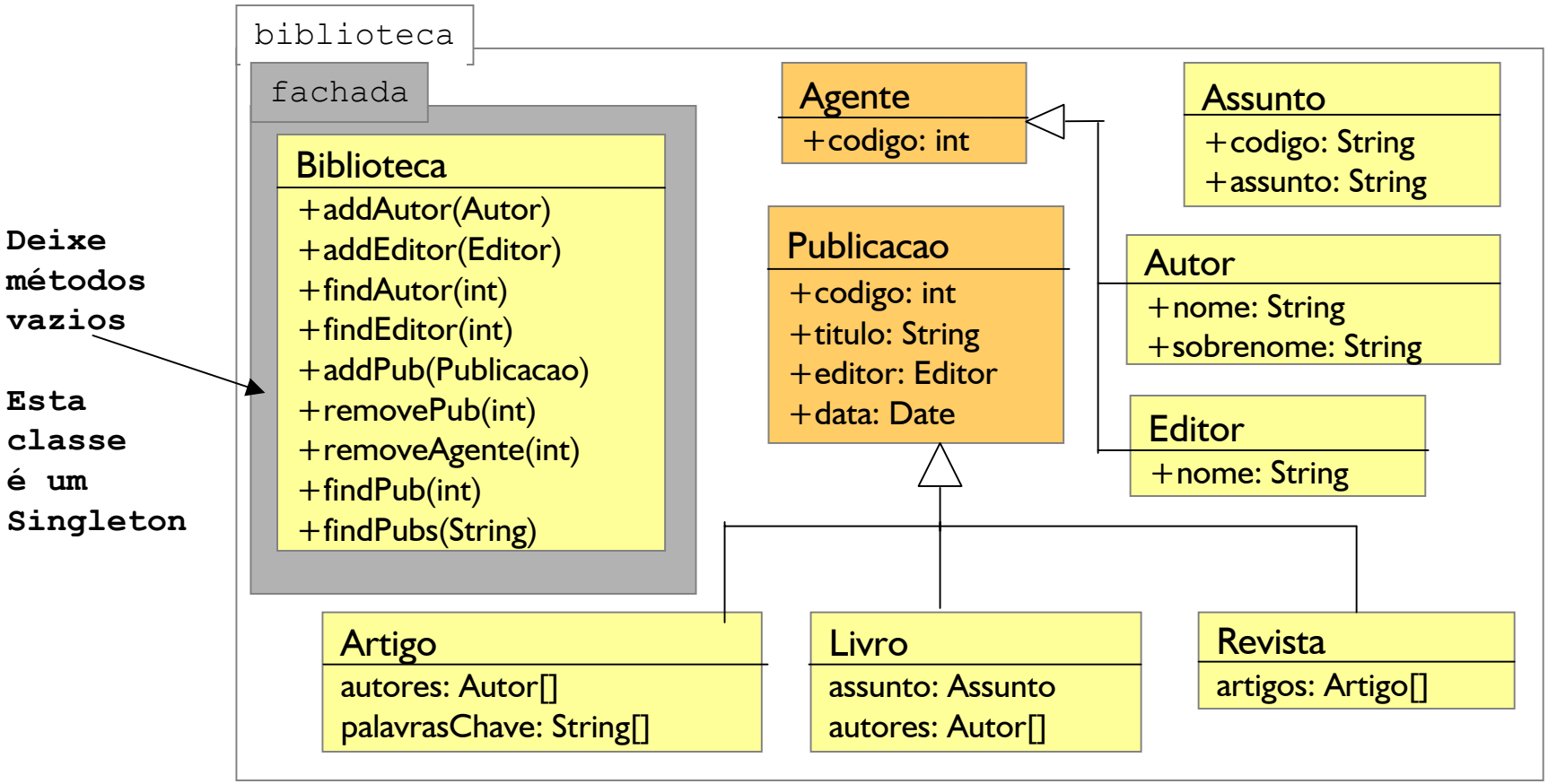
Esta classe implementa o padrão de projeto Singleton

```
public class Fabrica {
  public static void main(String[] args) {
    Highlander h1, h2, h3;
    //h1 = new Highlander(); // nao compila!
    h2 = Highlander.criarInstancia();
    h3 = Highlander.criarInstancia();
    if (h2 == h3) {
       System.out.println("h2 e h3 são mesmo objeto!");
    }
  }
}
```

Esta classe cria apenas um objeto Highlander

Exercícios

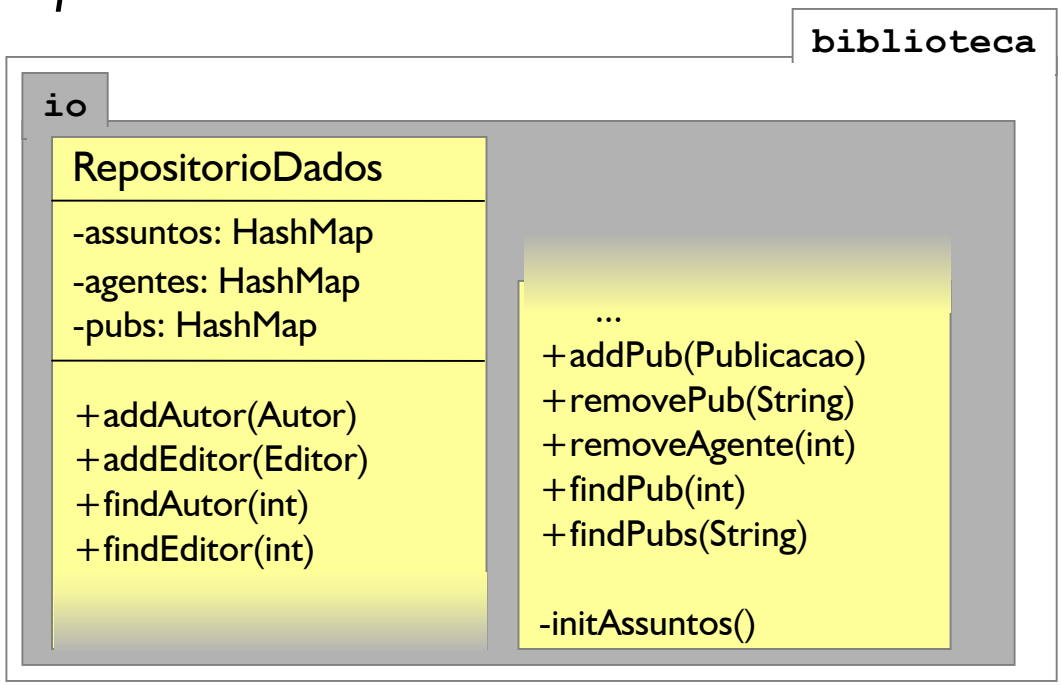
- *1. Implemente a seguinte hierarquia de classes*



- *a) Os dados representam propriedades (pares get/set) e não campos de dados (que devem ser private)*
- *b) Implemente equals(), toString() e hashCode() e construtores em cada classe biblioteca.**

Exercícios (2)

- *2. Implemente a classe abaixo*



`Inicie previamente os assuntos (veja slide seguinte)`

- *3. Implemente os métodos de Biblioteca para que chamem os métodos de RepositorioDados.*
- *4. Coloque tudo em um JAR.*
- *5. Escreva uma classe que contenha um main(), importe os pacotes da biblioteca, crie uma Biblioteca e acrescente autores, livros, artigos, revistas, e imprima os resultados.*

Apêndice: RepositorioDados (trecho)

```
private java.util.HashMap assuntos =
                    new java.util.HashMap();

public void initAssuntos() {
    assuntos = new java.util.HashMap(10);
    assuntos.put("000", "Generalidades");
    assuntos.put("100", "Filosofia");
    assuntos.put("200", "Religião");
    assuntos.put("300", "Ciências Sociais");
    assuntos.put("400", "Línguas");
    assuntos.put("500", "Ciências Naturais");
    assuntos.put("600", "Ciências Aplicadas");
    assuntos.put("700", "Artes");
    assuntos.put("800", "Literatura");
    assuntos.put("900", "História");
}
```

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Interfaces e polimorfismo

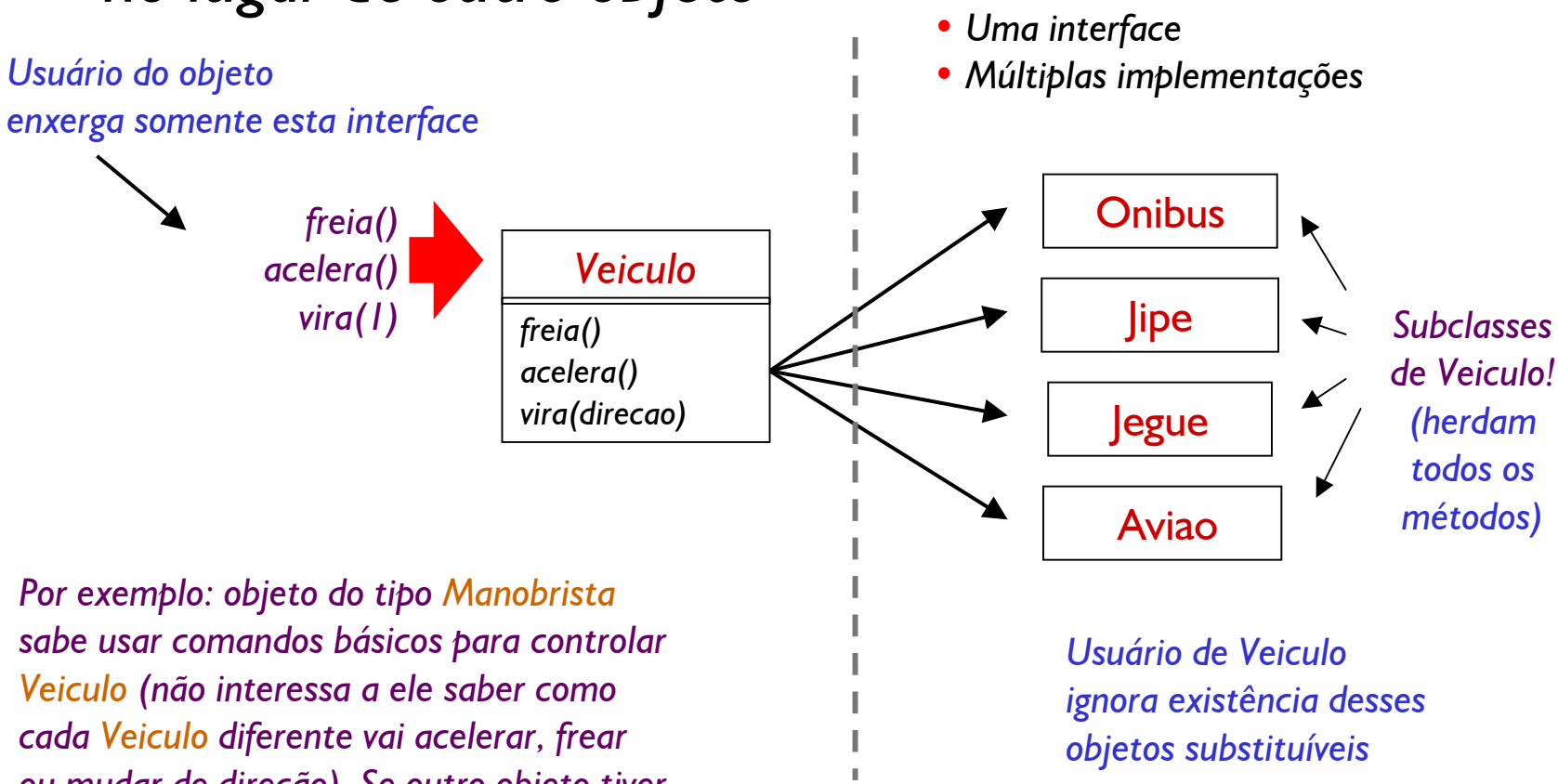
Helder da Rocha (helder@acm.org)

O que é polimorfismo

- **Polimorfismo** (poli=muitos, morfo=forma) é uma característica essencial de linguagens orientadas a objeto
- Como funciona?
  - Um objeto que faz papel de interface serve de **intermediário** fixo entre o programa-cliente e os objetos que irão executar as mensagens recebidas
  - O programa-cliente não precisa saber da existência dos outros objetos
  - Objetos podem ser substituídos sem que os programas que usam a interface sejam afetados

Objetos substituíveis

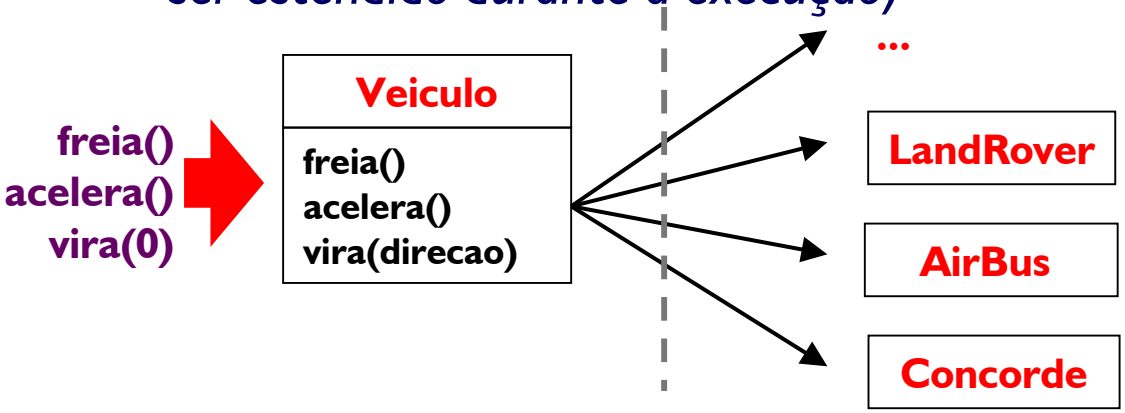
- *Polimorfismo significa que um objeto pode ser usado no lugar de outro objeto*

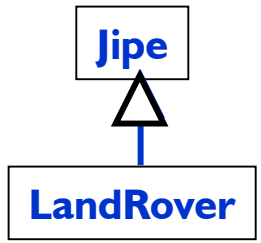


Por exemplo: objeto do tipo Manobrista sabe usar comandos básicos para controlar Veiculo (não interessa a ele saber como cada Veiculo diferente vai acelerar, frear ou mudar de direção). Se outro objeto tiver a mesma interface, Manobrista saberá usá-lo

Programas extensíveis

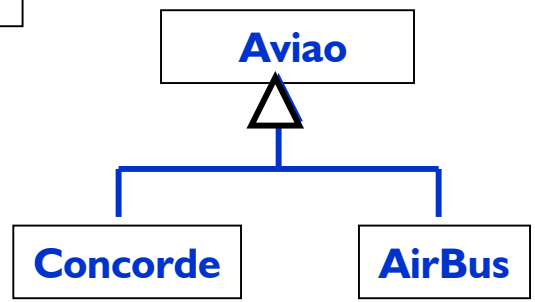
- *Novos objetos podem ser usados em programas que não previam a sua existência*
  - *Garantia que métodos da interface existem nas classes novas*
  - *Objetos de novas classes podem ser criados e usados (programa pode ser estendido durante a execução)*





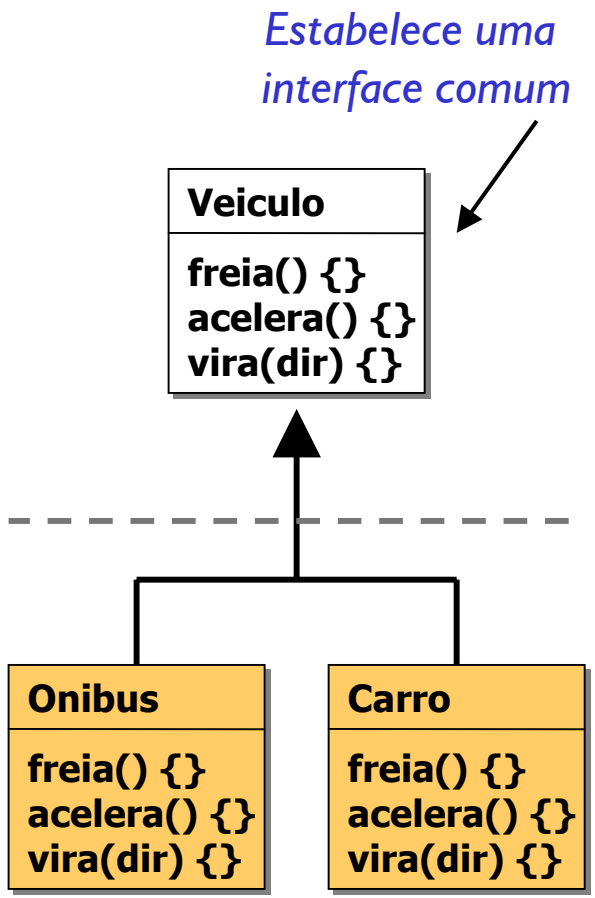
Mesmo nome. Implementações diferentes.

```
Veiculo v1 = new Veiculo();
Veiculo v2 = new Aviao();
Veiculo v3 = new Airbus();
v1.acelera(); // acelera Veiculo
v2.acelera(); // acelera Aviao
v3.acelera(); // acelera AirBus
```



Interface vs. implementação

- *Polimorfismo permite separar a interface da implementação*
- *A classe base define a interface comum*
  - *Não precisa dizer **como** isto vai ser feito*
    Não diz: eu sei como frear um Carro ou um Ônibus
  - *Diz apenas que os métodos existem, que eles retornam determinados tipos de dados e que requerem certos parâmetros*
    Diz: Veiculo pode acelerar, frear e virar para uma direção, mas a direção deve ser fornecida

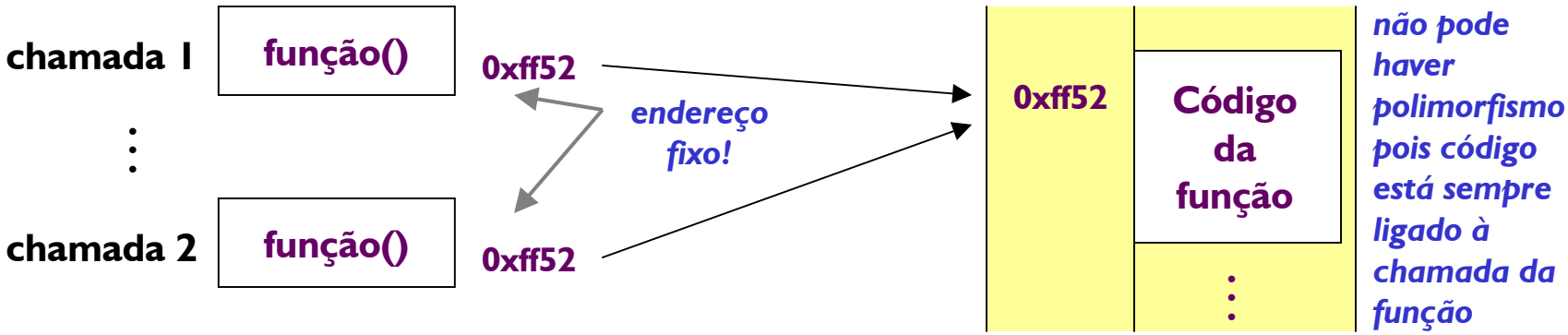


Como funciona

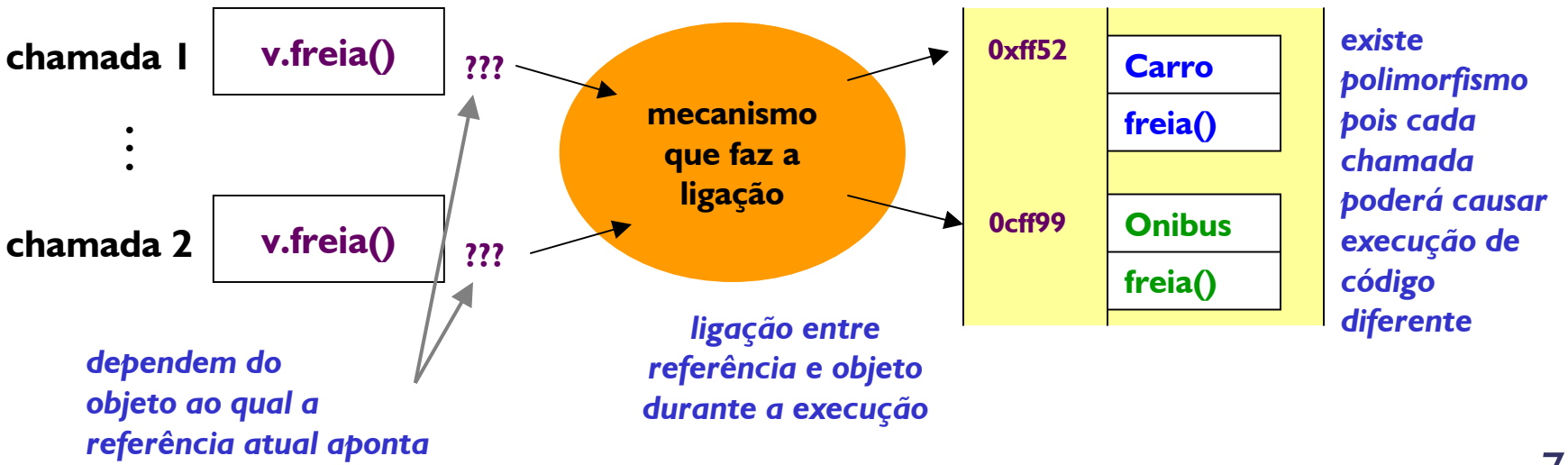
- Suporte a polimorfismo depende do suporte à ligação tardia (late binding) de chamadas de função
  - A referência (interface) é conhecida em **tempo de compilação** mas o objeto a que ela aponta (implementação) não é
  - O objeto pode ser da mesma classe ou de uma subclasse da referência (garante que a TODA a interface está implementada no objeto)
  - Uma única referência, pode ser ligada, **durante a execução**, a vários objetos diferentes (a referência é polimorfa: pode assumir muitas formas)

Ligação de chamadas de função

- *Em tempo de compilação (early binding) - C!*

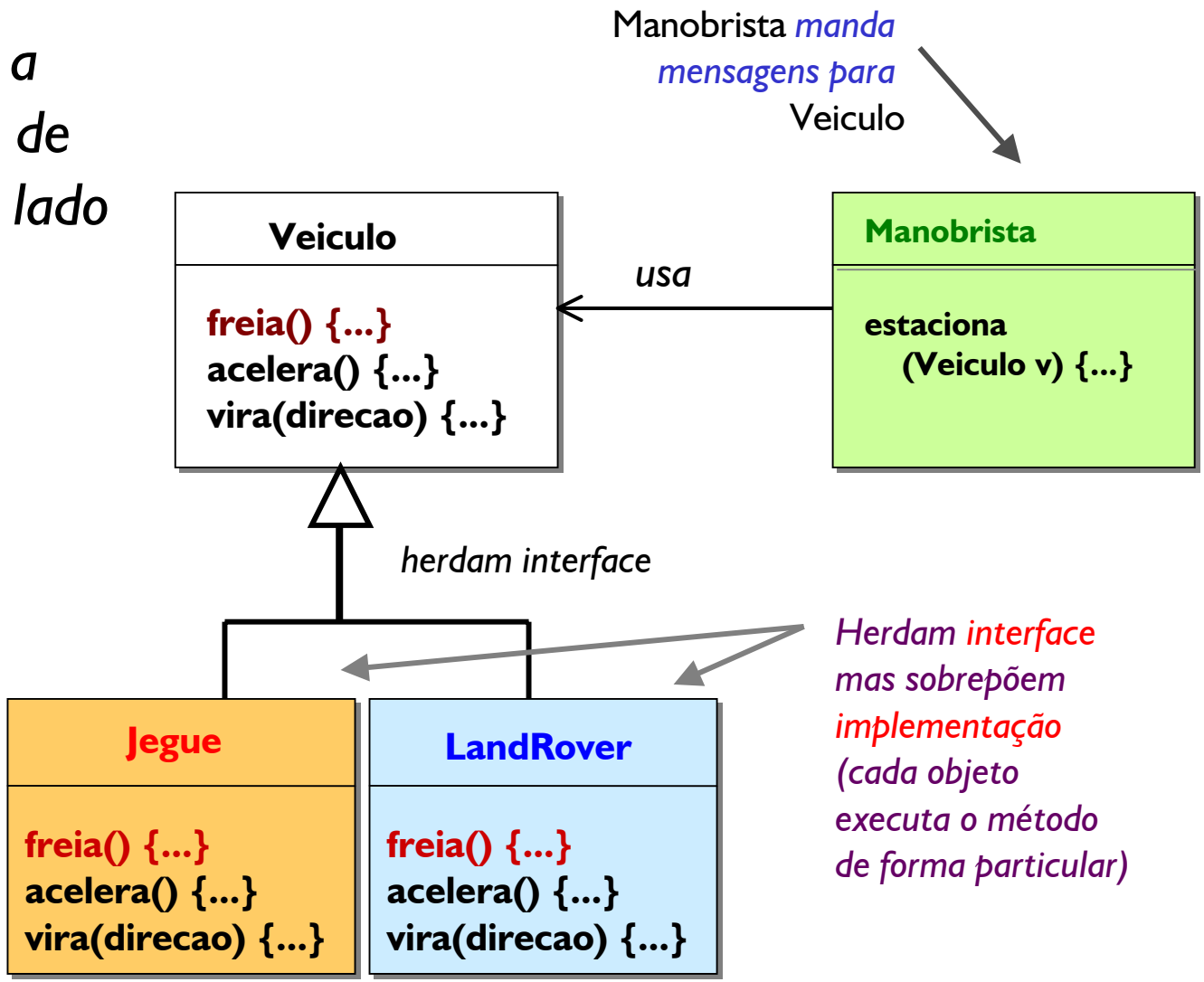


- *Em tempo de execução (late binding) - Java!*



Exemplo (1)

- *Considere a hierarquia de classes ao lado*

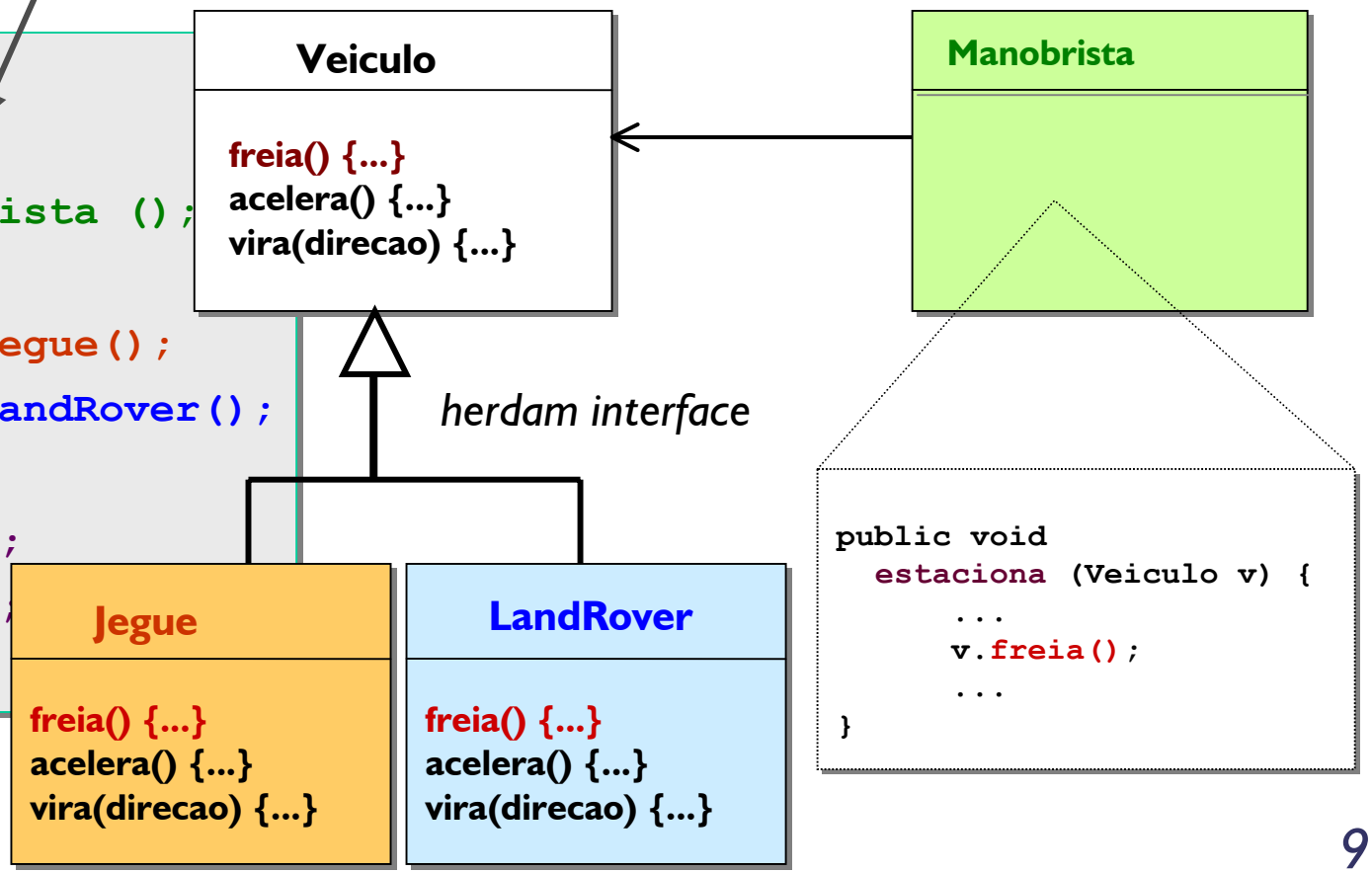


Exemplo (2)

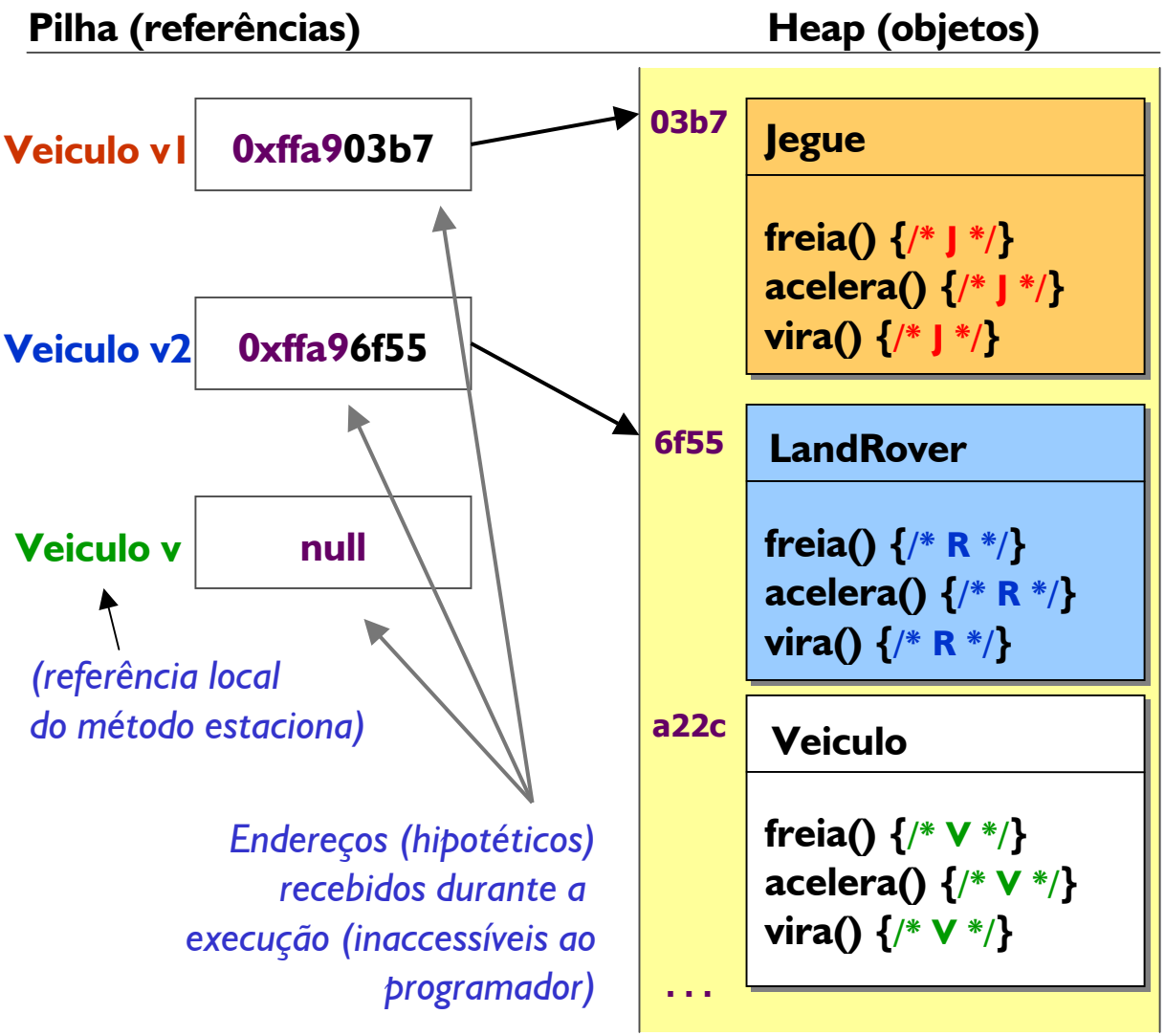
Trecho de programa que usa Manobrista:
Em tempo de execução passa implementação de Jegue e LandRover no lugar da implementação original de Veiculo
(aproveita apenas a interface de Veiculo)

Manobrista usa a classe Veiculo (e ignora a existência de tipos específicos de Veiculo como Jegue e LandRover

```
(...)
Manobrista mano
      = new Manobrista ();

Veiculo v1 = new Jegue();
Veiculo v2 = new LandRover();

mano.estaciona(v1);
mano.estaciona(v2);
(...)
```



Detalhes (I)



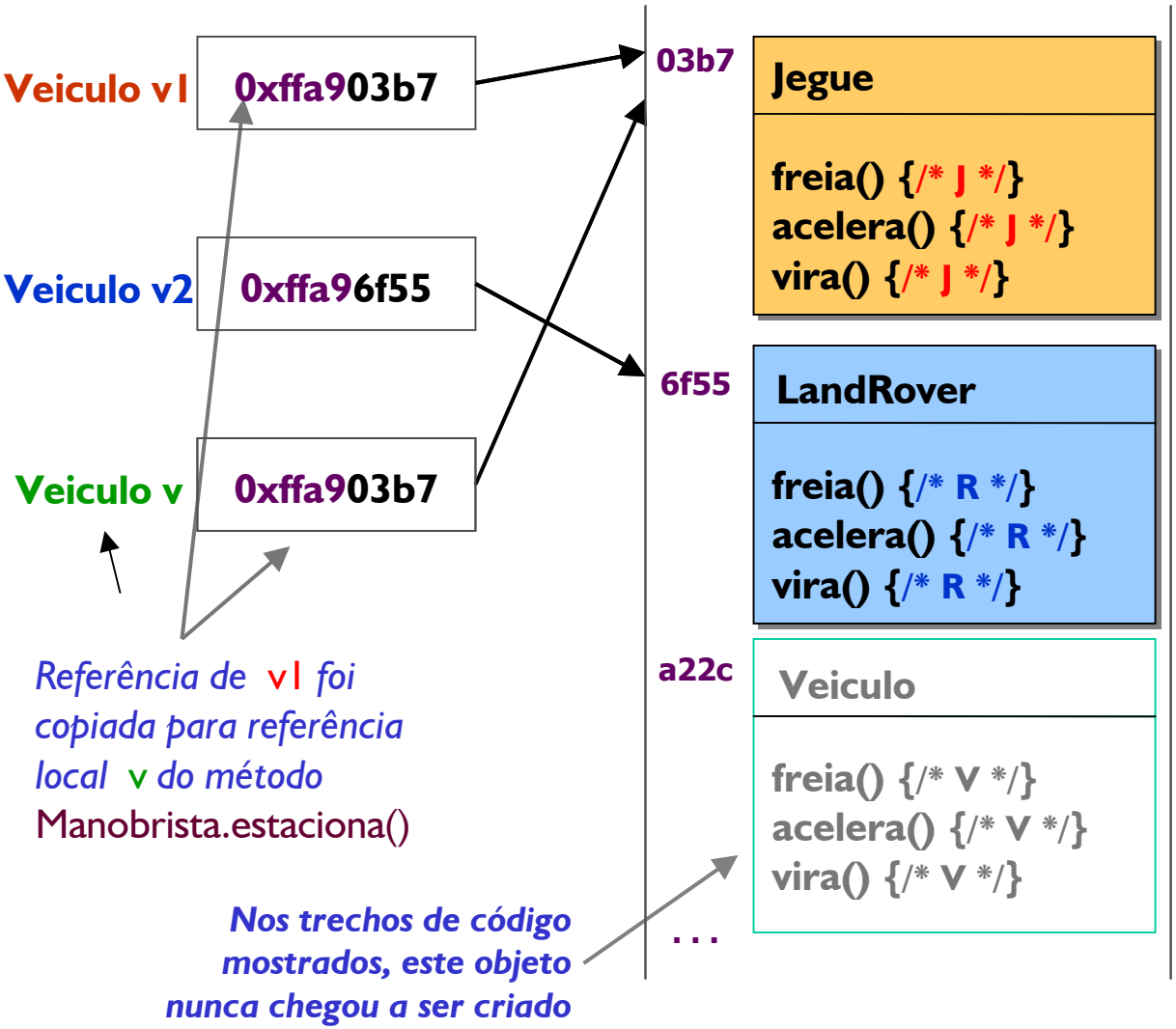

```
Manobrista
public void
  estaciona (Veiculo v) {
     ...
     v.freia();
     ...
}
```

Qual freia() *será executado quando o trecho abaixo for executado?*

```
(...)
Veiculo v1 =
      new Jegue();
mano.estaciona(v1);
(...)
```

(**mano** ***é do tipo Manobrista***)

Como funciona (3)




```
Manobrista
public void
  estaciona (Veiculo v) {
     ...
     v.freia();
     ...
}
```

Na chamada abaixo, Veiculo *foi "substituído" com* Jegue. *A implementação usada foi* Jegue.freia()

```
(...)
Veiculo v1 =
      new Jegue();
mano.estaciona(v1);
(...)
```

```
Veiculo v = v1
```

Argumento do método estaciona()

Conceitos abstratos

- *Como deve ser implementado freia() na classe Veiculo?*
  - *Faz sentido dizer como um veículo genérico deve frear?*
  - *Como garantir que cada tipo específico de veículo redefina a implementação de freia()?*
- *O método freia() é um procedimento abstrato em Veiculo*
  - *Deve ser usada apenas a implementação das subclasses*
- *E se não houver subclasses?*
  - *Como freia um Veiculo genérico?*
  - *Com que se parece um Veiculo generico?*
- *Conclusão: não há como construir objetos do tipo Veiculo*
  - *É um conceito genérico demais*
  - *Mas é ótimo como interface! Eu posso saber dirigir um Veiculo sem precisar saber dos detalhes de sua implementação*

Métodos e classes abstratos

- *Procedimentos genéricos que têm a finalidade de servir apenas de interface são métodos abstratos*
  - *declarados com o modificador abstract*
  - *não têm corpo {}. Declaração termina em ";"*

```
public abstract void freia();

public abstract float velocidade();
```

- *Métodos abstratos não podem ser usados, apenas declarados*
  - *São usados através de uma subclasse que os implemente!*

Classes abstratas

- *Uma classe pode ter métodos concretos e abstratos*
  - *Se tiver um ou mais método abstrato, classe não pode ser usada para criar objetos e precisa ter declaração* ***abstract***

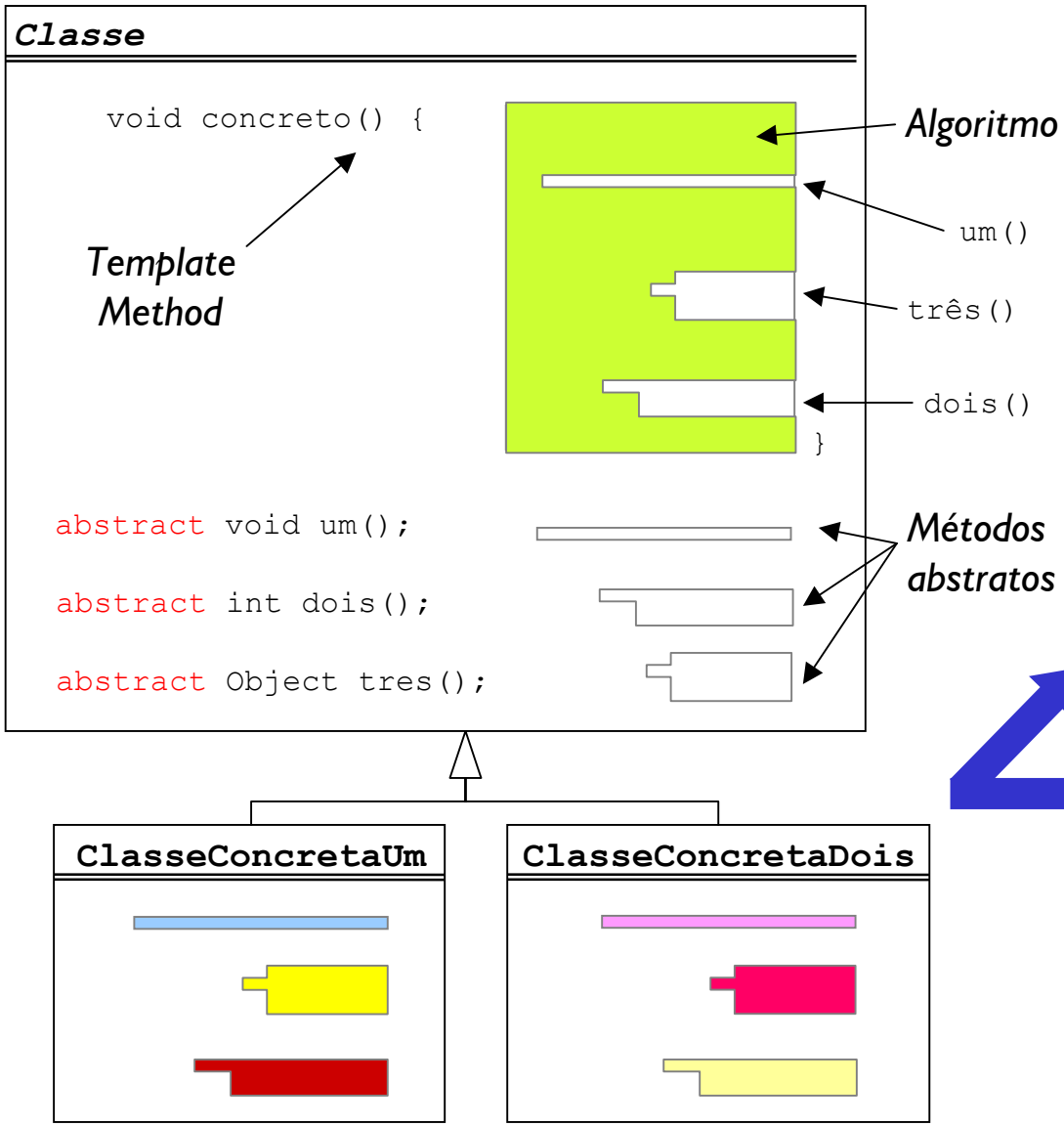
```
public abstract class Veiculo { ... }
```

  - *Objetos do tipo Veiculo não podem ser criados*
  - *Subclasses de Veiculo podem ser criados desde que implementem TODOS os métodos abstratos herdados*
  - *Se a implementação for parcial, a subclasse também terá que ser declarada abstract*

Classes abstratas (2)

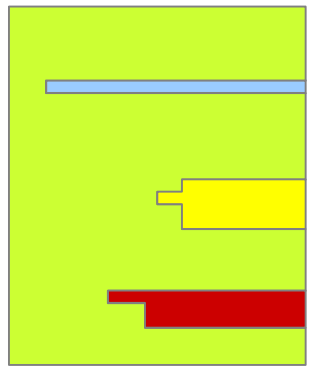
- Classes abstratas são criadas para serem estendidas
- Podem ter
  - métodos concretos (usados através das subclasses)
  - campos de dados (memória é alocada na criação de objetos pelas suas subclasses)
  - construtores (chamados via super() pelas subclasses)
- Classes abstratas "puras"
  - não têm procedimentos no construtor (construtor vazio)
  - não têm campos de dados (a não ser constantes estáticas)
  - todos os métodos são abstratos
- Classes abstratas "puras" podem ser definidas como "interfaces" para maior flexibilidade de uso

Template Method design pattern

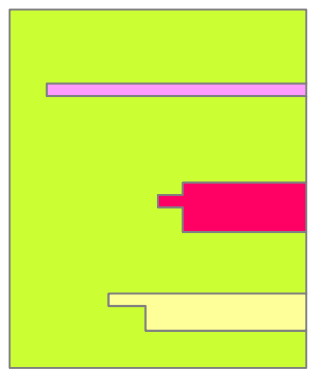


Algoritmos resultantes

```
Classe x =
  new ClasseConcretaUm()
x.concreto()
```

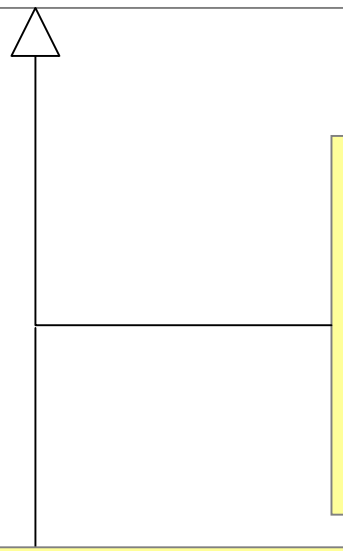


```
Classe x =
  new ClasseConcretaDois()
x.concreto()
```



Template method: implementação

```
public abstract class Template {
   public abstract String link(String texto, String url);
   public String transform(String texto) { return texto; }

   public String templateMethod() {
      String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");
      return transform(msg);
   }
```



```
public class HTMLData extends Template {
   public String link(String texto, String url) {
      return "<a href='"+url+"'>"+texto+"</a>";
   }
   public String transform(String texto) {
      return texto.toLowerCase();
   }
}
```

```
public class XMLData extends Template {
   public String link(String texto, String url) {
      return "<endereco xlink:href='"+url+"'>"+texto+"</endereco>";
   }
}
```

Exercício

- 1. Crie um novo projeto
  - Copie um build.xml genérico
- 2. Implemente os exemplos da aula:
  - Manobrista, Veiculo, Jegue e LandRover
  - a) Implemente os métodos com instruções de impressão, por exemplo:

```
public void freia() {
    System.out.println("Chamou Jegue.freia()");
}
```

  - b) Faça com que os métodos de Veiculo sejam abstratos e refaça o exercício

Upcasting

- *Tipos genéricos (acima, na hierarquia) sempre podem receber objetos de suas subclasses:* ***upcasting***

```
Veiculo v = new Carro();
```

  - *Há garantia que subclasses possuem pelo menos os mesmos métodos que a classe*
  - *v só tem acesso à "parte Veiculo" de Carro. Qualquer extensão (métodos definidos em Carro) não faz parte da extensão e não pode ser usada pela referência v.*

Downcasting

- *Tipos específicos (abaixo, na hierarquia) não podem receber explicitamente seus objetos que foram declarados como referências de suas superclasses:* ***downcasting***

```
Carro c = v; // não compila!
```

  - *O código acima não compila, apesar de v apontar para um Carro! É preciso converter a referência:*

```
Carro c = (Carro) v;
```

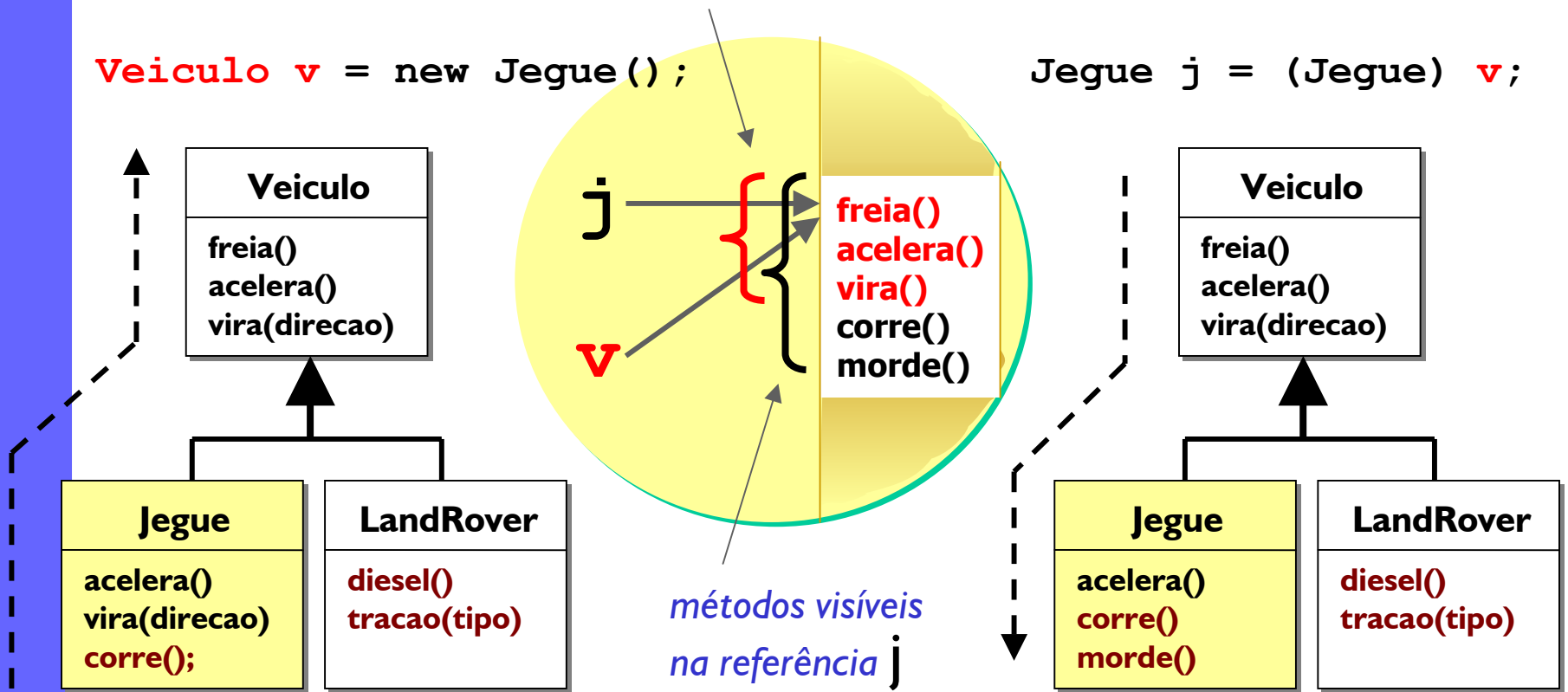
  - *E se v for Onibus e não Carro?*

Upcasting e downcasting

- *Upcasting*
  - *sobe a hierarquia*
  - *não requer cast*
- *Downcasting*
  - *desce a hierarquia*
  - *requer operador de cast*

```
Veiculo v = new Jegue();
```

```
Jegue j = (Jegue) v;
```



ClassCastException

- *O downcasting explícito sempre é aceito pelo compilador se o tipo da direita for superclasse do tipo da esquerda*

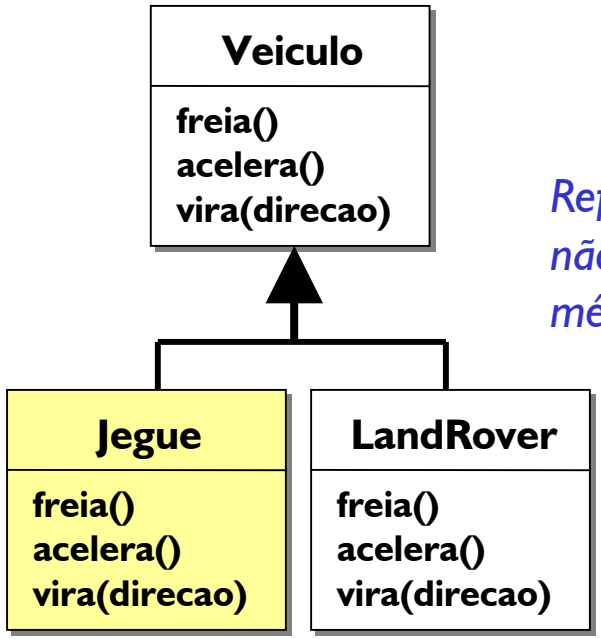
```
Veiculo v = new Onibus();
Carro c = (Carro) v; // passa na compilação
```

  - *Object, portanto, pode ser atribuída a qualquer tipo de referência*
- *Em tempo de execução, a referência terá que ser ligada ao objeto*
  - *Incompatibilidade provocará ClassCastException*
- *Para evitar a exceção, use instanceof*

```
if (v instanceof Carro)
        c = (Carro) v;
```

Herança Pura vs. Extensão

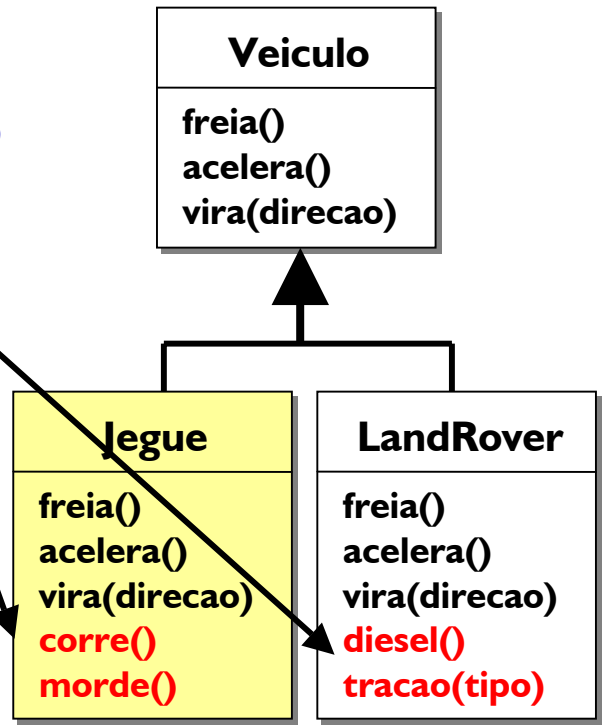
- *Herança pura: referência têm acesso a todo o objeto*




```
Veiculo v = new Jegue();
v.freia() // freia o Jegue
v.acelera(); // acelera o Jegue
```

- *Extensão: referência apenas tem acesso à parte definida na interface da classe base*

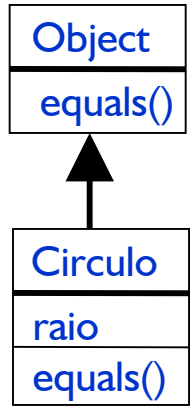
Referência Veiculo não enxerga estes métodos




```
Veiculo v = new Jegue();
v.corre() // ERRADO!
v.acelera(); //OK
```

Ampliação da referência

- *Uma referência pode apontar para uma classe estendida, mas só pode usar métodos e campos de sua interface*
  - *Para ter acesso total ao objeto que estende a interface original, é preciso usar referência que conheça toda sua interface pública*
- *Exemplo*



ERRADO: ***raio*** *não faz parte da interface de* ***Object***

```
class Circulo extends Object {
  public int raio;
  public boolean equals(Object obj) {
    if (this.raio == obj.raio)
      return true;
    return false;
  }
} // CÓDIGO ERRADO!
```

verifica se ***obj*** *realmente é um* ***Circulo***

cria nova referência que tem acesso a toda a interface de **Circulo**

```
class Circulo extends Object {
  public int raio;
  public boolean equals(Object obj) {
    if (obj instanceof Circulo) {
      Circulo k = (Circulo) obj;
      if (this.raio == k.raio)
        return true;
    }
    return false;
  }
}
```

Como **k** *é* **Circulo** *possui raio*

Interfaces Java

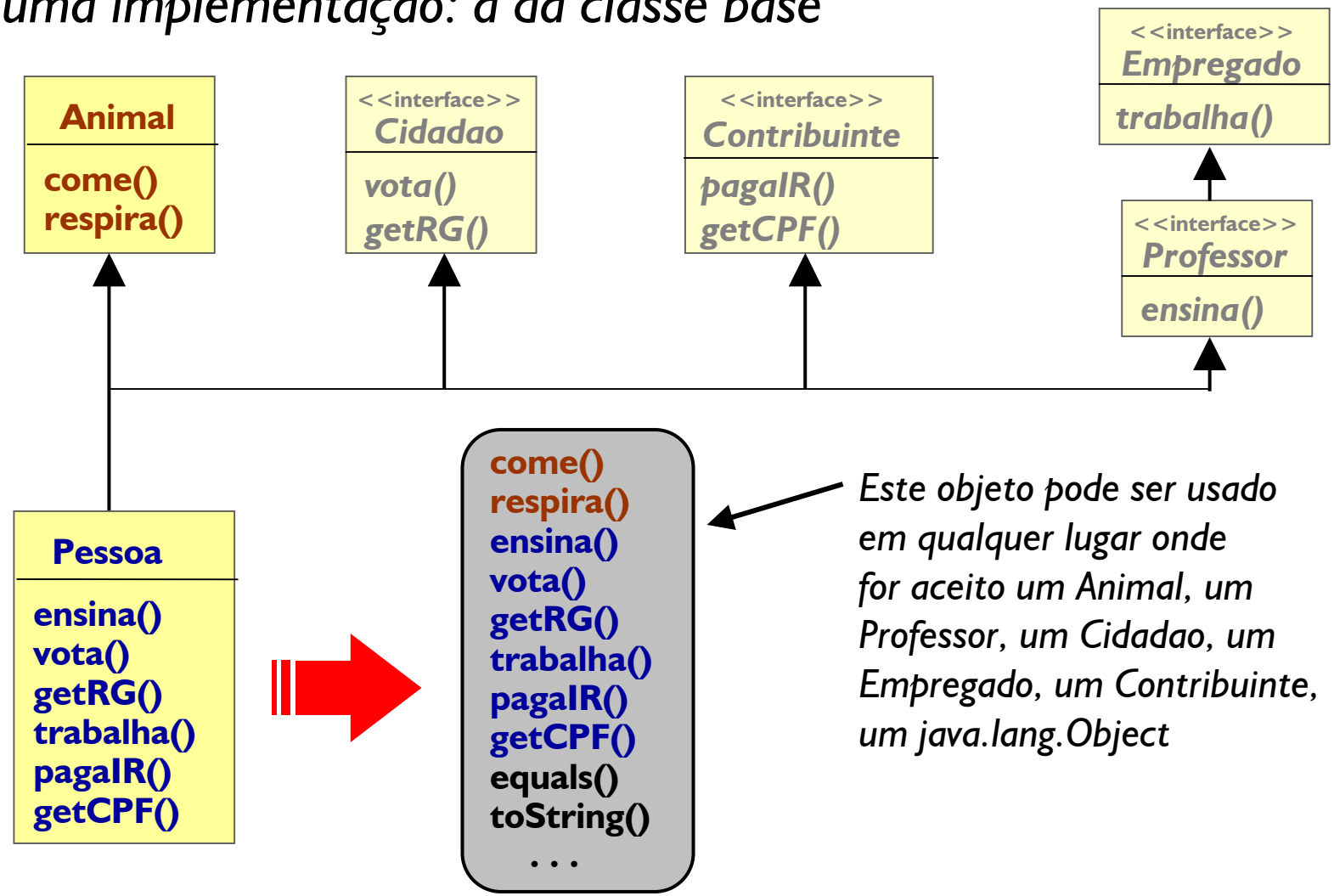
- Interface é uma estrutura que representa uma classe abstrata "pura" em Java
  - Não têm atributos de dados (só pode ter constantes estáticas)
  - Não tem construtor
  - Todos os métodos são abstratos
  - Não é declarada como class, mas como interface
- Interfaces Java servem para fornecer polimorfismo sem herança
  - Uma classe pode "herdar" a interface (assinaturas dos métodos) de várias interfaces Java, mas apenas de uma classe
  - Interfaces, portanto, oferecem um tipo de herança múltipla

Herança múltipla em C++

- Em linguagens como C++, uma classe pode herdar métodos de duas ou mais classes
  - A classe resultante pode ser usada no lugar das suas duas superclasses via upcasting
  - Vantagem de herança múltipla: mais flexibilidade
- Problema
  - Se duas classes A e B estenderem uma mesma classe Z e herdarem um método x() e, uma classe C herdar de A e de B, qual será a implementação de x() que C deve usar? A de A ou de B?
  - Desvantagem de herança múltipla: ambigüidade. Requer código mais complexo para evitar problemas desse tipo

Herança múltipla em Java

- *Classe resultante combina todas as interfaces, mas só possui uma implementação: a da classe base*



Exemplo

```
interface Empregado {
        void trabalha();
}

interface Cidadao {
        void vota();
        int getRG();
}

interface Professor
   extends Empregado {
        void ensina();
}

interface Contribuinte {
        boolean pagaIR();
        long getCPF();
}
```

- *Todos os métodos são implicitamente*
  - *public*
  - *abstract*
- *Quaisquer campos de dados têm que ser inicializadas e são implicitamente*
  - *static*
  - *final (constantes)*
- *Indicar public, static, abstract e final é opcional*
- *Interface pode ser declarada public (default: package-private)*

Exemplo (2)

```
public class Pessoa
        extends Animal
        implements Professor, Cidadao, Contribuinte {

      public void ensina() { /* votar */ }
      public void vota() { /* votar */ }
      public int getRG(){ return 12345; }
      public void trabalha() {}
      public boolean pagaIR() { return false; }
      public long getCPF() { return 1234567890; }
}
```

- *Palavra implements declara interfaces implementadas*
  - *Exige que **cada um dos métodos** de cada interface sejam de fato implementados (na classe atual ou em alguma superclasse)*
  - *Se alguma implementação estiver faltando, classe só compila se for declarada abstract*

Uso de interfaces

```
public class Cidade {
    public void contrata(Professor p) {
        p.ensina();
        p.trabalha();
    }
    public void contrata(Empregado e) { e.trabalha();}
    public void cobraDe(Contribuinte c) { c.pagaIR();}
    public void registra(Cidadao c)     { c.getRG();}
    public void alimenta(Animal a)      { a.come();}

    public static void main (String[] args) {
        Pessoa joao = new Pessoa();
        Cidade sp = new Cidade();
        sp.contrata(joao); // considera Professor
        sp.contrata((Empregado)joao); // Empregado
        sp.cobraDe(joao);  // considera Contribuinte
        sp.registra(joao); // considera Cidadao
        sp.alimenta(joao); // considera Animal
    }
}
```

Conclusão

- *Use interfaces sempre que possível*
  - *Seu código será mais reutilizável!*
  - *Classes que já herdam de outra classe podem ser facilmente redesenhadas para implementar uma interface sem quebrar código existente que a utilize*
- *Planeje suas interfaces com muito cuidado*
  - *É mais fácil evoluir classes concretas que interfaces*
  - *Não é possível acrescentar métodos a uma interface depois que ela já estiver em uso (as classes que a implementam não compilarão mais!)*
  - *Quando a evolução for mais importante que a flexibilidade oferecido pelas interfaces, deve-se usar classes abstratas.*

Exercícios

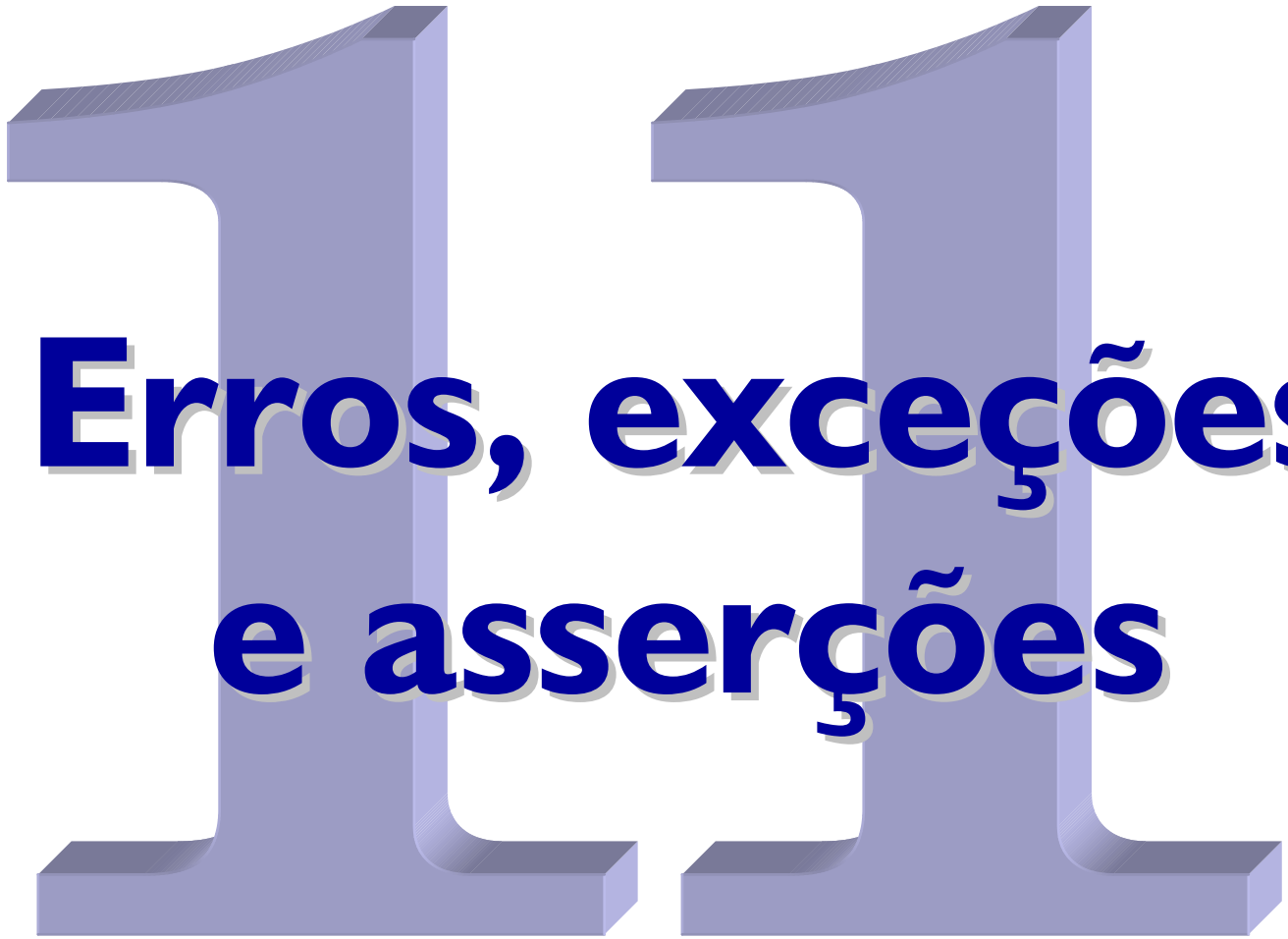
1. Implemente e execute o exemplo mostrado
   - Coloque texto em cada método (um println() para mostrar que o método foi chamado e descrever o que aconteceu)
   - Faça experimentos deixando de implementar certos métodos para ver as mensagens de erro obtidas
2. Implemente o exercício do capítulo 9 com interfaces
   - Mude o nome de RepositorioDados para RepositorioDadosMemoria
   - Crie uma interface RepositorioDados implementada por RepositorioDadosMemoria
   - Altere a linha em que o RepositorioDados é construido na classe Biblioteca e teste a aplicação.

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br



Erros, exceções e asserções

Helder da Rocha
www.argonavis.com.br

Controle de erros com Exceções

- Exceções são
  - Erros de tempo de execução
  - Objetos criados a partir de classes especiais que são "lançados" quando ocorrem condições excepcionais
- Métodos podem capturar ou deixar passar exceções que ocorrerem em seu corpo
  - É obrigatório, para a maior parte das exceções, que o método declare quaisquer exceções que ele não capturar
- Mecanismo try-catch é usado para tentar capturar exceções enquanto elas passam por métodos

Três tipos de erros de tempo de execução

- 1. Erros de lógica de programação
  - Ex: limites do vetor ultrapassados, divisão por zero
  - Devem ser corrigidos pelo programador
- 2. Erros devido a condições do ambiente de execução
  - Ex: arquivo não encontrado, rede fora do ar, etc.
  - Fogem do controle do programador mas podem ser contornados em tempo de execução
- 3. Erros graves onde não adianta tentar recuperação
  - Ex: falta de memória, erro interno do JVM
  - Fogem do controle do programador e não podem ser contornados

Como causar uma exceção?

- *Uma exceção é um tipo de objeto que sinaliza que uma condição excepcional ocorreu*
  - *A identificação (nome da classe) é sua parte mais importante*
- *Precisa ser criada com* ***new*** *e depois lançada com* ***throw***
  - ```
    IllegalArgumentException e = new
                IllegalArgumentException("Erro!");
    throw e;  // exceção foi lançada!
    ```
- *A referência é desnecessária. A sintaxe abaixo é mais usual:*
  - ```
    throw new IllegalArgumentException("Erro!");
    ```

Exceções e métodos

- *Uma declaração **throws** (observe o "s") é obrigatória em métodos e construtores que deixam de capturar uma ou mais exceções que ocorrem em seu interior*

```
public void m() throws Excecao1, Excecao2 {...}
public Circulo() throws ExcecaoDeLimite {...}
```

- ***throws** declara que o método pode provocar exceções do tipo declarado (ou de qualquer subtipo)*
  - *A declaração abaixo declara que o método pode provocar qualquer exceção (nunca faça isto)*

```
public void m() throws Exception {...}
```

- *Métodos sobrepostos não podem provocar mais exceções que os métodos originais*

O que acontece?

- Uma exceção lançada interrompe o fluxo normal do programa
  - O fluxo do programa segue a exceção
  - Se o método onde ela ocorrer não a capturar, ela será propagada para o método que chamar esse método e assim por diante
  - Se ninguém capturar a exceção, ela irá causar o término da aplicação
  - Se em algum lugar ela for capturada, o controle pode ser recuperado

Captura e declaração de exceções

```
public class RelatorioFinanceiro {
    public void metodoMau() throws ExcecaoContabil {
        if (!dadosCorretos) {
            throw new ExcecaoContabil("Dados Incorretos");
        }
    }
    public void metodoBom() {
        try {
            ... instruções ...
            metodoMau();
            ... instruções ...
        } catch (ExcecaoContabil ex) {
            System.out.println("Erro: " + ex.getMessage());
        }
        ... instruções ...
    }
}
```

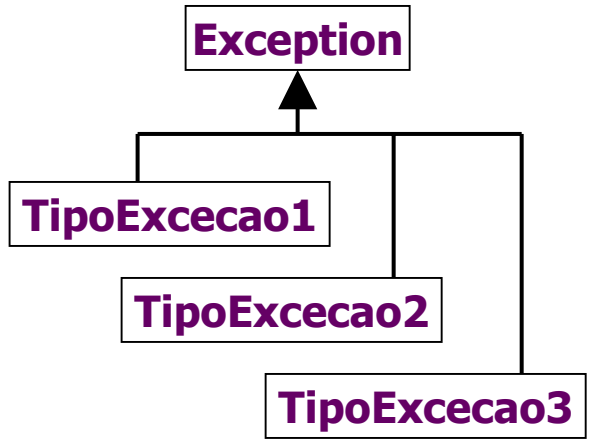
instruções que sempre serão executadas

instruções serão executadas se exceção não ocorrer

instruções serão executadas se exceção não ocorrer ou se ocorrer e for capturada

try e catch

- O bloco try "tenta" executar um bloco de código que pode causar exceção
- Deve ser seguido por
  - Um ou mais blocos catch(TipoDeExcecao ref)
  - E/ou um bloco finally
- Blocos catch recebem tipo de exceção como argumento
  - Se ocorrer uma exceção no try, ela irá descer pelos catch até encontrar um que declare capturar exceções de uma classe ou superclasse da exceção
  - Apenas um dos blocos catch é executado




```
try {
    ... instruções ...
} catch (TipoExcecao1 ex) {
    ... faz alguma coisa ...
} catch (TipoExcecao2 ex) {
    ... faz alguma coisa ...
} catch (Exception ex) {
    ... faz alguma coisa ...
}

... continuação ...
```

finally

- O bloco try não pode aparecer sozinho
  - deve ser seguido por pelo menos um catch ou por um finally
- O bloco finally contém instruções que devem se executadas *independentemente da ocorrência ou não* de exceções

```
try {
  // instruções: executa até linha onde ocorrer exceção
} catch (TipoExcecao1 ex) {
   // executa somente se ocorrer TipoExcecao1

} catch (TipoExcecao2 ex) {
   // executa somente se ocorrer TipoExcecao2
} finally {
   // executa sempre ...
}

// executa se exceção for capturada ou se não ocorrer
```

Como criar uma exceção

- A não ser que você esteja construindo uma API de baixo-nível ou uma ferramenta de desenvolvimento, você so usará exceções do tipo (2) (veja página 3)
- Para criar uma classe que represente sua exceção, basta estender java.lang.Exception:

```
class NovaExcecao extends Exception {}
```

- Não precisa de mais nada. O mais importante é herdar de Exception e fornecer uma identificação diferente
  - Bloco catch usa nome da classe para identificar exceções.

Como criar uma exceção (2)

- *Você também pode acrescentar métodos, campos de dados e construtores como em qualquer classe.*
- *É comum é criar a classe com dois construtores*

```
class NovaExcecao extends Exception {
    public NovaExcecao () {}
    public NovaExcecao (String mensagem) {
        super(mensagem);
    }
}
```

- *Esta implementação permite passar mensagem que será lida através de toString() e getMessage()*

Principais métodos

- Construtores de Exception
  - Exception ()
  - Exception (String message)
  - Exception (String message, Throwable cause) [Java 1.4]
- Métodos de Exception
  - String getMessage()
    - Retorna mensagem passada pelo construtor
  - Throwable getCause()
    - Retorna exceção que causou esta exceção [Java 1.4]
  - String toString()
    - Retorna nome da exceção e mensagem
  - void printStackTrace()
    - Imprime detalhes (stack trace) sobre exceção

Como pegar qualquer exceção

- Se, entre os blocos catch, houver exceções da mesma hierarquia de classes, as classes mais específicas (que estão mais abaixo na hierarquia) devem aparecer primeiro
  - Se uma classe genérica (ex: Exception) aparecer antes de uma mais específica, uma exceção do tipo da específica jamais será capturado
  - O compilador detecta a situação acima e não compila o código
- Para pegar qualquer exceção (geralmente isto não é recomendado), faça um catch que pegue Exception

```
catch (Exception e) { ... }
```

Relançar uma exceção

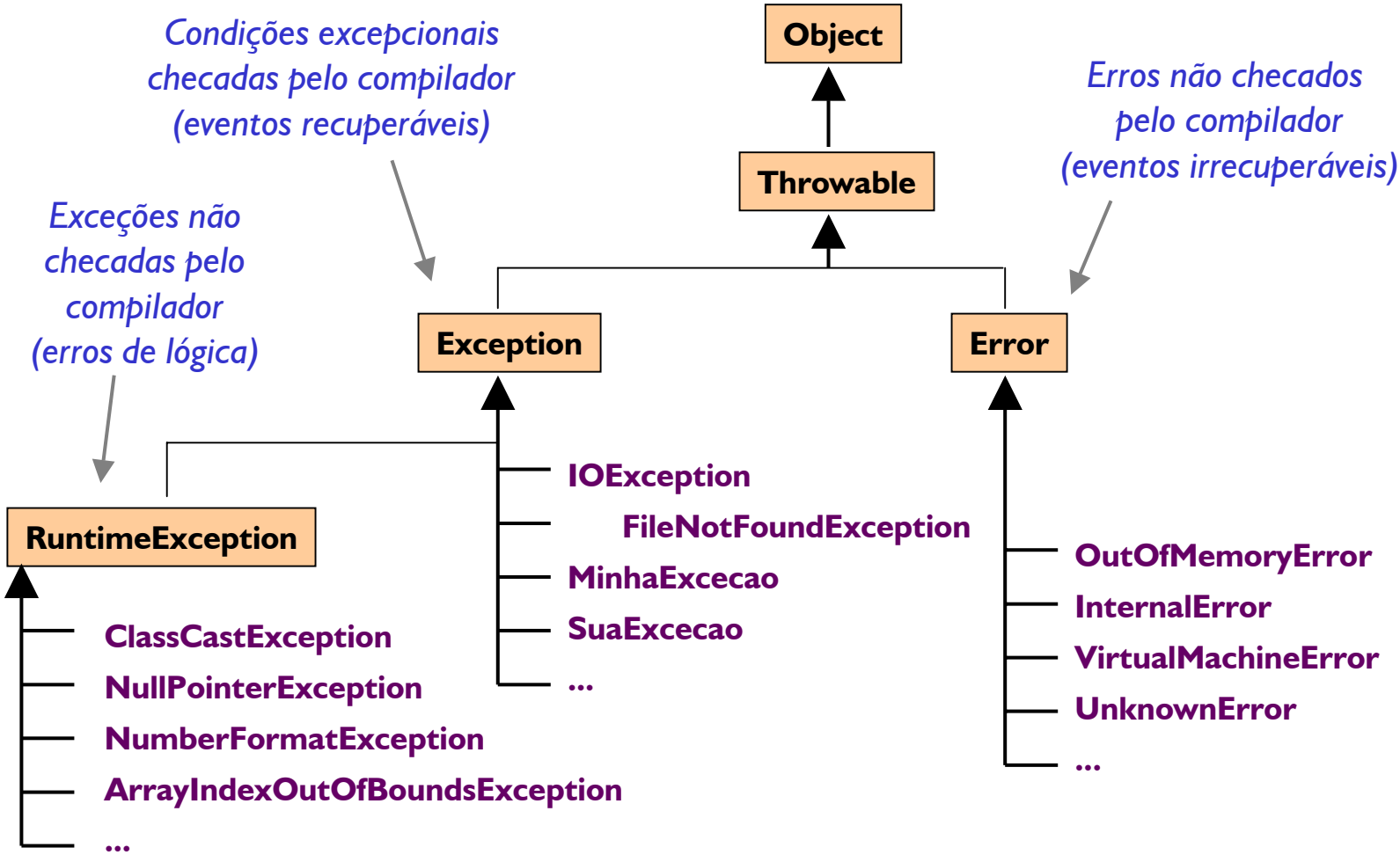
- *Às vezes, após a captura de uma exceção, é desejável relançá-la para que outros métodos lidem com ela*
  - *Isto pode ser feito da seguinte forma*

```
public void metodo() throws ExcecaoSimples {
   try {
     // instruções
   } catch (ExcecaoSimples ex) {
     // faz alguma coisa para lidar com a exceção
     throw ex;  // relança exceção
   }
}
```

Classes base da API

- *RuntimeException* e *Error*
  - *Exceções não verificadas em tempo de compilação*
  - *Subclasses de Error não devem ser capturadas (são situações graves em que a recuperação é impossível ou indesejável)*
  - *Subclasses de RuntimeException representam erros de lógica de programação que devem ser corrigidos (podem, mas não devem ser capturadas: erros devem ser corrigidos)*
- *Exception*
  - *Exceções verificadas em tempo de compilação (exceção à regra são as subclasses de RuntimeException)*
  - *Compilador exige que sejam ou capturadas ou declaradas pelo método que potencialmente as provoca*

Hierarquia



Boa prática: *Prefira sempre usar as classes de exceções existentes na API antes de criar suas próprias exceções!*

Como cavar a própria cova

- *Não tratar exceções e simplesmente declará-las em todos os métodos evita trabalho, mas torna o código menos robusto*
- *Mas o pior que um programador pode fazer é capturar exceções e fazer nada, permitindo que erros graves passem despercebidos e causem problemas dificílimos de localizar no futuro.*
- ***NUNCA** escreva o seguinte código:*

```
try {
// .. código que pode causar exceções
} catch (Exception e) {}
```

- *Ele pega até NullPointerException, e não diz nada. O mundo se acaba, o programa trava ou funciona de modo estranho e ninguém saberá a causa a não ser que mande imprimir o valor de e, entre as chaves do catch.*
- *Pior que isto só se no lugar de Exception houver Throwable.*

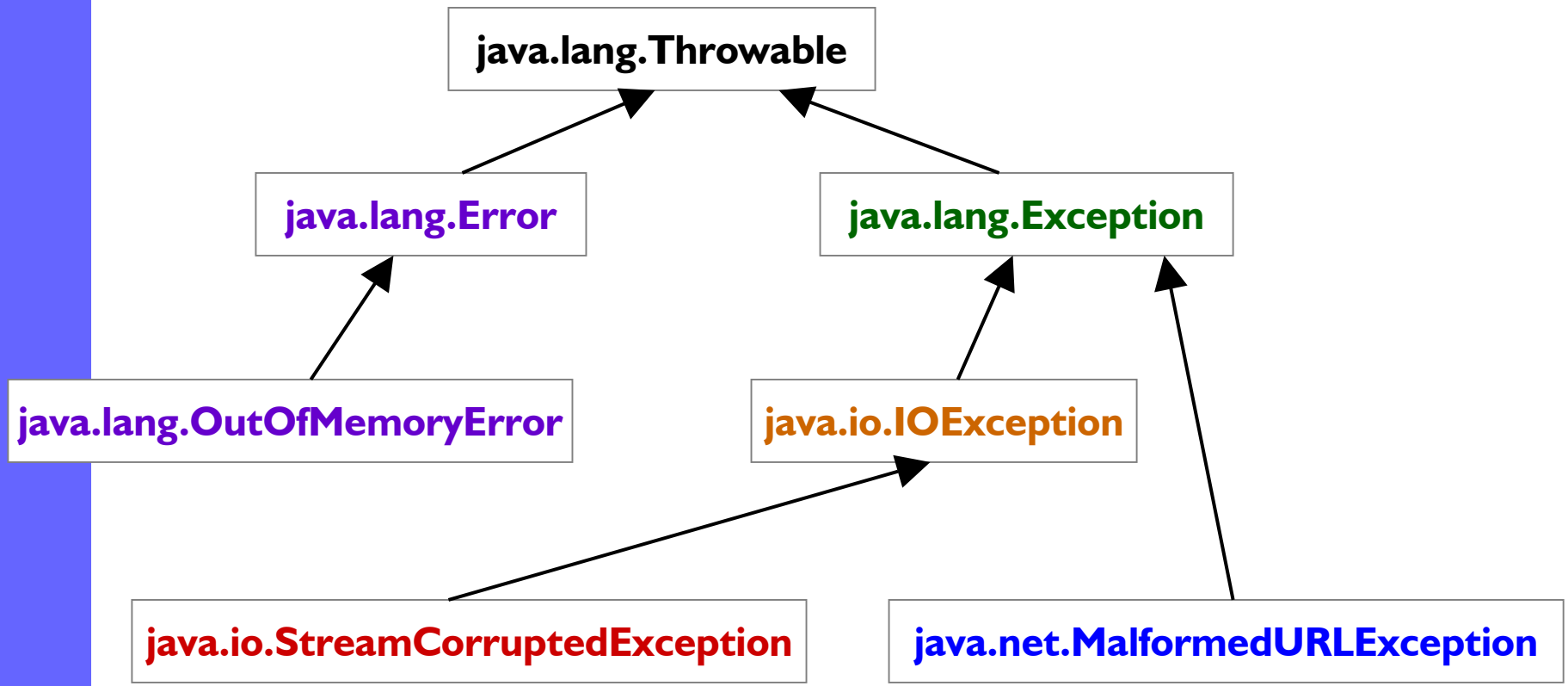
Testes (Enunciado parte 1)

- *Considere o seguinte código* [Roberts]

```
1.  try {
2.    URL u = new URL(s); // s is a previously defined String
3.    Object o = in.readObject(); // in is valid ObjectInputStream
4.    System.out.println("Success");
5.  }
6.  catch (MalformedURLException e) {
7.    System.out.println("Bad URL");
8.  }
9.  catch (IOException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Testes (Enunciado parte 2)

- Considere a seguinte hierarquia



Teste 1

- *1. Que linhas são impressas se os métodos das linhas 2 e 3 completarem com sucesso sem provocar exceções?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Teste 2

- *2. Que linhas são impressas se o método da linha 3 provocar um OutOfMemoryError?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Teste 3

- *3. Que linhas são impressas se o método da linha 2 provocar uma MalformedURLException?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Teste 4

- *4. Que linhas são impressas se o método da linha 3 provocar um StreamCorruptedException?*
  - *A. Success*
  - *B. Bad URL*
  - *C. Bad File Contents*
  - *D. General Exception*
  - *E. Doing finally part*
  - *F. Carrying on*

Asserções

- *São expressões booleanas que o programador define para afirmar uma condição que ele acredita ser verdade*
  - *Asserções são usadas para validar código (ter a certeza que um vetor tem determinado tamanho, ter a certeza que o programa não passou por determinado lugar)*
  - *Melhoram a qualidade do código: tipo de teste caixa-branca*
  - *Devem ser usadas durante o desenvolvimento e desligadas na produção (afeta a performance)*
  - *Não devem ser usadas como parte da lógica do código*
- *Asserções são um recurso novo do JSDK1.4.0*
  - *Nova palavra-chave:* ***assert***
  - *É preciso compilar usando a opção -source 1.4:*

    ```
    > javac -source 1.4 Classe.java
    ```
  - *Para executar, é preciso habilitar afirmações (enable assertions):*

    ```
    > java -ea Classe
    ```

Asserções: sintaxe

- Asserções testam uma condição. Se a condição for falsa, um AssertionError é lançado
- Sintaxe:
  - `assert expressão;`
  - `assert expressãoUm : expressãoDois;`
- Se primeira expressão for true, a segunda não é avaliada
  - Sendo falsa, um AssertionError é lançado e o valor da segunda expressão é passado no seu construtor.
- Exemplo

Em vez de usar comentário...

```
if (i%3 == 0) {
   ...
} else if (i%3 == 1) {
   ...
} else { // (i%3 == 2)
   ...
}
```

... use uma asserção!.

```
if (i%3 == 0) {
   ...
} else if (i%3 == 1) {
   ...
} else {
   assert i%3 == 2;
   ...
}
```

Asserções: exemplo

- *Trecho de código que afirma que controle nunca passará pelo default:*

```
switch(estacao) {
  case Estacao.PRIMAVERA:
    ...
    break;
  case Estacao.VERAO:
    ...
    break;
  case Estacao.OUTONO:
    ...
    break;
  case Estacao.INVERNO:
    ...
    break;
  default:
    assert false: "Controle nunca chegará aqui!";
}
```

Exercícios

- 1. Crie a seguinte hierarquia de exceções
  - ExcecaoDePublicacao
  - AgenteInexistente
  - AgenteDuplicado
  - PublicacaoInexistente extends ExcecaoDePublicacao
  - PublicacaoDuplicada extends ExcecaoDePublicacao
- 2. Quais métodos das classes da aplicação Biblioteca (cap. 9) devem definir essas exceções?
  - Declare (throws) as excecoes nos métodos escolhidos
  - Inclua o teste e lançamento de exceções no seu código (RepositorioDadosMemoria)

Exercícios

- *3. Implemente um procedimento que tenha um switch para quatro estações (primavera: 1, verão: 2, outono: 3, inverno: 4)*
  - *a) Em cada case, coloque um System.out.println() que imprima a estação escolhida. Escreva um main que selecione algumas estações.*
  - *b) Coloque um assert false no default para afirmar que o código jamais deveria passar por lá*
  - *c) Invente uma quinta estação e veja se o AssertError acontece*
- *Não se esqueça de compilar com -source 1.4 e executar com -ea*
  - *(no Ant use o atributos source no <javac>)*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Testes de Unidade com JUnit

Helder da Rocha (helder@acm.org)

Sobre este módulo

- Este módulo é mais sobre boas práticas de desenvolvimento e menos sobre Java
  - Abordagem será superficial (material é extenso), mas visa despertar seu interesse no hábito de escrever testes.
- Objetivos
  - Apresentar e incentivar a prática de testes de unidade durante o desenvolvimento
  - Apresentar a ferramenta JUnit, que ajuda na criação e execução de testes de unidade em Java
  - Discutir as dificuldades relativas à "arte" de testar e como podem ser superadas ou reduzidas
  - Torná-lo(a) uma pessoa "viciada" em testes: Convencê-lo(a) a nunca escrever uma linha sequer de código sem antes escrever um teste executável que a justifique.

O que é "Testar código"?

- É a coisa mais importante do desenvolvimento
  - Se seu código não funciona, ele não presta!
- Todos testam
  - Você testa um objeto quando escreve uma classe e cria algumas instâncias no método main()
  - Seu cliente testa seu software quando ele o utiliza (ele espera que você o tenha testado antes)
- O que são testes automáticos?
  - São programas que avaliam se outro programa funciona como esperado e retornam resposta tipo "sim" ou "não"
  - Ex: um main() que cria um objeto de uma classe testada, chama seus métodos e avalia os resultados
  - Validam os requisitos de um sistema

Por que testar?

- Por que não?
  - Como saber se o recurso funciona sem testar?
  - Como saber se ainda funciona após refatoramento?
- Testes dão maior segurança: coragem para mudar
  - Que adianta a OO isolar a interface da implementação se programador tem medo de mudar a implementação?
  - Código testado é mais confiável
  - Código testado pode ser alterado sem medo
- Como saber quando o projeto está pronto
  - Testes == requisitos 'executáveis'
  - Testes de unidade devem ser executados o tempo todo
  - Escreva os testes antes. Quando todos rodarem 100%, o projeto está concluído!

O que é JUnit?

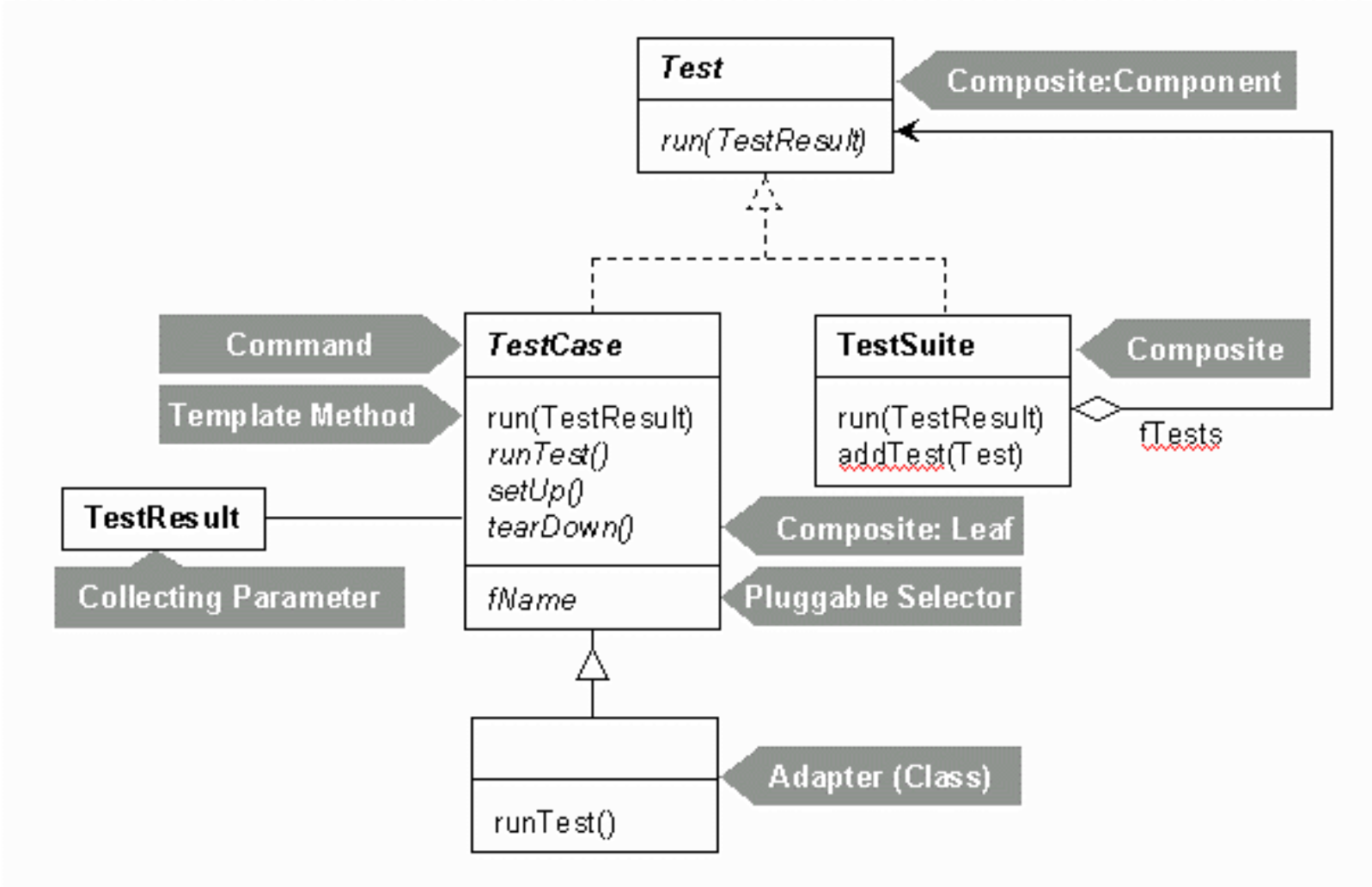
- Um **framework** que facilita o desenvolvimento e execução de **testes de unidade** em código Java
  - Uma API para construir os testes
  - Aplicações para executar testes
- A API
  - Classes Test, TestCase, TestSuite, etc. oferecem a infraestrutura necessária para criar os testes
  - Métodos assertTrue(), assertEquals(), fail(), etc. são usados para testar os resultados
- Aplicação TestRunner
  - Roda testes individuais e suites de testes
  - Versões texto, Swing e AWT
  - Apresenta diagnóstico sucesso/falha e detalhes

Para que serve?

- 'Padrão' para testes de unidade em Java
  - Desenvolvido por Kent Beck (XP) e Erich Gamma (GoF)
  - Design muito simples
- Testar é uma boa prática, mas é chato; JUnit torna as coisas mais agradáveis, facilitando
  - A criação e execução automática de testes
  - A apresentação dos resultados
- JUnit pode verificar se cada unidade de código funciona da forma esperada
  - Permite agrupar e rodar vários testes ao mesmo tempo
  - Na falha, mostra a causa em cada teste
- Serve de base para extensões

Arquitetura do JUnit

- *Diagrama de classes*



Fonte: Manual do JUnit (Cooks Tour)

Como usar o JUnit?

- Há várias formas de usar o JUnit. Depende da metodologia de testes que está sendo usada
  - Código existente: precisa-se escrever testes para classes que já foram implementadas
  - Desenvolvimento guiado por testes (TDD): código novo só é escrito se houver um teste sem funcionar
- Onde obter?
  - www.junit.org
- Como instalar?
  - Incluir o arquivo junit.jar no classpath para compilar e rodar os programas de teste
- Para este curso
  - Inclua o junit.jar no diretório lib/ de seus projetos

JUnit para testar código existente

Exemplo de um roteiro típico

1. Crie uma classe que estenda *junit.framework.TestCase* para cada classe a ser testada

```
import junit.framework.*;
class SuaClasseTest extends TestCase {...}
```

2. Para cada método *xxx(args)* a ser testado defina um método *public void testXxx()* no test case
   - *SuaClasse:*
     - `public boolean equals(Object o) { ... }`
   - *SuaClasseTest:*
     - `public void testEquals() {...}`
- Sobreponha o método *setUp()*, se necessário
- Sobreponha o método *tearDown()*, se necessário

JUnit para guiar o desenvolvimento

Cenário de Test-Driven Development (TDD)

1. Defina uma lista de tarefas a implementar
2. Escreva uma classe (test case) e implemente um método de teste para uma tarefa da lista.
3. Rode o JUnit e certifique-se que o teste falha
4. Implemente o código mais simples que rode o teste
   - Crie classes, métodos, etc. para que código compile
   - Código pode ser código feio, óbvio, mas deve rodar!
5. Refatore o código para remover a duplicação de dados
6. Escreva mais um teste ou refine o teste existente
7. Repita os passos 2 a 6 até implementar toda a lista

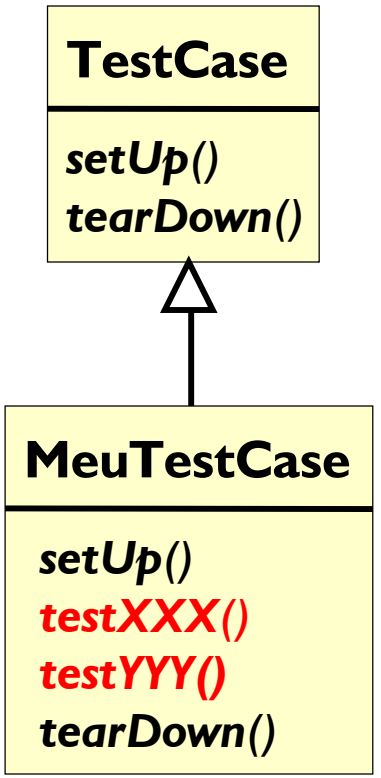
Como implementar?

- *Dentro de cada teste, utilize os métodos herdados da classe TestCase*
  - *assertEquals(objetoEsperado, objetoRecebido),*
  - *assertTrue(valorBooleano), assertNotNull(objeto)*
  - *assertSame(objetoUm, objetoDois), fail (), ...*
- *Exemplo de test case com um setUp() e um teste:*

```
public class CoisaTest extends TestCase {
  // construtor padrão omitido
  private Coisa coisa;
  public void setUp() { coisa = new Coisa("Bit"); }
  public void testToString() {
    assertEquals("<coisa>Bit</coisa>",
                 coisa.toString());
  }
}
```

Como funciona?

- O TestRunner recebe uma subclasse de junit.framework.TestCase
  - Usa reflection (Cap 14) para achar métodos
- Para cada método testXXX(), executa:
  - 1. o método setUp()
  - 2. o próprio método testXXX()
  - 3. o método tearDown()
- O test case é instanciado para executar um método testXXX() de cada vez.
  - As alterações que ele fizer ao estado do objeto não afetarão os demais testes
- Método pode **terminar**, **falhar** ou provocar **exceção**



Exemplo: um test case

```
package junitdemo;

import junit.framework.*;
import java.io.IOException;

public class TextUtilsTest extends TestCase {

    public TextUtilsTest(String name) {
        super(name);
    }

    public void testRemoveWhiteSpaces() throws IOException {
        String testString = "one, ( two | three+) ,       "+
                            "(((four+ |\t  five)?\n \n,  six?";
        String expectedString = "one,(two|three+)"+
                                ",(((four+|five)?,six?";
        String results = TextUtils.removeWhiteSpaces(testString);
        assertEquals(expectedString, results);
    }
}
```

Construtor precisa ser publico, receber String name e chamar `super(String name)` (JUnit 3.7 ou anterior)

*Método começa com "**test**" e é sempre **public void***

Exemplo: uma classe que faz o teste passar

```
package junitdemo;
import java.io.*;

public class TextUtils {

  public static String removeWhiteSpaces(String text)
                                                throws IOException {
      return "one,(two|three+),(((four+|five)?,six?";
  }
}
```

- *O teste passa... e daí? A solução está pronta? Não! Tem dados duplicados! Remova-os!*
- *Escreva um novo teste que faça com que esta solução falhe, por exemplo:*

```
String test2 = " a   b\nc   ";
assertEquals("abc", TextUtils.removeWhiteSpaces(test2));
```

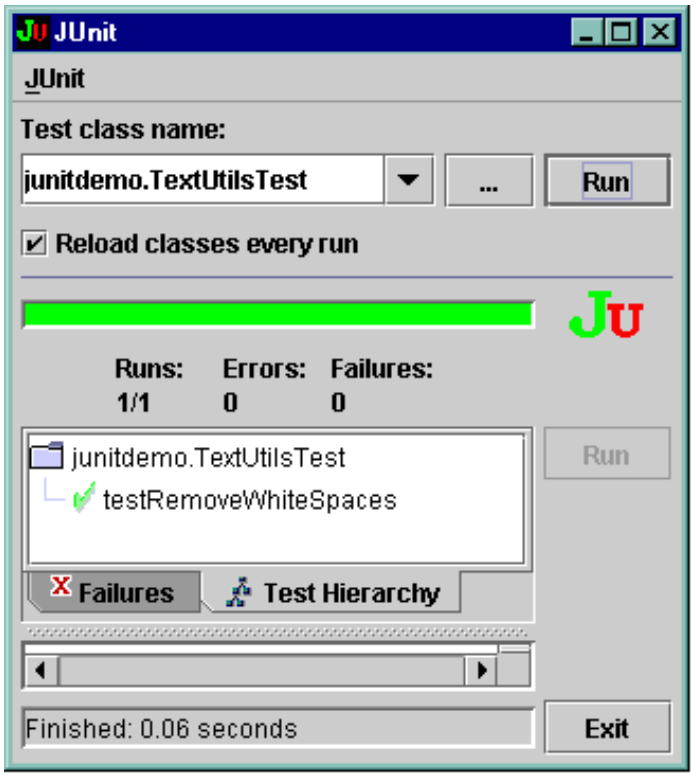
Outra classe que faz o teste passar

```
package junitdemo;
import java.io.*;

public class TextUtils {

  public static String removeWhiteSpaces(String text)
                                                throws IOException {
    StringReader reader = new StringReader(text);
    StringBuffer buffer = new StringBuffer(text.length());
    int c;
    while( (c = reader.read()) != -1) {
      if (c ==' '||c =='\n'||c =='\r'|| c =='\f'||c =='\t') {
         ; /* do nothing */
      } else {
         buffer.append((char)c);
      }
    }
    return buffer.toString();
  }
}
```

Exemplo: como executar

- Use a interface de texto
  - java -cp junit.jar junit.textui.TestRunner
    junitdemo.TextUtilsTest
- Ou use a interface gráfica
  - java -cp junit.jar junit.swingui.TestRunner
    junitdemo.TextUtilsTest
- Use Ant <junit>
  - tarefa do Apache Ant
- Ou forneça um main():

```
public static void main (String[] args) {
    TestSuite suite =
      new TestSuite(TextUtilsTest.class);
    junit.textui.TestRunner.run(suite);
}
```



TestSuite

- *Permite executar uma coleção de testes*
  - *Método addTest(TestSuite) adiciona um teste na lista*
- *Padrão de codificação (usando reflection):*
  - *retornar um TestSuite em cada test-case:*

```
public static TestSuite suite() {
   return new TestSuite(SuaClasseTest.class);
}
```

  - *criar uma classe AllTests que combina as suites:*

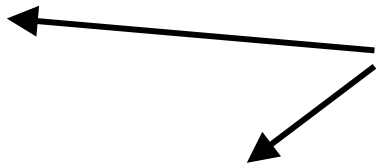
Pode incluir outras suites

```
public class AllTests {
    public static Test suite() {
        TestSuite testSuite =
                          new TestSuite("Roda tudo");
        testSuite.addTest(pacote.AllTests.suite());
        testSuite.addTest(MinhaClasseTest.suite());
        testSuite.addTest(SuaClasseTest.suite());
        return testSuite;
    }
}
```

Fixtures

- *São os dados reutilizados por vários testes*

```
public class AttributeEnumerationTest extends TestCase {
    String   testString;
    String[] testArray;
    AttributeEnumeration testEnum;
    public void setUp() {
        testString  = "(alpha|beta|gamma)";
        testArray = new String[]{"alpha", "beta", "gamma"};
        testEnum = new AttributeEnumeration(testArray);
    }
    public void testGetNames() {
        assertEquals(testEnum.getNames(), testArray);
    }
    public void testToString() {
        assertEquals(testEnum.toString(), testString);
    }
(...)
```



- *Se os mesmos dados são usados em vários testes, inicialize-os no setUp() e faça a faxina no tearDown() (se necessário)*
- *Não perca tempo pensando nisto antes. Escreva seus testes. Depois, se achar que há duplicação, monte o fixture.*

Teste situações de falha

- *É tão importante testar o cenário de falha do seu codigo quanto o sucesso*
- *Método fail() provoca uma falha*
  - *Use para verificar se exceções ocorrem quando se espera que elas ocorram*
- *Exemplo*

```
public void testEntityNotFoundException() {
    resetEntityTable(); // no entities to resolve!
    try {
        // Following method call must cause exception!
        ParameterEntityTag tag = parser.resolveEntity("bogus");
        fail("Should have caused EntityNotFoundException!");
    } catch (EntityNotFoundException e) {
         // success: exception occurred as expected
    }
}
```

JUnit vs. asserções

- Afirmações do J2SDK 1.4 são usadas dentro do código
  - Podem incluir testes dentro da lógica procedural de um programa

```
if (i%3 == 0) {
   doThis();
} else if (i%3 == 1) {
   doThat();
} else {
   assert i%3 == 2: "Erro interno!";
}
```

  - Provocam um AssertionError quando falham (que pode ser encapsulado pelas exceções do JUnit)
- Afirmações do JUnit são usadas em classe separada (TestCase)
  - Não têm acesso ao interior dos métodos (verificam se a interface dos métodos funciona como esperado)
- Afirmações do J2SDK1.4 e JUnit são complementares
  - JUnit testa a interface dos métodos
  - assert testa trechos de lógica dentro dos métodos

Limitações do JUnit

- Acesso aos dados de métodos sob teste
  - Métodos private e variáveis locais não podem ser testadas com JUnit.
  - Dados devem ser pelo menos package-private (friendly)
- Soluções com refatoramento
  - Isolar em métodos private apenas código inquebrável
  - Transformar métodos private em package-private
    - Desvantagem: quebra ou redução do encapsulamento
    - Classes de teste devem estar no mesmo pacote que as classes testadas para ter acesso
- Solução usando extensão do JUnit (open-source)
  - **JUnitX**: usa reflection para ter acesso a dados private
  - http://www.extreme-java.de/junitx/index.html

Resumo: JUnit

- *Para o JUnit,*
  - *Um teste é um método*
  - *Um caso de teste é uma classe contendo uma coleção de testes (métodos que possuem assertions)*
  - *Cada teste testa o comportamento de uma unidade de código do objeto testado (pode ser um método, mas pode haver vários testes para o mesmo método ou um teste para todo o objeto)*
- *Fixtures são os dados usados em testes*
- *TestSuite é uma composição de casos de teste*
  - *Pode-se agrupar vários casos de teste em uma suite*
- *JUnit testa apenas a interface das classes*
  - *Mantenha os casos de teste no mesmo diretório que as classes testadas para ter acesso a métodos package-private*
  - *Use padrões de nomenclatura: ClasseTest, AllTests*
  - *Use o Ant para separar as classes em um release*

Apêndice: boas práticas e dificuldades com testes

Como escrever bons testes

- JUnit facilita bastante a criação e execução de testes, mas elaborar bons testes exige mais
  - O que testar? Como saber se testes estão completos?
- "Teste tudo o que pode falhar" [2]
  - Métodos triviais (get/set) não precisam ser testados.
  - E se houver uma rotina de validação no método set?
- É melhor ter testes a mais que testes a menos
  - Escreva testes curtos (quebre testes maiores)
  - Use assertNotNull() (reduz drasticamente erros de NullPointerException difíceis de encontrar)
  - Reescreva seu código para que fique mais fácil de testar

Como descobrir testes?

- Escreva listas de tarefas (to-do list)
  - Comece pelas mais simples e deixe os testes "realistas" para o final
  - Requerimentos, use-cases, diagramas UML: rescreva os requerimentos em termos de testes
- Dados
  - Use apenas dados suficientes (não teste 10 condições se três forem suficientes)
- Bugs revelam testes
  - Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)!
- Teste sempre! Não escreva uma linha de código sem antes escrever um teste!

Test-Driven Development (TDD)

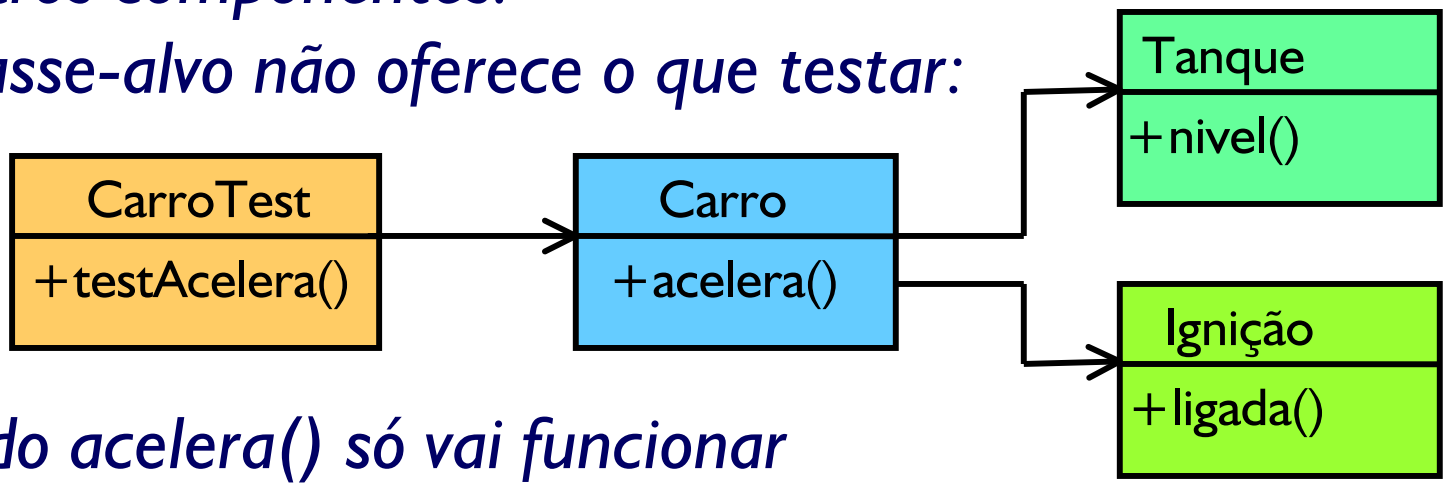
- Desenvolvimento guiado pelos testes
  - Só escreva código novo se um teste falhar
  - Refatore até que o teste funcione
  - Alternância: "red/green/refactor" - nunca passe mais de 10 minutos sem que a barra do JUnit fique verde.
- Técnicas
  - "Fake It Til You Make It": faça um teste rodar simplesmente fazendo método retornar constante
  - Triangulação: abstraia o código apenas quando houver dois ou mais testes que esperam respostas diferentes
  - Implementação óbvia: se operações são simples, implemente-as e faça que os testes rodem

Como lidar com testes difíceis

- Testes devem ser simples e suficientes
  - **XP**: design mais simples que resolva o problema; sempre pode-se escrever novos testes, quando necessário
- Não complique
  - Não teste o que é responsabilidade de outra classe/método
  - Assuma que outras classes e métodos funcionam
- Testes difíceis (ou que parecem difíceis)
  - Aplicações gráficas: eventos, layouts, threads
  - Objetos inacessíveis, métodos privativos, Singletons
  - Objetos que dependem de outros objetos
  - Objetos cujo estado varia devido a fatores imprevisíveis
- Soluções
  - Alterar o design da aplicação para facilitar os testes
  - Simular dependências usando proxies e stubs

Dependência de código-fonte

- *Problema*
  - *Como testar componente que depende do código de outros componentes?*
  - *Classe-alvo não oferece o que testar:*



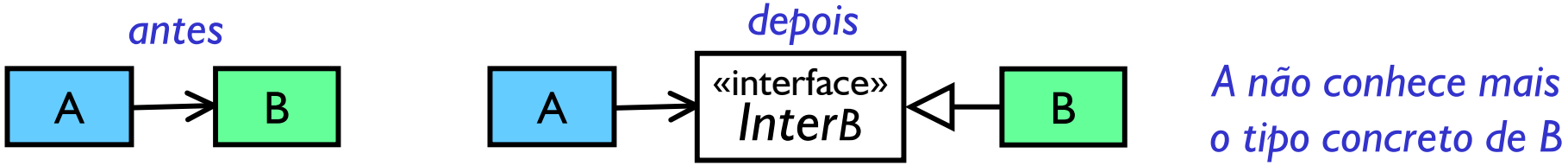
- *Método acelera() só vai funcionar se nível() do tanque for > 0 e ignição estiver ligada()*
- *Como saber se condições são verdadeiras se não temos acesso às dependências?*

```
public void testAcelera() {
    Carro carro =
                new Carro();
    carro.acelera();
    assert???(???);
}
```

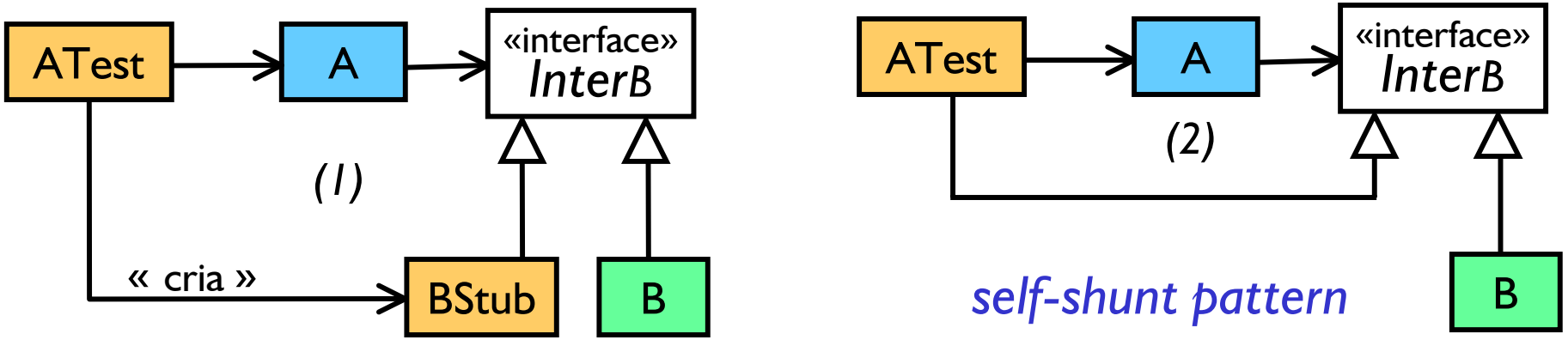
Fonte: www.objectmentor.com, 2002

Stubs: objetos "impostores"

- *É possível remover dependências de código-fonte refatorando o código para usar interfaces*

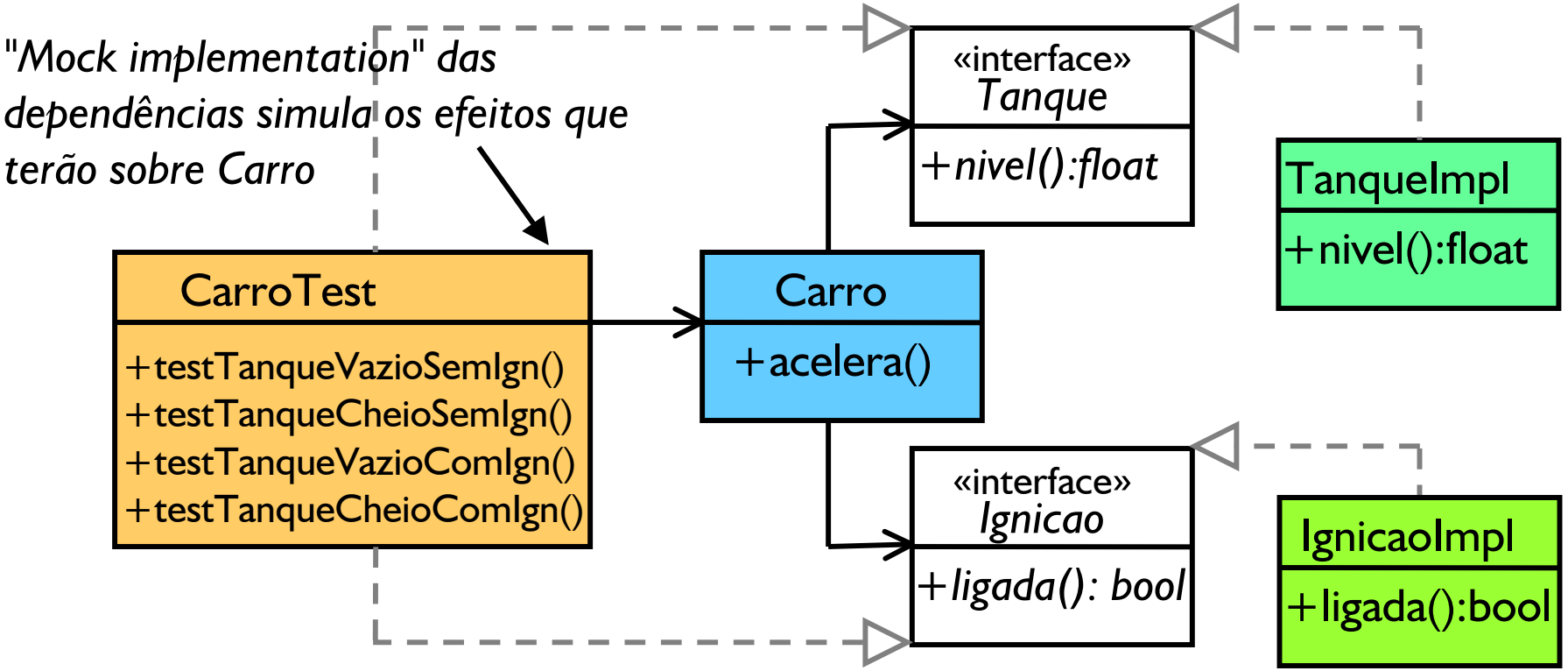


- *Agora B pode ser substituída por um stub*
  - *BStub está sob controle total de ATest (1)*
  - *Em alguns casos, ATest pode implementar InterB (2)*



Fonte: www.objectmentor.com, 2002

Dependência: solução usando stubs



- *Quando criar o objeto, passe a implementação falsa*
  - *carro.setTanque(new CarroTest());*
  - *carro.setIgnicao(new CarroTest());*
- *Depois preencha-a com dados suficientes para que objeto possa ser testado*

Fonte: www.objectmentor.com, 2002

Dependências de servidores

- Usar **stubs** para simular serviços e dados
  - É preciso implementar classes que devolvam as respostas esperadas para diversas situações
  - Complexidade muito grande da dependência pode não compensar investimento (não deixe de fazer testes por causa disto!)
  - Vários tipos de stubs: mock objects, self-shunts.
- Usar **proxies** (mediadores) para serviços reais
  - Oferecem interface para simular comunicação e testa a integração real do componente com seu ambiente
  - Não é teste unitário: teste pode falhar quando código está correto (se os fatores externos falharem)
  - Exemplo em J2EE: **Jakarta Cactus**

Mock Objects

- **Mock objects** (MO) é uma estratégia de uso de stubs que não implementa nenhuma lógica
  - Um mock object não é exatamente um stub, pois não simula o funcionamento do objeto em qualquer situação
- Comportamento é controlado pela classe de teste que
  - Define comportamento esperado (valores retornados, etc.)
  - Passa MO configurado para objeto a ser testado
  - Chama métodos do objeto (que usam o MO)
- Implementações open-source que facilitam uso de MOs
  - EasyMock (tammofreese.de/easymock/) e MockMaker (www.xpdeveloper.com) geram MOs a partir de interfaces
  - Projeto MO (mockobjects.sourceforge.net) coleção de mock objects e utilitários para usá-los

Ant + JUnit

- Com Ant, pode-se executar todos os testes após a integração com um único comando:
  - `ant roda-testes`
- Com as tarefas <junit> e <junitreport> é possível
  - executar todos os testes
  - gerar um relatório simples ou detalhado, em diversos formatos (XML, HTML, etc.)
  - executar testes de integração
- São tarefas opcionais. É preciso ter em `$ANT_HOME/lib`
  - optional.jar (distribuído com Ant)
  - junit.jar (distribuído com JUnit)

Exemplo: <junit>

```
<target name="test" depends="build">
   <junit printsummary="true" dir="${build.dir}"
          fork="true">
      <formatter type="plain" usefile="false" />
      <classpath path="${build.dir}" />
      <test name="argonavis.dtd.AllTests" />
   </junit>
</target>
```

Formata os dados na tela (plain)
Roda apenas arquivo AllTests

```
<target name="batchtest" depends="build" >
   <junit dir="${build.dir}" fork="true">
      <formatter type="xml" usefile="true" />
      <classpath path="${build.dir}" />
      <batchtest todir="${test.report.dir}">
         <fileset dir="${src.dir}">
            <include name="**/*Test.java" />
            <exclude name="**/AllTests.java" />
         </fileset>
      </batchtest>
   </junit>
</target>
```

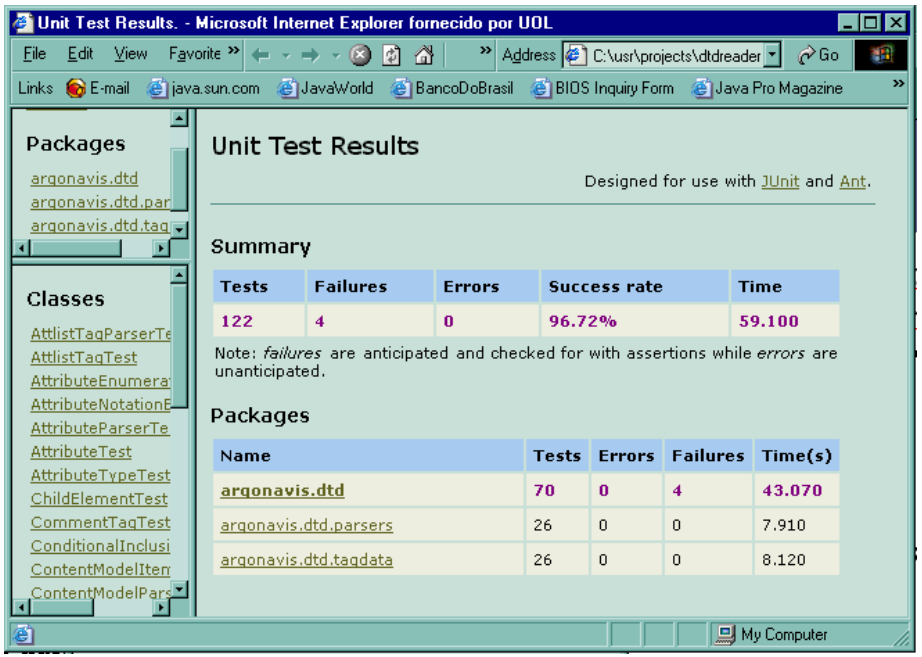
Gera arquivo XML
Inclui todos os arquivos que terminam em TEST.java

<junitreport>

- *Gera um relatório detalhado (estilo JavaDoc) de todos os testes, sucessos, falhas, exceções, tempo, ...*

```
<target name="test-report" depends="batchtest" >
  <junitreport todir="${test.report.dir}">
    <fileset dir="${test.report.dir}">
       <include name="TEST-*.xml" />
    </fileset>
    <report todir="${test.report.dir}/html"
            format="frames" />
  </junitreport>
</target>
```

Usa arquivos XML gerados por <formatter>



Resumo

- *Testar é tarefa essencial do desenvolvimento de software.*
- *Testar unidades de código durante o desenvolvimento é uma prática que traz inúmeros benefícios*
  - *Menos tempo de depuração (muito, muito menos!)*
  - *Melhor qualidade do software*
  - *Segurança para alterar o código*
  - *Usando TDD, melhores estimativas de prazo*
- *JUnit é uma ferramenta open-source que ajuda a implementar testes em projetos Java*
- *TDD ou Test-Driven Development é uma técnica onde os testes são usados para guiar o desenvolvimento*
  - *Ajuda a focar o desenvolvimento em seus objetivos*
- *Mock objects ou stubs podem ser usados para representar dependência e diminuir as responsabilidades de testes*

Exercício

- 1. A classe Exercicio.java possui quatro métodos vazios:

```
int soma(int a, int b)
long fatorial(long n);
double fahrToCelsius(double fahrenheit);
String inverte(String texto);
```

  Escreva, na classe ExercicioTest.java, test-cases para cada método, que preencham os requisitos:
  - **Soma**: 1+1 = 2, 2+4 = 4
  - **Fatorial**: 0! = 1, 1! = 1, 2! = 2, 3! = 6, 4! = 24, 5! = 120
    A fórmula é n! = n(n-1)(n-2)(n-3)...3*2*1
  - **Celsius**: -40C = -40F, 0C=32F, 100C=212F
    A fórmula é F = 9/5 * C + 32
  - **Inverte** recebe "Uma frase" e retorna "esarf amU"
- Implemente um teste e execute-o (o esqueleto de um deles já está pronto). Ele deve falhar.
- Implemente os métodos, um de cada vez, e rode os testes até que não falhem mais (tarja verde), antes de prosseguir.

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

13

Fundamentos de Programação Concorrente

Helder da Rocha
www.argonavis.com.br

Programação concorrente

- O objetivo deste módulo é oferecer uma introdução a Threads que permita o seu uso em aplicações gráficas e de rede
- Tópicos abordados
  - A classe Thread e a interface Runnable
  - Como criar threads
  - Como controlar threads
  - Tópicos sobre deadlock
  - Exemplos de monitores: wait() e notify()
- Para mais detalhes, consulte as referências no final do capítulo

Múltiplas linhas de execução

- Múltiplos threads oferecem uma nova forma de dividir e conquistar um problema de computação
  - Em vez de dividir o problema apenas em objetos independentes ...
  - ... divide o problema em tarefas independentes
- Threads vs. Processos
  - Processos: tarefas em espaços de endereços diferentes se comunicam usando pipes oferecidos pelo SO
  - Threads: tarefas dentro do espaço de endereços da aplicação se comunicam usando pipes fornecidos pela JVM
- O suporte a multithreading de Java é nativo
  - Mais fácil de usar que em outras linguagens onde não é um recurso nativo

O que é um thread

- Um thread parece e age como um programa individual. Threads, em Java, são objetos.
- Individualmente, cada thread faz de conta que tem total poder sobre a CPU
- Sistema garante que, de alguma forma, cada thread tenha acesso à CPU de acordo com
  - Cotas de tempo (time-slicing)
  - Prioridades (preemption)
- Programador pode controlar parcialmente forma de agendamento dos threads
  - Há dependência de plataforma no agendamento

Por que usar múltiplos threads?

- Todo programa tem pelo menos um thread, chamado de Main Thread.
  - O método main() roda no Main Thread
- Em alguns tipos de aplicações, threads adicionais são essenciais. Em outras, podem melhorar o bastante a performance.
- Interfaces gráficas
  - Essencial para ter uma interface do usuário que responda enquanto outra tarefa está sendo executada
- Rede
  - Essencial para que servidor possa continuar a esperar por outros clientes enquanto lida com as requisições de cliente conectado.

Como criar threads

- Há duas estratégias
  - Herdar da classe java.lang.Thread
  - Implementar a interface java.lang.Runnable
- Herdar da classe Thread
  - O objeto é um Thread, e sobrepõe o comportamento padrão associado à classe Thread
- Implementar a interface Runnable
  - O objeto, que define o comportamento da execução, é passado para um Thread que o executa
- Nos dois casos
  - Sobreponha o método run() que é o "main()" do Thread
  - O run() deve conter um loop que irá rodar pelo tempo de vida do thread. Quando o run(), terminar, o thread morre.

Exemplo: herdar da classe Thread

```
public class Trabalhador extends Thread {
    String produto; int tempo;
    public Trabalhador(String produto,
                       int tempo) {
        this.produto = produto;
        this.tempo = tempo;
    }
    public void run() {
      for (int i = 0; i < 50; i++) {
        System.out.println(i + " " + produto);
        try {
          sleep((long)(Math.random() * tempo));
        } catch (InterruptedException e) {}
      }
      System.out.println("Terminei " + produto);
    }
}
```

Exemplo: implementar Runnable

```
public class Operario implements Runnable {
   String produto; int tempo;
   public Operario (String produto,
                    int tempo) {
      this.produto = produto;
      this.tempo = tempo;
   }
   public void run() {
     for (int i = 0; i < 50; i++) {
       System.out.println(i + " " + produto);
       try {
         Thread.sleep((long)
                      (Math.random() * tempo));
       } catch (InterruptedException e) {}
     }
     System.out.println("Terminei " + produto);
   }
}
```

Como usar o Thread

- *Para o Trabalhador que é Thread*

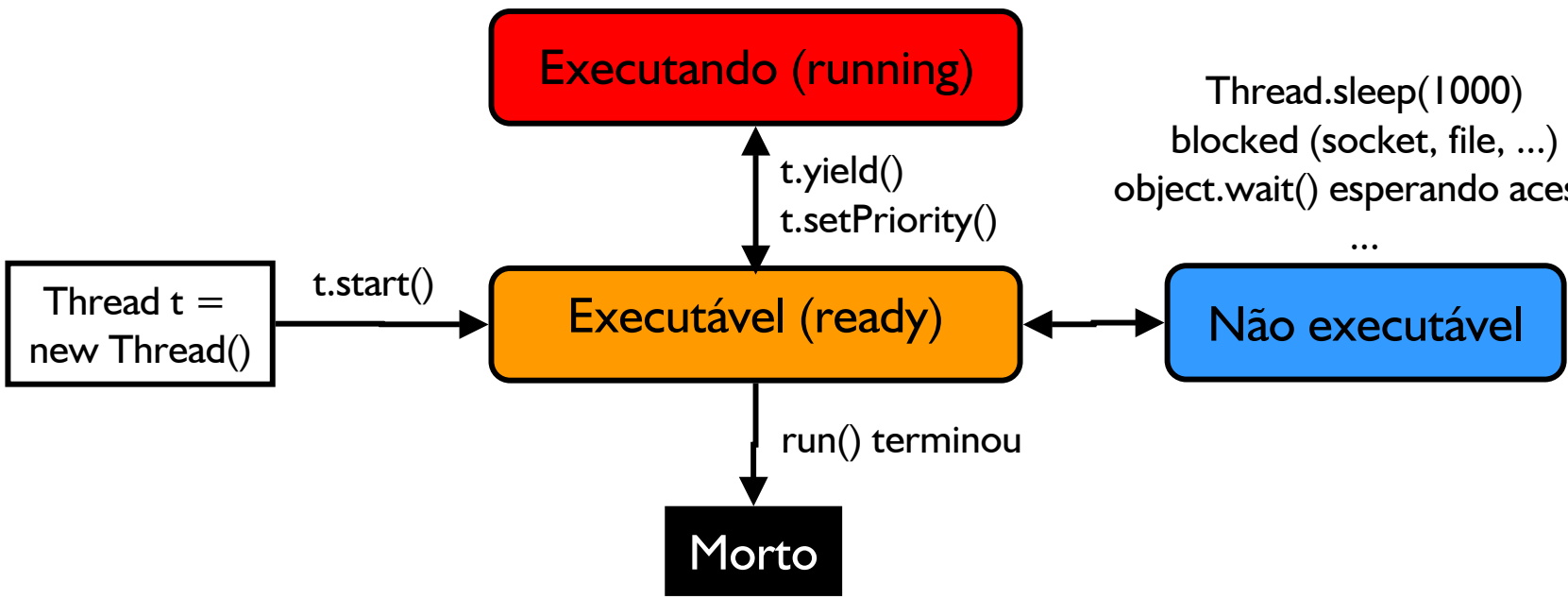
```
Trabalhador severino =
        new Trabalhador("sapato", 100);
Trabalhador raimundo =
        new Trabalhador("bota", 500);
severino.start();
raimundo.start();
```

- *Para o Trabalhador que é Runnable*

```
Operario biu = new Operario ("chinelo", 100);
Operario rai = new Operario ("sandalia", 500);
Thread t1 = new Thread(biu);
Thread t2 = new Thread(rai);
t1.start();
t2.start();
```

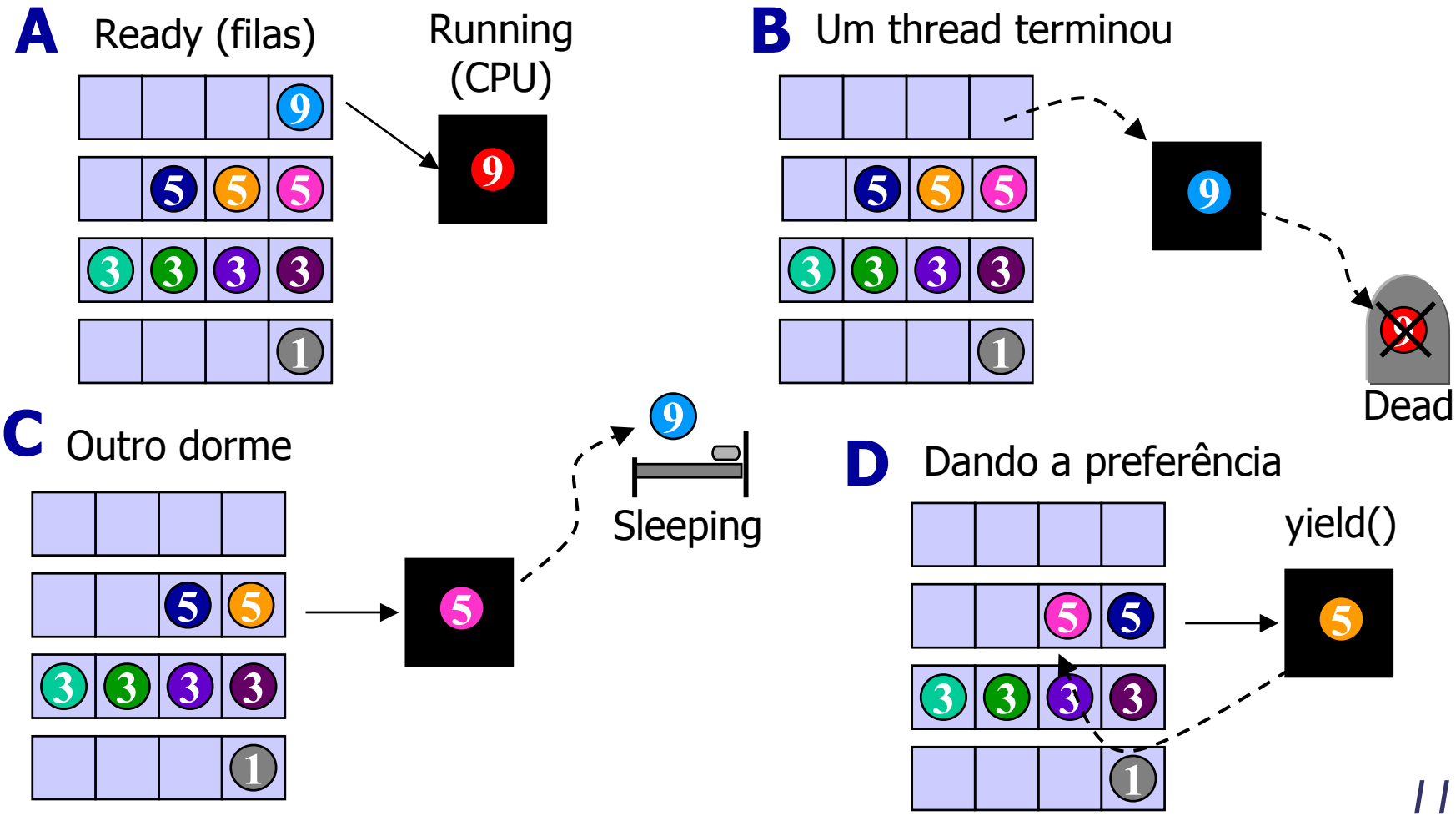
Ciclo de vida

- *Um thread está geralmente em um dentre três estados: executável (e possívelmente executando) e não-executável (esperando, bloqueado, etc.)*
- *O thread entre no estado executável com t.start(), que causa o início de run(), e passa para o estado morto quando run() chega ao fim.*



Filas de prioridades

- *O estado ready não é garantia de execução. Threads são regidos por prioridades. Threads de baixa prioridade têm menos chances*



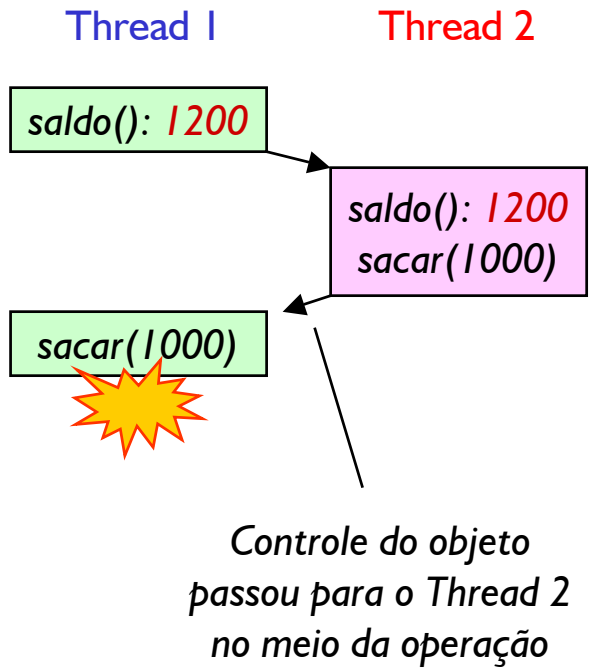
Principais métodos da classe Thread

- Estáticos
  - Thread currentThread(): retorna referência para o thread que está executando no momento
  - void sleep(long tempo): faz com que o thread que está executando no momento pare por tempo milissegundos no mínimo
  - void yield(): faz com que o thread atual pare e permita que outros que estão na sua fila de prioridades executem
- De instância
  - void run(): é o "main" do Thread. Deve ser implementado no Thread ou no objeto Runnable passado ao thread
  - void start(): é um bootstrap. Faz o JVM chamar o run()
  - boolean isAlive(): verifica se thread está vivo

Compartilhamento de recursos limitados

- *Recursos limitados podem ser compartilhados por vários threads simultaneamente*
  - *Cada objeto têm um bloqueio que pode ser acionado pelo método que o modifica para evitar corrupção de dados*
- *Dados podem ser corrompidos se um thread deixar um objeto em um estado incompleto e outro thread assumir a CPU*

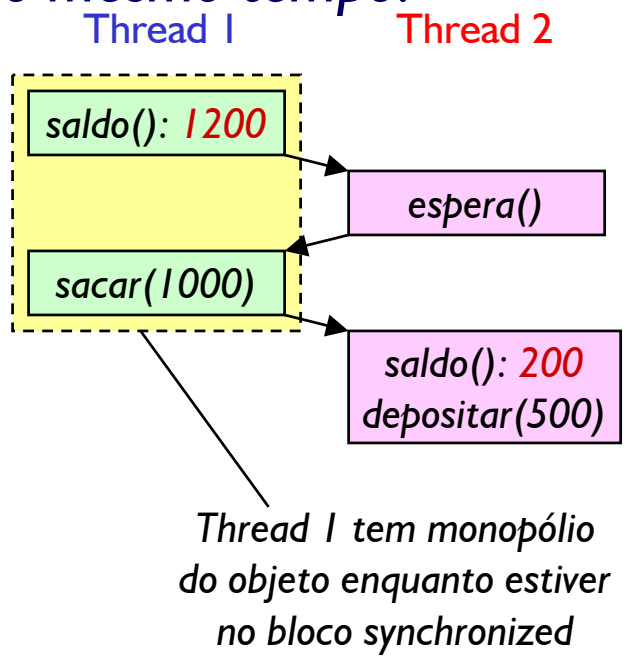
```
void operacaoCritica() {
   if (saldo() > 1000)
     sacar(1000);
   else depositar(500);
}
```



Corrupção

- *Recursos compartilhados devem ser protegidos*
  - *A palavra-reservada synchronized permite que blocos sensíveis ao acesso simultâneo sejam protegidos de corrupção impedindo que objetos os utilizem ao mesmo tempo.*
- *Synchronized deve limitar-se aos trechos críticos (performance!)*

```
void operacaoCritica() {
  // ... trechos thread-safe
  synchronized (this) {
    if (saldo() > 1000)
      sacar(1000);
    else depositar(500);
  } // (... trechos seguros ...)
}
```



- *Métodos inteiros podem ser synchronized*

```
synchronized void operacaoCritica() {}
```

Comunicação entre threads

- Se um recurso crítico está sendo usado, só um thread tem acesso. É preciso que
  - Os outros esperem até que o recurso esteja livre
  - O thread que usa o recurso avise aos outros que o liberou
- Esse controle é possível através de dois métodos da classe Object, que só podem ser usados em blocos synchronized
  - *wait()*: faz com que o Thread sobre o qual é chamado espere por um tempo indeterminado, até receber um...
  - *notify()*: notifica o próximo Thread que o recurso bloqueado foi liberado. Se há mais threads interessados, deve-se usar o
  - *notifyAll()*: avisa a todos os threads.

Exemplo clássico de comunicação (I)

- *A seguinte classe é uma pilha compartilhada por dois threads. Como os métodos push() e pop() contém código que pode corromper os dados, caso não sejam executados atomicamente, eles são synchronized*

```
public class PilhaSincronizada {
   private int index = 0;
   private int[] buffer = new int[10];

   public synchronized int pop() {
      index--;
      return buffer[index];
   }

   public synchronized void push(int i) {
      buffer[index] = i;
      index++;
   }
}
```

Exemplo de comunicação: (2) Produtor

- *O objeto abaixo produz 40 componentes em intervalos de 0 a 1 segundo e os tenta armazenar na pilha.*

```
public class Producer implements Runnable {
   PilhaSincronizada pilha;

   public Producer(PilhaSincronizada pilha) {
      this.pilha = pilha;
   }

   public void run() {
     int colorIdx;
     for (int i = 0; i < 40; i++) {
       colorIdx = (int)(Math.random() * Colors.color.length);
       pilha.push(colorIdx);
       System.out.println("Criado: "+ Colors.color[colorIdx]);
       try {
           Thread.sleep((int)(Math.random() * 1000));
       } catch (InterruptedException e) {}
     }
   }
}
```

Exemplo de comunicação (3) Consumidor

- *O objeto abaixo consome os 40 componentes da pilha mais lentamente, esperando de 0 a 5 segundos*

```
public class Consumer implements Runnable {

   PilhaSincronizada pilha;

   public Producer(PilhaSincronizada pilha) {
      this.pilha = pilha;
   }

   public void run() {
     int colorIdx;
     for (int i = 0; i < 20; i++) {
       colorIdx = pilha.pop();
       System.out.println("Usado: "+ Colors.color[colorIdx]);
       try {
         Thread.sleep((int)(Math.random() * 5000));
       } catch (InterruptedException e) {}
     }
   }
}
```

Monitor com wait() e notify()

- *A pilha foi modificada e agora faz com que os threads executem wait() e notify() ao utilizarem seus métodos*

```
public class PilhaSincronizada {
   private int index = 0;
   private int[] buffer = new int[10];

   public synchronized int pop() {
      while (index == 0) {
         try { this.wait();
         } catch (InterruptedException e) {}
      }
      this.notify();
      index--;
      return buffer[index];
   }

   public synchronized void push(int i) {
      while (index == buffer.length) {
         try { this.wait();
         } catch (InterruptedException e) {}
      }
      this.notify();
      buffer[index] = i;
      index++;
   }
}
```

Apesar de aparecer antes, a notificação só terá efeito quando o bloco synchronized terminar

Problemas de sincronização

- Quando métodos sincronizados chamam outros métodos sincronizados há risco de deadlock
- Exemplo: para alterar valor no objeto C:
  - O Thread A espera liberação de acesso a objeto que está com Thread B
  - O Thread B aguarda que alguém (A, por exemplo) faça uma alteração no objeto para que possa liberá-lo (mas ninguém tem acesso a ele, pois B o monopoliza!)
- Solução
  - Evitar que métodos sincronizados chamem outros métodos sincronizados
  - Se isto não for possível, controlar liberação e retorno dos acessos (hierarquia de chaves de acesso)

Exemplo de deadlock

```
public class Caixa {
    double  saldoCaixa = 0.0;
    Cliente clienteDaVez = null;

    public synchronized void atender(Cliente c, int op, double v) {
        while (clienteDaVez != null) { wait(); } //espera vez
        clienteDaVez = c;
        switch (op) {
            case -1:  sacar(c, v); break;
            case  1:  depositar(c, v); break;
        }
    }

    private synchronized void sacar(Cliente c, double valor) {
        while (saldoCaixa <= valor) { wait(); } //espera saldo, vez
        if (valor > 0) {
           saldoCaixa -= valor; clienteDaVez = null;
           notifyAll();
        }
    }

    private synchronized void depositar(Cliente c, double valor) {
        if (valor > 0 ) {
           saldoCaixa += valor; clienteDaVez = null;
           notifyAll();
        }
    }
}
```

Deadlock (2)

- Cenário 1:
  - Saldo do caixa: R$0.00
  - Cliente 1 (thread) é atendido (recebe acesso do caixa), deposita R$100.00 e libera o caixa
  - Cliente 2 (thread) é atendido (recebe o acesso do caixa), saca R$50.00 e libera o caixa
- Cenário 2: cliente 2 chega antes de cliente 1
  - Saldo do caixa: R$0.00
  - Cliente 2 é atendido (recebe acesso do caixa), tenta sacar R$50.00 mas não há saldo. Cliente 2 espera haver saldo.
  - Cliente 1 tenta ser atendido (quer depositar R$100.00) mas não é sua vez na fila. Cliente 1 espera sua vez.
  - DEADLOCK!

Exercícios

- *1. Implemente e rode o exemplo Trabalhador mostrado neste capítulo*
- *2. Altere a classe para que o Thread rode para sempre*
  - *Crie um método parar() que altere um flag que faça o loop infinito terminar*
- *3. Implemente e rode o exemplo Produtor - Consumidor*
- *4. Implemente o exemplo de deadlock e encontre um meio de evitá-lo.*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

14 Coleções, Propriedades, Resources e Strings

Helder da Rocha
www.argonavis.com.br

Assuntos abordados neste módulo

- Coleções
  - Vetores, comparação e ordenação
  - Listas e Conjuntos
  - Iteradores
  - Mapas
  - Propriedades
- Manipulação de strings
  - Classes String, StringBuffer e StringTokenizer
  - Classes para expressões regulares em Java
- Recursos avançados
  - ClassLoader: resources e reflection (fundamentos)
  - JavaBeans

O que são coleções?

- São estruturas de dados comuns
  - Vetores (listas)
  - Conjuntos
  - Pilhas
  - Árvores binárias
  - Tabelas de hash
  - etc.
- Oferecem formas diferentes de colecionar dados com base em fatores como
  - Eficiência no acesso ou na busca ou na inserção
  - Forma de organização dos dados
  - Forma de acesso, busca, inserção

Java Collections API

- Oferece uma biblioteca de classes e interfaces (no pacote java.util) que
  - Implementa as principais estruturas de dados de forma reutilizável (usando apenas duas interfaces comuns)
  - Oferece implementações de cursor para iteração (Iterator pattern) para extrair dados de qualquer estrutura usando uma única interface
  - Oferece implementações de métodos estáticos utilitários para manipulação de coleções e vetores

Tipos de coleções em Java

- **Vetores**
  - Mecanismo nativo para colecionar valores primitivos e referências para objetos
  - Podem conter objetos (referências) tipos primitivos
  - Forma mais eficiente de manipular coleções
- **Coleções**
  - Não suporta primitivos (somente se forem empacotados dentro de objetos)
  - Classes e interfaces do pacote java.util
  - Interfaces Collection, List, Set e Map e implementações
  - Iterator, classes utilitárias e coleções legadas

Vetores de objetos

- Forma mais eficiente de manter referências
- Características
  - Tamanho fixo. É preciso criar um novo vetor e copiar o conteúdo do antigo para o novo. Vetores não podem ser redimensionados depois de criados.
  - Quantidade máxima de elementos obtida através da propriedade length (comprimento do vetor)
  - Verificados em tempo de execução. Tentativa de acessar índice inexistente provoca, na execução, um erro do tipo ArrayIndexOutOfBoundsException
  - Tipo definido. Pode-se restringir o tipo dos elementos que podem ser armazenados

Vetores são objetos

- *Quando um vetor é criado no heap, ele possui "métodos" e campos de dados como qualquer outro objeto*
- *Diferentes formas de inicializar um vetor*

```
class Coisa {} // uma classe
(...)
Coisa[] a;  // referência do vetor (Coisa[]) é null

Coisa[] b = new Coisa[5]; // referências Coisa null

Coisa[] c = new Coisa[4];
for (int i = 0; i < c.length; i++) {
   c[i] = new Coisa(); // refs. Coisa inicializadas
}

Coisa[] d = {new Coisa(), new Coisa(), new Coisa()};

a = new Coisa[] {new Coisa(), new Coisa()};
```

Como retornar vetores

- *Como qualquer vetor (mesmo de primitivos) é objeto, só é possível manipulá-lo via referências*
  - *Atribuir um vetor a uma variável copia a referência do vetor à variável*

```
int[] vet = intArray; // se intArray for int[]
```

  - *Retornar um vetor através de um método retorna a referência para o vetor*

```
int[] aposta = sena.getDezenas();
```

```
public static int[] getDezenas() {
    int[] dezenas = new int[6];
    for (int i = 0; i < dezenas.length; i++) {
        dezenas[i] = Math.ceil((Math.random()*50));
    }
    return dezenas;
}
```

Como copiar vetores

- *Método utilitário de java.lang.System*

```
static void arraycopy(origem_da_copia, offset,
                      destino_da_copia, offset,
                      num_elementos_a_copiar)
```

  - *Ex:*

```
int[] um = {12, 22, 3};
int[] dois = {9, 8, 7, 6, 5};
System.arraycopy(um, 0, dois, 1, 2);
```

  - *Resultado:* `dois: {9, 12, 22, 6, 5};`

- *Vetores de objetos*
  - *Apenas as referências são copiadas (shallow copy)*

Exercícios

- *1. Vetores e System.arraycopy()*
  - *(a) Crie dois vetores de inteiros. Um com 10 elementos e outro com 20.*
  - *(b) Preencha o primeiro com uma seqüência e o segundo com uma série exponencial.*
  - *(c) Crie uma função estática que receba um vetor e retorne uma String da forma "[a1, a2, a3]" onde a* são elementos do vetor.*
  - *(d) Imprima os dois vetores.*
  - *(e) Copie um vetor para o outro e imprima novamente.*
  - *(f) experimente mudar os offsets e veja as mensagens obtidas.*

java.util.Arrays

- Classe utilitária com diversos métodos estáticos para manipulação de vetores
- Métodos suportam vetores de quaisquer tipo
- Principais métodos (sobrecarregados p/ vários tipos)
  - void Arrays.sort(vetor)

    - Usa Quicksort para primitivos

    - Usa Mergesort para objetos (classe do objeto deve implementar a interface Comparable)
  - boolean Arrays.equals(vetor1, vetor2)
  - int Arrays.binarySearch(vetor, chave)
  - void Arrays.fill(vetor, valor)

Ordenação

- Para ordenar objetos é preciso compará-los.
- Como estabelecer os critérios de comparação?
  - equals() apenas informa se um objeto é igual a outro, mas não informa se "é maior" ou "menor"
- Solução: interface java.lang.Comparable
  - Método a implementar:

```
public int compareTo(Object obj);
```

- Para implementar, retorne
  - Um inteiro menor que zero se objeto atual for "menor" que o recebido como parâmetro
  - Um inteiro maior que zero se objeto atual for "maior" que o recebido como parâmetro
  - Zero se objetos forem iguais

Exemplo: java.lang.Comparable

```
public class Coisa implements Comparable {
  private int id;
  public Coisa(int id) {
    this.id = id;
  }
  public int compareTo(Object obj) {
    Coisa outra = (Coisa) obj;
    if (id >  outra.id) return 1;
    if (id <  outra.id) return -1;
    if (id == outra.id) return 0;
  }
}
```

- *Como usar*

```
Coisa c1 = new Coisa(123);
Coisa c2 = new Coisa(456);
if (c1.compareTo(c2)==0) System.out.println("igual");
Coisa coisas[] = {c2, c1, new Coisa(3)};
Arrays.sort(coisas);
```

Usa compareTo()

Comparator

- *Comparable exige que a classe do objeto a ser comparado implemente uma interface*
  - *E se uma classe inacessível não a implementa?*
  - *O que fazer se você não tem acesso para modificar ou estender a classe?*
- *Solução: interface utilitária java.util.Comparator*
  - *Crie uma classe utilitária que implemente Comparator e passe-a como segundo argumento de Arrays.sort().*
- *Método a implementar:*

```
public int compare(Object o1, Object o2);
```

Exemplo: java.util.Comparator

```
public class MedeCoisas implements Comparator {
  public int compare(Object o1, Object o2) {
    Coisa c1 = (Coisa) o1;
    Coisa c2 = (Coisa) o2;
    if (c1.id <  c2.id) return 1;
    if (c1.id >  c2.id) return -1;
    if (c1.id == c2.id) return 0;
  }
}
```

Ordenando ao contrário

- *Como usar*

```
Coisa c1 = new Coisa(123);
Coisa c2 = new Coisa(456);
Comparator comp = new MedeCoisas();
if(comp.compare(c1, c2)==0) System.out.println("igual");
Coisa coisas[] = {c2, c1, new Coisa(3)};
Arrays.sort(coisas, new MedeCoisas());
```

Usa compare() de MedeCoisas que tem precedência sobre compareTo() de Coisa

Não confunda Comparator e Comparable

- Ao projetar classes novas, considere sempre implementar java.lang.Comparable
  - Objetos poderão ser ordenados mais facilmente
  - Critério de ordenação faz parte do objeto
  - compareTo() compara objeto atual com um outro
- java.util.Comparator não faz parte do objeto comparado
  - Implementação de Comparator é uma classe utilitária
  - Use quando objetos não forem Comparable ou quando não quiser usar critério de ordenação original do objeto
  - compare() compara dois objetos recebidos

Outras funções úteis de Arrays

`boolean equals(vetor1, vetor2)`

- *Retorna true apenas se vetores tiverem o mesmo conteúdo (mesmas referências) na mesma ordem*
- *Só vale para comparar vetores do mesmo tipo primitivo ou vetores de objetos*

`void fill(vetor, valor)`

Não serve para referências!

- *Preenche o vetor (ou parte do vetor) com o valor passado como argumento (tipo deve ser compatível)*

`int binarySearch(vetor, valor)`

- *Retorna inteiro com posição do valor no vetor ou valor negativo com a posição onde deveria estar*
- *Não funciona se o vetor não estiver ordenado*
- *Se houver valores duplicados não garante qual irá ser localizado*

Exercícios (2)

- 2. interface java.lang.Comparable
  - a) Use ou crie uma classe Circulo que tenha x, y e raio inteiros e um construtor que receba os três parâmetros para inicializar um novo Circulo. A classe deve ter métodos equals() e toString().
  - b) Faça o Circulo implementar Comparable. Use o raio como critério de ordenação
  - c) Em um main(), crie um vetor de 10 Círculos de tamanho diferente. Coloque-os em ordem e imprima o resultado

Exercícios (3)

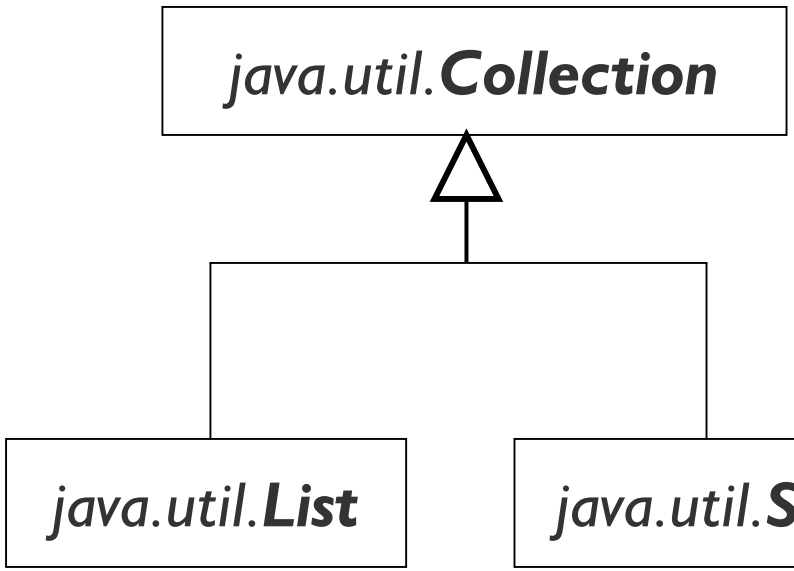
- 3. Escreva um java.util.Comparator que ordene Strings de acordo com o último caractere
  - Crie uma classe que implemente Comparator (LastCharComparator)
  - Use s.charAt(s.length() -1) (método de String s) para obter o último caractere e usá-lo em compare()
  - Use um vetor de palavras. Imprima o vetor na ordem natural, uma palavra por linha
  - Rode o Arrays.sort() usando o Comparator que você criou e imprima o vetor novamente

Coleções

- Classes e interfaces do pacote java.util que representam listas, conjuntos e mapas
- Solução flexível para armazenar objetos
  - Quantidade armazenada de objetos não é fixa, como ocorre com vetores
- Poucas interfaces (duas servem de base) permitem maior reuso e um vocabulário menor de métodos
  - add(), remove() - principais métodos de interface Collection
  - put(), get() - principais métodos de interface Map
- Implementações parciais (abstratas) disponíveis para cada interface
- Há duas ou três implementações de cada interface

As interfaces

Coleções de elementos individuais



- *seqüência definida*
- *elementos indexados*

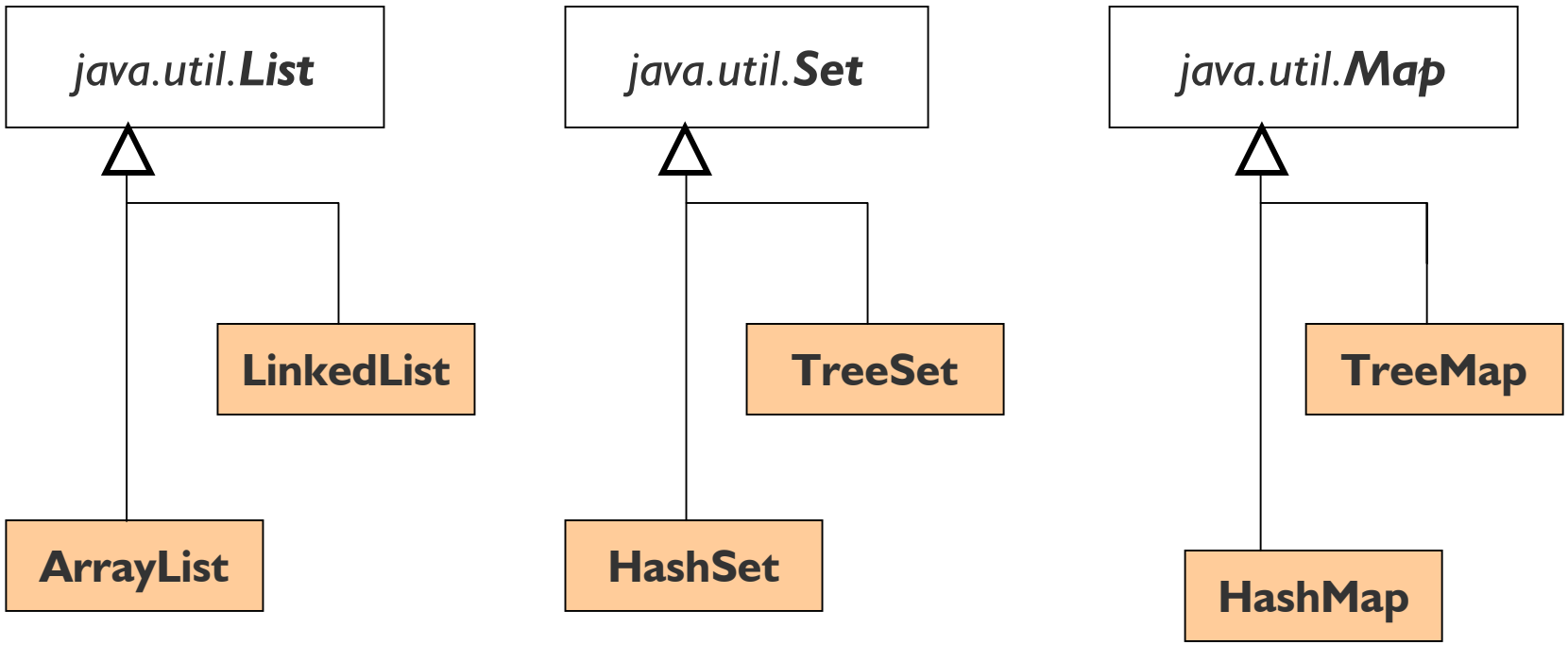
- *seqüência arbitrária*
- *elementos não repetem*

Coleções de pares de elementos

*java.util.**Map***

- *Pares chave/valor (vetor associativo)*
- ***Collection** de valores (podem repetir)*
- ***Set** de chaves (unívocas)*

Principais implementações concretas



- *Alguns detalhes foram omitidos:*
  - *Classes abstratas intermediárias*
  - *Interfaces intermediárias*
  - *Implementações menos usadas*

Desvantagens das Coleções

- *Menos eficientes que vetores*
- *Não aceitam tipos primitivos (só empacotados)*
- *Não permitem restringir o tipo específico dos objetos guardados (tudo é java.lang.Object)*
  - *Aceitam qualquer objeto: uma coleção de Galinhas aceita objetos do tipo Raposa*
  - *Requer cast na saída para poder usar objeto*

```
List galinheiro = new ArrayList();
galinheiro.add(new Galinha("Chocagilda"));
galinheiro.add(new Galinha("Cocotalva"));
galinheiro.add(new Raposa("Galius"));
for (int i = 0; i < galinheiro.size(); i++) {
   Galinha g = (Galinha) galinheiro.get(i);
   g.ciscar();
}
```

Ocorrerá ClassCastException quando Object retornado apontar para uma Raposa e não para uma Galinha

Typesafe Collection

- *Pode-se criar uma implementação de coleção que restringe o tipo de objetos que aceita usando delegação*
- *A classe abaixo é uma coleção type-safe, que não permite a entrada de objetos que não sejam do tipo Galinha ou descendentes*
  - *Para recuperar objetos, não é necessário usar cast!*

```
import java.util.*;
public class Galinheiro {
  private List galinhas;
  public Galinheiro(List list) {
    galinhas = list;
  }
  public get(int idx) {
    return (Galinha) galinhas.get(idx);
  }
  public add(Galinha g) {
    galinhas.add(g);
  }
}
```

Não compila!

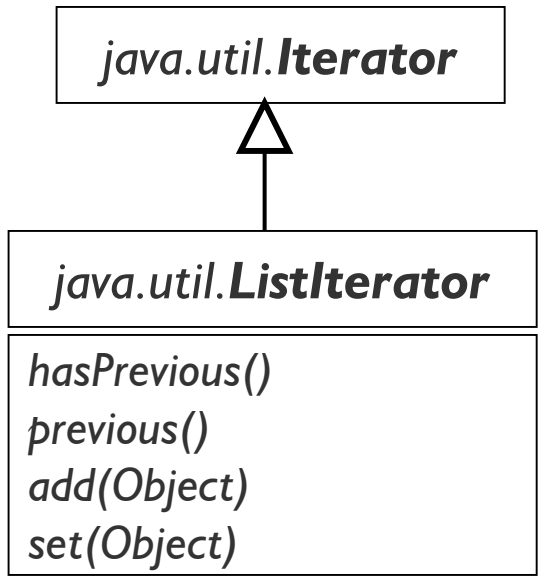
Não requer cast!

```
Galinheiro g =
  new Galinheiro(new ArrayList());
g.add(new Galinha("Frida"));
// g.add(new Raposa("Max"));
Galinha frida = g.get(0);
```

Interface Iterator

- *Para navegar dentro de uma Collection e selecionar cada objeto em determinada seqüência*
  - *Uma coleção pode ter vários Iterators*
  - *Isola o tipo da Coleção do resto da aplicação*
  - *Método iterator() (de Collection) retorna Iterator*

```
package java.util;
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```



- *ListIterator possui mais métodos*
  - *Método listIterator() de List retorna ListIterator*

Iterator (exemplo típico)

```
HashMap map = new HashMap();
map.put("um", new Coisa("um"));
map.put("dois", new Coisa("dois"));
map.put("tres", new Coisa("tres"));

(...)

Iterator it = map.values().iterator();
while(it.hasNext()) {
    Coisa c = (Coisa)it.next();
    System.out.println(c);
}
```

Interface Collection

- Principais subinterfaces
  - List
  - Set
- Principais métodos (herdados por todas as subclasses)
  - boolean add(Object o): adiciona objeto na coleção
  - boolean contains(Object o)
  - boolean isEmpty()
  - Iterator iterator(): retorna iterator
  - boolean remove(Object o)
  - int size(): retorna o número de elementos
  - Object[] toArray(Object[]): converte coleção em Array

Interface List

- Principais subclasses
  - ArrayList
  - LinkedList
- Principais métodos adicionais
  - void add(int index, Object o): adiciona objeto na posição indicada (empurra elementos existentes para a frente)
  - Object get(int index): recupera objeto pelo índice
  - int indexOf(Object o): procura objeto e retorna índice da primeira ocorrência
  - Object set(int index, Object o): grava objeto na posição indicada (apaga qualquer outro que ocupava a posição).
  - Object remove(int index)
  - ListIterator listIterator(): retorna um iterator

Implementações: ArrayList e LinkedList

- *ArrayList*
  - *Escolha natural quando for necessário usar um vetor redimensionável: mais eficiente para leitura*
  - *Implementado internamente com vetores*
  - *Ideal para acesso aleatório*
- *LinkedList*
  - *Muito mais eficiente que ArrayList para remoção e inserção no meio da lista*
  - *Ideal para implementar pilhas, filas unidirecionais e bidirecionais. Possui métodos para manipular essas estruturas*
  - *Ideal para acesso seqüencial*

List: exemplo

```
List lista = new ArrayList();
lista.add(new Coisa("um"));
lista.add(new Coisa("dois"));
lista.add(new Coisa("tres"));

(...)

Coisa c3 = lista.get(2); // == índice de vetor
ListIterator it = lista.listIterator();
Coisa c = it.last();
Coisa d = it.previous();
Coisa[] coisas =
    (Coisa[])list.toArray(new Coisa[lista.size()]);
```

Interface Set

- Set representa um conjunto matemático
  - Não possui valores repetidos
- Principais subclasses
  - TreeSet (implements SortedSet)
  - HashSet (implements Set)
- Principais métodos alterados
  - boolean add(Object): só adiciona o objeto se ele já não estiver presente (usa equals() para saber se o objeto é o mesmo)
  - contains(), retainAll(), removeAll(), ...: redefinidos para lidar com restrições de não-duplicação de objetos (esses métodos funcionam como operações sobre conjuntos)

Exemplo

```
Set conjunto = new HashSet();
conjunto.add("Um");
conjunto.add("Dois");
conjunto.add("Tres");
conjunto.add("Um");
conjunto.add("Um");

Iterator it = conjunto.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- *Imprime*
  - *Um*
  - *Dois*
  - *Tres*

Interface Map

- Objetos Map são semelhantes a vetores mas, em vez de índices numéricos, usam chaves, que são objetos
  - Chaves são unívocas (um Set)
  - Valores podem ser duplicados (um Collection)
- Principais subclasses: HashMap e TreeMap
- Métodos
  - void put(Object key, Object value): acrescenta um objeto
  - Object get (Object key): recupera um objeto
  - Set keySet(): retorna um Set de chaves
  - Collection values(): retorna um Collection de valores
  - Set entrySet(): retorna um set de pares chave-valor contendo objetos representados pela classe interna Map.Entry

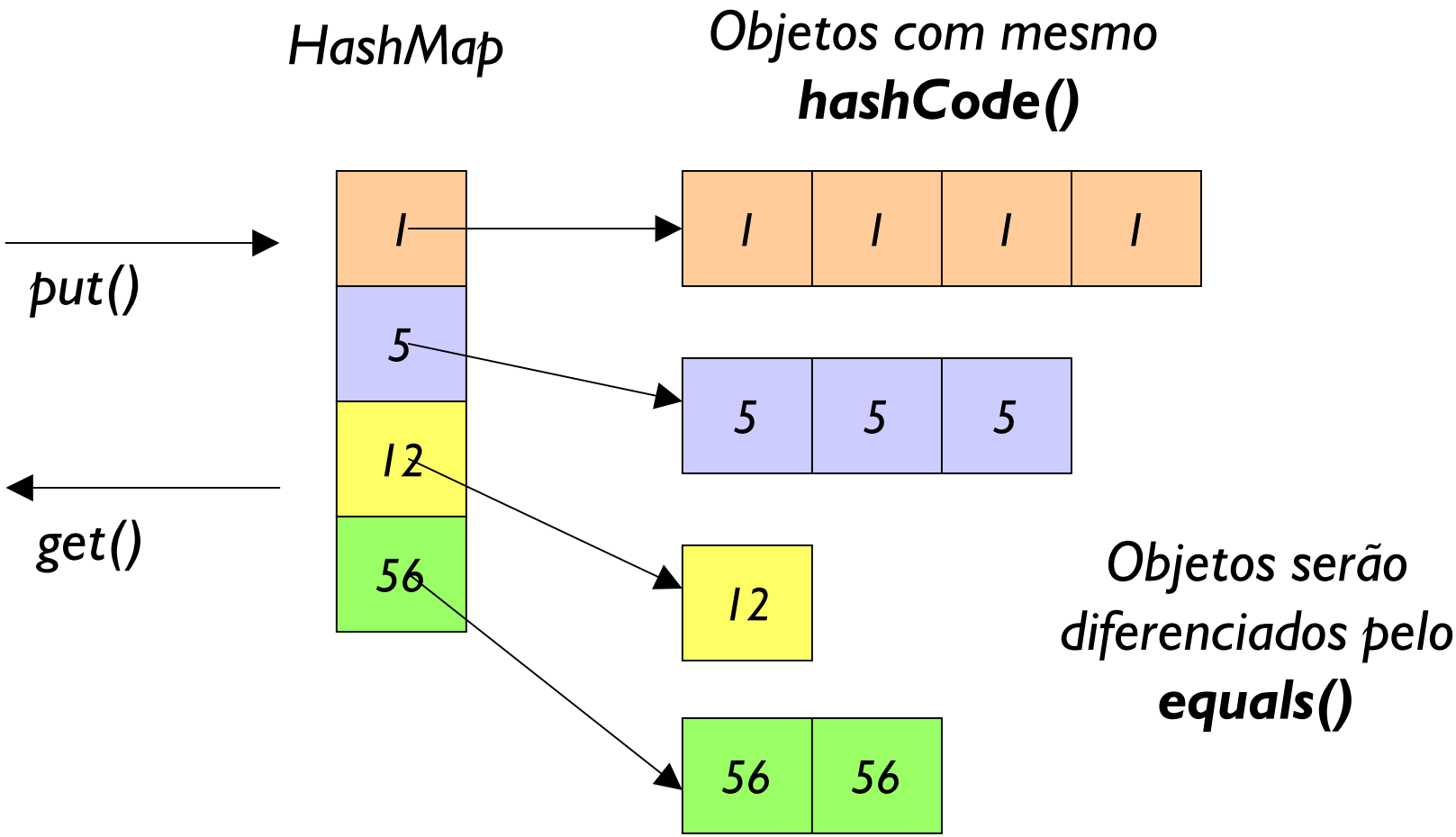
Implementações de Map e Map.Entry

- HashMap
  - Escolha natural quando for necessário um vetor associativo
  - Acesso rápido: usa Object.hashCode() para organizar e localizar objetos
- TreeMap
  - Mapa ordenado
  - Contém métodos para manipular elementos ordenados
- Map.Entry
  - Classe interna usada para manter pares chave-valor em qualquer implementação de Map
  - Principais métodos: Object getKey(), retorna a chave do par; Object getValue(), retorna o valor.

Exemplo

```
Map map = new HashMap();
map.put("um", new Coisa("um"));
map.put("dois", new Coisa("dois"));
(...)
Set chaves = map.keySet();
Collection valores = map.values();
(...)
Coisa c = (Coisa) map.get("dois");
(...)
Set pares = map.entrySet();
Iterator entries = pares.iterator();
Map.Entry one = entries.next();
String chaveOne = (String)one.getKey();
Coisa valueOne = (Coisa)one.getValue();
```

HashMap: como funciona



Coleções legadas de Java 1.0/1.1

- São thread-safe e, portanto, menos eficientes
- Vector, implementa List
  - Use ArrayList, e não Vector, em novos programas
- Stack, subclasse de Vector
  - Implementa métodos de pilha: void push(Object), Object pop() e Object peek().
- Hashtable, implementa Map
  - Use HashMap, e não Hashtable, em novos programas
- Enumeration: tipo de Iterator
  - Retornada pelo método elements() em Vector, Stack, Hashtable e por vários outros métodos de classes mais antigas da API Java
  - boolean hasMoreElements(): equivalente a Iterator.hasNext()
  - Object nextElement(): equivalente a Iterator.next()
  - Use Iterator, e não Enumeration, em novos programas

Propriedades do sistema e Properties

- ***java.util.Properties***: Tipo de *Hashtable* usada para manipular com *propriedades do sistema*
- *Propriedades podem ser*
  1. *Definidas pelo sistema* (user.dir, java.home, file.separator)*
  2. *Passadas na linha de comando java (**-D**prop=valor)*
  3. *Carregadas de arquivo de propriedades contendo pares nome=valor*
  4. *Definidas internamente através da classe Properties*
- *Para ler propriedades passadas em linha de comando (2) ou definidas pelo sistema (1) use System.getProperty():*

```
String javaHome = System.getProperty("java.home");
```

- *Para ler todas as propriedades (sistema e linha de comado)*

```
Properties props = System.getProperties();
```

- *Para adicionar uma nova propriedade à lista*

```
props.setProperty("dir.arquivos", "/imagens");
```

* *Veja Javadoc de System.getProperties() para uma lista.*

Arquivos de propriedades

- *Úteis para guardar valores que serão usados pelos programas*
  - *Pares nome=valor*
  - *Podem ser carregados de um caminho ou do classpath (resource)*
- *Sintaxe*
  - *propriedade=valor ou propriedade: valor*
  - *Uma propriedade por linha (termina com \n ou \r)*
  - *Para continuar na linha seguinte, usa-se "\"*
  - *Caracteres "\", ":" e "=" são reservados e devem ser escapados com "\"*
  - *Comentários: linhas que começam com ! ou # ou vazias são ignoradas*
- *Exemplo*

```
# Propriedades da aplicação
driver=c:\\drivers\\Driver.class
classe.um=pacote.Classe
nome.aplicacao=JCoisas Versão 1.0\:Beta
```

- *Para carregar em propriedades (Properties props)*

```
props.load(new FileInputStream("arquivo.conf"));
```

java.util.Properties

- Métodos de Properties
  - load(InputStream in): carrega um arquivo de Properties para o objeto Properties criado (acrescenta propriedades novas e sobrepõe existentes do mesmo nome)
  - store(OutputStream out, String comentario): grava no OutputStream as propriedades atualmente armazenadas no objeto Properties. O comentário pode ser null.
  - String getProperty(String nome): retorna o valor da propriedade ou null se ela não faz parte do objeto
  - String setProperty(String nome, String valor): grava uma nova propriedade
  - Enumeration propertyNames(): retorna uma lista com os nomes de propriedades armazenados no objeto

A classe java.lang.String

- *É uma seqüência de caracteres imutável*
  - *Representa uma cadeia de caracteres Unicode*
  - *Otimizada para ser lida, mas não alterada*
  - *Nenhum método de String modifica o objeto armazenado*
- *Há duas formas de criar Strings*
  - *Através de construtores, metodos, fontes externas, etc:*

```
String s1 = new String("Texto");
String s2 = objeto.getText(); // método de API
String s3 = coisa.toString();
```

  - *Através de atribuição de um literal*

```
String s3 = "Texto";
```

- *Strings criados através de literais são automaticamente armazenadas em um pool para possível reuso*
  - *Mesmo objeto é reutilizado: comparação de Strings iguais criados através de literais revelam que se tratam do mesmo objeto*

Pool de strings

- *Como Strings são objetos imutáveis, podem ser reusados*
- *Strings iguais criados através de literais são o mesmo objeto*

```
String um = "Um";
String dois = "Um";
if (um == dois)
  System.out.println("um e dois são um!");
```

Todos os blocos de texto abaixo são impressos

- *Mas Strings criados de outras formas não são*

```
String tres = new String("Um");
if (um != tres)
  System.out.println("um nao é três!");
```

- *Literais são automaticamente guardados no pool. Outros Strings podem ser acrescentados no pool usando intern():*

```
quatro = tres.intern();
if (um == quatro)
   System.out.println("quatro é um!");
```

Principais Métodos de String

- Métodos que criam novos Strings
  - String concat(String s): retorna a concatenação do String atual com outro passado como parâmetro
  - String replace(char old, char new): troca todas as ocorrências de um caractere por outro
  - String substring(int start, int end): retorna parte do String incluindo a posição inicial e excluíndo a final
  - String toUpperCase() e String toLowerCase(): retorna o String em caixa alta e caixa baixa respectivamente
- Métodos para pesquisa
  - boolean endsWith(String) e startsWith(String)
  - int indexOf(String), int indexOf(String, int offset): retorna posição
  - char charAt(int posição): retorna caractere em posição
- Outros métodos
  - char[] toCharArray(): retorna o vetor de char correspondente ao String
  - int length(): retorna o comprimento do String

A classe StringBuffer

- *É uma seqüência de caracteres mutável*
  - *Representa uma cadeia de caracteres Unicode*
  - *Otimizada para ser alterada, mas não lida*
- *StringBuffers podem ser criados através de seus construtores*

```
StringBuffer buffer1 = new StringBuffer();
StringBuffer buffer2 = new StringBuffer("Texto inicial");
StringBuffer buffer3 = new StringBuffer(40);
```

- *Métodos de StringBuffer operam sobre o próprio objeto*
  - *StringBuffer append(String s): adiciona texto no final*
  - *StringBuffer insert(int posição, String s): insere na posição*
  - *void setCharAt(int posição, char c): substitui na posição*
  - *String toString(): transforma o buffer em String para que possa ser lido*
- *Exemplo*

```
StringBuffer buffer = new StringBuffer("H");
buffer.append("e").append("l").append("l").append(o");
System.out.println(buffer.toString());
```

Quando usar String e StringBuffer

- Use String para manipular com valores constantes
  - Textos carregados de fora da aplicação
  - Valores literais
  - Textos em geral que não serão modificados intensivamente
- Use StringBuffer para alterar textos
  - Acrescentar, concatenar, inserir, etc.
- Prefira usar StringBuffer para construir Strings
  - Concatenação de strings usando "+" é extremamente cara: um novo objeto é criado em cada fase da compilação apenas para ser descartado em seguida
  - Use StringBuffer.append() e, no final, transforme o resultado em String

java.util.StringTokenizer

- *Classe utilitária que ajuda a dividir texto em tokens*
  - *Recebe um String e uma lista de tokens*
  - *Usa um Enumeration para iterar entre os elementos*
- *Exemplo:*

```
String regStr = "Primeiro,Segundo,Terceiro,Quarto";
StringTokenizer tokens =
        new StringTokenizer(regStr, ",");
String[] registros = null;
List regList = new ArrayList();
while (tokens.hasMoreTokens()) {
        String item = tokens.nextToken();
        regList.add(item);
}
int size = regList.size();
registros = (String[])regList.toArray(new String[size]);
```

Expressões regulares (1.4)

- O pacote java.util.regex contém duas classes que permitem a compilação e uso de expressões regulares (padrões de substituição): Pattern e Matcher
- Exemplo

```
String padrao = "a*b?";
```

- O padrão de pesquisa pode ser compilado

```
Pattern p = Pattern.compile(padrao);
```

e reutilizado

```
Matcher m = p.matcher("aaaaab");
if(m.matches()) { ...}
```

- Ou usado diretamente (sem compilacao)

```
if(Pattern.matches("a*b?", "aaaaa")) { ... }
```

Veja mais sobre expressões regulares nos JavaDocs

Resources

- *Arquivos de propriedades, arquivos de configuração, imagens e outros freqüentemente não estão em local definido*
  - *Aplicações que podem mudar de lugar*
  - *Aplicações empacotadas em um JAR*
- *Usando o ClassLoader, é possível carregar arquivos do disco sem precisar saber sua localização exata, desde que estejam no Class path*
  - *Usa-se um padrão que define parte do caminho até o recurso (deve-se usar o caminho mais completo possível). Ex: conf/config.txt*
  - *Sistema irá localizar o arquivo e retornar uma URL ou stream para que seja possível ler os dados*
- *Exemplo*

```
String recurso = "config/dados.properties";
InputStream stream =
  ClassLoader.getSystemResourceAsStream(recurso);
System.getProperties().load(stream);
```

Reflection

- *Reflection é o nome de uma API que permite descobrir e utilizar informações sobre um objeto em tempo de execução, tendo-se apenas o bytecode e nome de sua classe*
  - *Carregar classes pelo nome dinamicamente via ClassLoader*
  - *Instanciar objetos dessas classes*
  - *Descobrir e chamar todos os seus métodos*
- *Com reflection, é possível escrever programas genéricos, que se adaptam a uma API desconhecida*
- *As classes usadas são java.lang.Class e várias outras no pacote java.lang.reflect*
- *Exemplo*

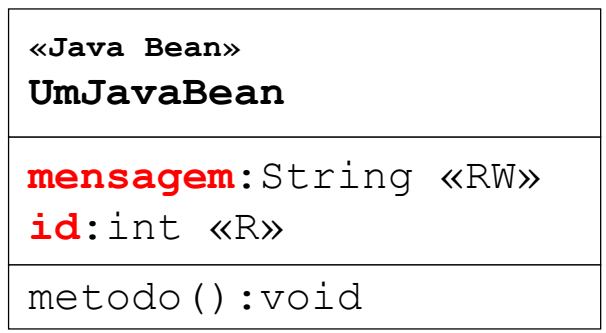
```
Class classe = Class.forName("com.xyz.ClasseNova");
Method metodo = classe.getDeclaredMethod("toString");
Object o = classe.newInstance(); // cria novo objeto
metodo.invoke(o);
```

JavaBeans

- Um JavaBean é um componente reutilizável que tem como finalidade representar um modelo de dados
  - Define convenções para que atributos de dados sejam tratados como "propriedades"
  - Permite manipulação de suas propriedades, ativação de eventos, etc. através de um framework que reconheça as convenções utilizadas
- Basicamente, um JavaBean é uma classe Java qualquer, que tem as seguintes características
  - Construtor público default (sem argumentos)
  - Atributos de dados private
  - Métodos de acesso (acessors) e/ou de alteração (mutators) para cada atributo usando a convenção getPropriedade() (ou opcionalmente isPropriedade() se boolean) e setPropriedade()

Exemplo de um JavaBean

```
public class UmJavaBean {
    private String msg;
    private int id;

    public JavaBean () {}

    public String getMensagem() {
        return mensagem;
    }
    public void setMensagem(String msg) {
        this.msg = msg;
    }
    public String getId() {
        return mensagem;
    }
    public void metodo() {
       ...
    }
}
```



Exercícios

- 4. Implemente uma Type Safe Collection (List) que possa conter Círculos
  - Teste a coleção adicionando, removendo, buscando e imprimindo os objetos da lista
  - Crie um método que retorne a lista como um array.
- 5. Aplicação da Biblioteca: Use os recursos de pesquisa em Strings para e implementar o método findPublicacoes() da aplicação da biblioteca
  - O String passado pode ser uma palavra presente no título da publicação procurada.
- 6. Guarde os assuntos em um arquivo de propriedades (assuntos.properties) no classpath e carregue-os na inicialização da aplicação como propriedades do sistema
  - Use ClassLoader.getSystemResourceAsStream()
  - Copie as propriedades para o HashMap correspondente (assuntos)

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Helder da Rocha
www.argonavis.com.br

Assuntos abordados

- *Este módulo explora os componentes mais importantes do pacote java.io e outros recursos da linguagem relacionados à E/S e arquivos*
- *A classe File, que representa arquivos e diretórios*
- *Objetos que implementam entrada e saída*
  - *InputStream e OutputStream, Readers e Writers*
  - *Compressão com GZIP streams*
  - *FileChannels*
- *Objeto que implementa arquivo de acesso aleatório*
  - *RandomAccessFile*
- *Recursos de serialização básica*
  - *Serializable, ObjectOutputStream e ObjectInputStream*
- *Logging: fundamentos*

O pacote java.io

- Oferece abstrações que permitem ao programador lidar com arquivos, diretórios e seus dados de uma maneira independente de plataforma
  - File, RandomAccessFile
- Oferecem recursos para facilitar a manipulação de dados durante o processo de leitura ou gravação
  - bytes sem tratamento
  - caracteres Unicode
  - dados filtrados de acordo com certo critério
  - dados otimizados em buffers
  - leitura/gravação automática de objetos
- Pacote java.nio (New I/O): a partir do J2SDK 1.4.0
  - Suporta mapeamento de memória e bloqueio de acesso

A classe File

- Usada para representar o sistema de arquivos
  - É apenas uma abstração: a existência de um objeto File não significa a existência de um arquivo ou diretório
  - Contém métodos para testar a existência de arquivos, para definir permissões (nos S.O.s onde for aplicável), para apagar arquivos, criar diretórios, listar o conteúdo de diretórios, etc.
- Alguns métodos
  - String getAbsolutePath()
  - String getParent(): retorna o diretório (objeto File) pai
  - boolean exists()
  - boolean isFile()
  - boolean isDirectory()
  - boolean delete(): tenta apagar o diretório ou arquivo
  - long length(): retorna o tamanho do arquivo em bytes
  - boolean mkdir(): cria um diretório com o nome do arquivo
  - String[] list(): retorna lista de arquivos contido no diretório

File: exemplo de uso

```
File diretorio = new File("c:\tmp\cesto");
diretorio.mkdir(); // cria, se possível
File arquivo = new File(diretorio, "lixo.txt");
FileOutputStream out =
            new FileOutputStream(arquivo);
// se arquivo não existe, tenta criar
out.write( new byte[]{'l','i','x','o'} );

File subdir = new File(diretorio, "subdir");
subdir.mkdir();
String[] arquivos = diretorio.list();
for (int i = 0; arquivos.length; i++) {
   File filho = new File(diretorio, arquivos[i]);
   System.out.println(filho.getAbsolutePath());
}
if (arquivo.exists()) {
   arquivo.delete();
}
```

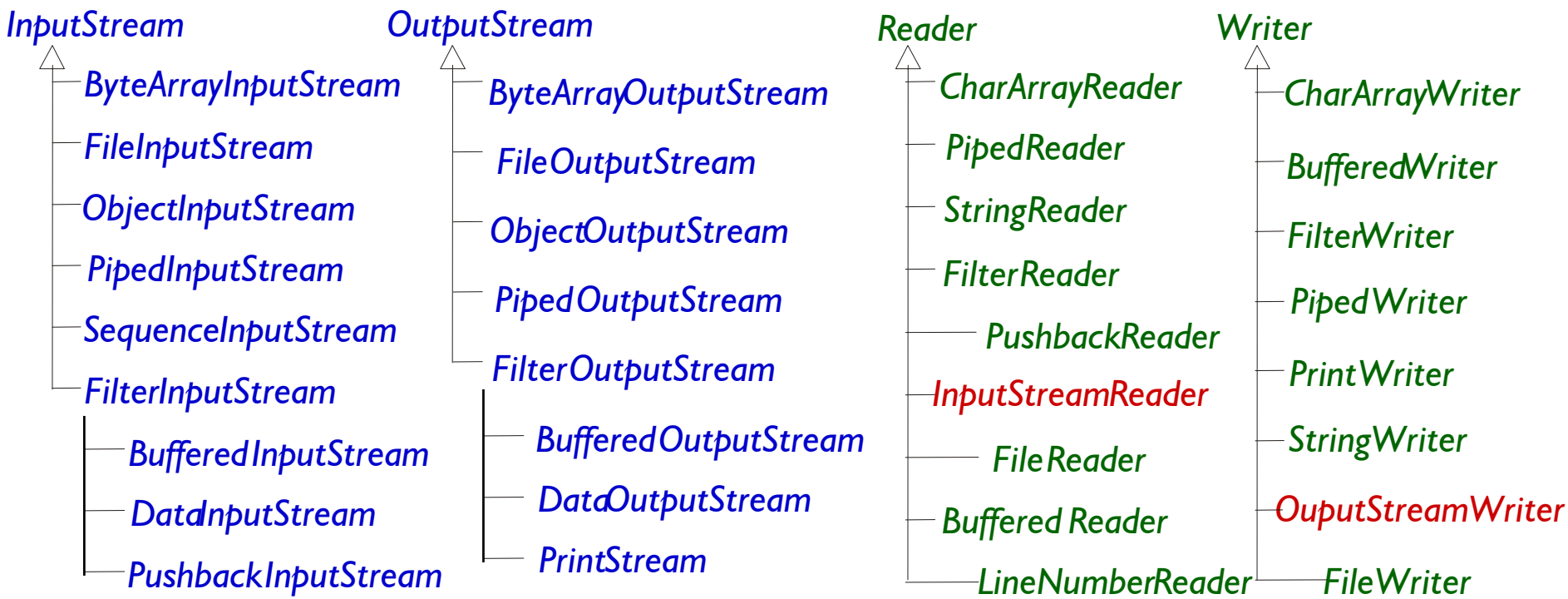
O bloco de código acima precisa tratar IOException

Fluxos de Entrada e Saída

- Há várias fontes de onde se deseja ler ou destinos para onde se deseja gravar ou enviar dados
  - Arquivos
  - Conexões de rede
  - Console (teclado / tela)
  - Memória
- Há várias formas diferentes de ler/escrever dados
  - Seqüencialmente / aleatoriamente
  - Como bytes / como caracteres
  - Linha por linha / palavra por palavra, ...
- APIs Java para I/O oferecem objetos que abstraem fontes/destinos (nós) e fluxos de bytes e caracteres

Classes e interfaces para fluxos de E/S

- Dois grupos:
  - e/s de bytes: InputStream e OutputStream
  - e/s de chars: Reader e Writer



E/S de bytes

- *InputStream*
  - *Classe genérica (abstrata) para lidar com fluxos de bytes de entrada e nós de fonte (dados para leitura).*
  - *Método principal: read()*
- *OutputStream*
  - *Classe genérica (abstrata) para lidar com fluxos de bytes de saída e nós de destino (dados para gravação).*
  - *Método principal: write()*
- *Principais implementações*
  - *Nós (fontes): FileInputStream (arquivo), ByteArrayInputStream (memória) e PipedInputStream (pipe).*
  - *Processamento de entrada: FilterInputStream (abstract) e subclasses*
  - *Nós (destinos): FileOutputStream (arquivo), ByteArrayOutputStream (memória) e PipedOutputStream (pipe).*
  - *Processamento de saída: FilterOutputStream (abstract) e subclasses.*

Métodos de InputStream e OutputStream

- Principais métodos de InputStream
  - int read(): retorna um byte (ineficiente)
  - int read(byte[] buffer): coloca bytes lidos no vetor passado como parâmetro e retorna quantidade lida
  - int read(byte[] buffer, int offset, int length): idem
  - void close(): fecha o stream
  - int available(): número de bytes que há para ler agora
- Métodos de OutputStream
  - void write(int c): grava um byte (ineficiente)
  - void write(byte[] buffer)
  - void write(byte[] buffer, int offset, int length)
  - void close(): fecha o stream (essencial)
  - void flush(): esvazia o buffer

Exemplo de leitura e gravação de arquivo

- *Trecho de programa que copia um arquivo**

```
String nomeFonte = args[0];
String nomeDestino = args[1];
File fonte = new File(nomeFonte);
File destino = new File(nomeDestino);
if (fonte.exists() && !destino.exists()) {
   FileInputStream in = new FileInputStream(fonte);
   FileOutputStream out = new FileOutputStream(destino);
   byte[] buffer = new byte[8192];
   int length = in.read(buffer);
   while ( length != -1) {
      out.write(buffer, 0, length);
      in.read(buffer);
   }
   in.close();
   out.flush();
   out.close();
}
```

-1 sinaliza EOF

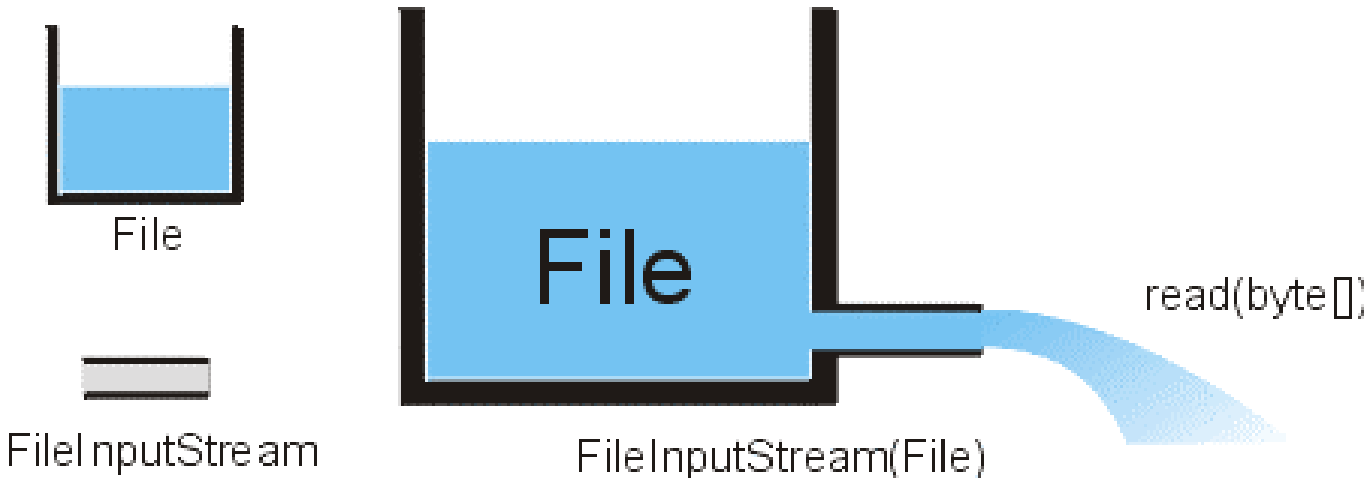
Grava apenas os bytes lidos (e não o buffer inteiro)

* *Método e blocos try-catch (obrigatórios) foram omitidos para maior clareza.*

FilterStreams

- Implementam o padrão de projeto Decorator
  - São concatenados em streams primitivos oferecendo métodos mais úteis com dados filtrados
- FilterInputStream: recebe fonte de bytes e oferece métodos para ler dados filtrados. Implementações:
  - DataInputStream: readInt(), readUTF(), readDouble()
  - BufferredInputStream: read() mais eficiente
  - ObjectOutputStream: readObject() lê objetos serializados
- FilterOutputStream: recebe destino de bytes e escreve dados via filtro. Implementações:
  - DataOutputStream: writeUTF(), writeInt(), etc.
  - BufferedOutputStream: write() mais eficiente
  - ObjectOutputStream: writeObject() serializa objetos
  - PrintStream: classe que implementa println()

Exemplo: leitura de um stream fonte (arquivo)



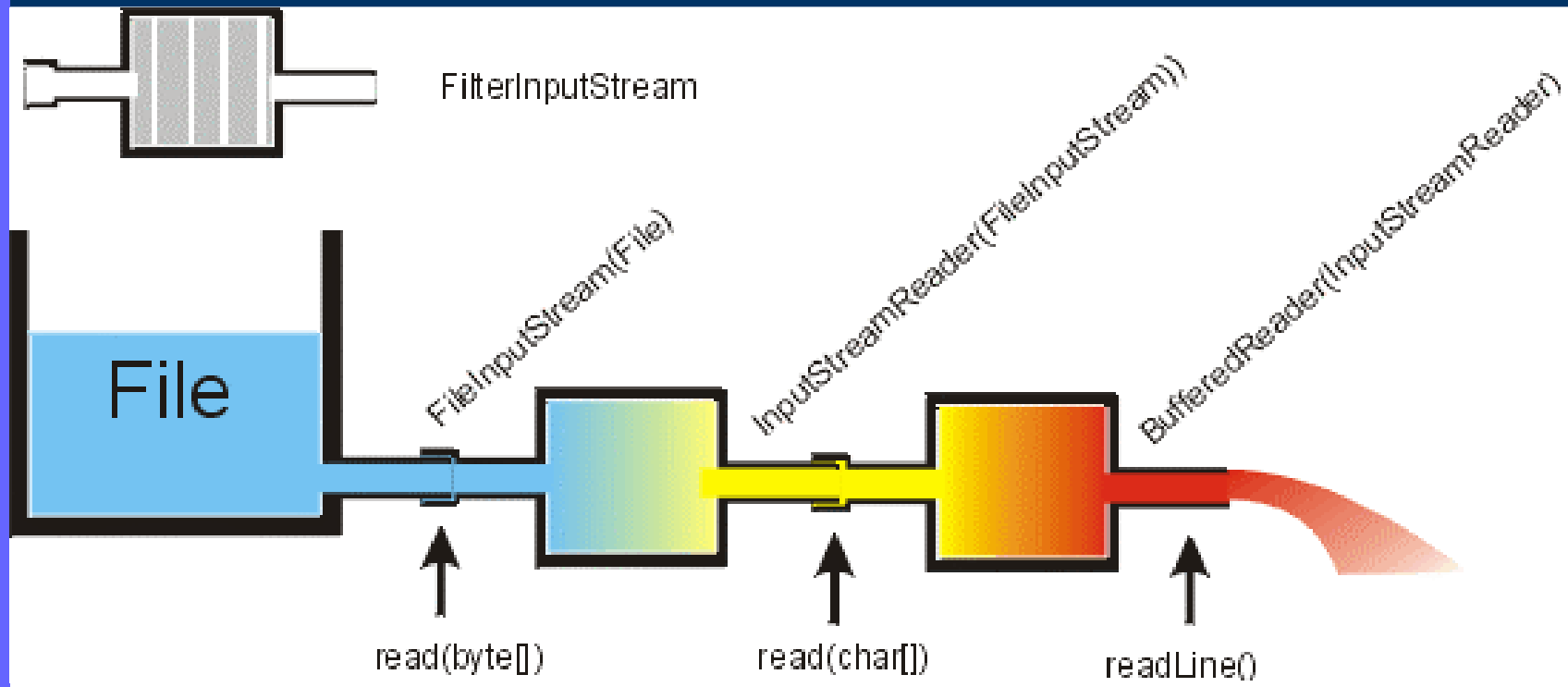

```
// objeto do tipo File
File tanque = new File("agua.txt");

// referência FileInputStream
// cano conectado no tanque
FileInputStream cano =
                  new FileInputStream(tanque);

// lê um byte a partir do cano
byte octeto = cano.read();
```

Usando filtro para ler char

- *InputStreamReader é um filtro que converte bytes em chars*
  - *Para ler chars de um arquivo pode-se usar diretamente um FileWriter em vez de concatenar os filtros abaixo.*

```
// objeto do tipo File
File tanque = new File("agua.txt");

// referência FileInputStream
// cano conectado no tanque
FileInputStream cano =
                  new FileInputStream(tanque);

// filtro chf conectado no cano
InputStreamReader chf =
              new InputStreamReader(cano);

// lê um char a partir do filtro chf
char letra = chf.read();
```

Usando outro filtro para ler linha




```
// filtro chf conectado no cano
InputStreamReader chf = new InputStreamReader(cano);
// filtro br conectado no chf
BufferedReader br = new BufferedReader (chf);
// lê linha de texto a de br
String linha = br.readLine();
```

E/S de caracteres

- Reader
  - Classe abstrata para lidar com fluxos de caracteres de entrada: método read() lê um caractere (16 bits) por vez
- Writer
  - Classe abstrata para lidar com fluxos de bytes de saída: método write() grava um caractere (16 bits) por vez
- Principais implementações
  - Nós (destinos): FileWriter (arquivo), CharArrayWriter (memória), PipedWriter (pipe) e StringWriter (memória).
  - Processamento de saída: FilterWriter (abstract), BufferedWriter, OutputStreamWriter (conversor de bytes para chars), PrintWriter
  - Nós (fontes): FileReader (arquivo), CharArrayReader (memória), PipedReader (pipe) e StringReader (memória).
  - Processamento de entrada: FilterReader (abstract), BufferedReader, InputStreamReader (conversor bytes p/ chars), LineNumberReader

Métodos de Reader e Writer

- Principais métodos de Reader
  - int read(): lê um char (ineficiente)
  - int read(char[] buffer): coloca chars lidos no vetor passado como parâmetro e retorna quantidade lida
  - int read(char[] buffer, int offset, int length): idem
  - void close(): fecha o stream
  - int available(): número de chars que há para ler agora
- Métodos de Writer
  - void write(int c): grava um char (ineficiente)
  - void write(char[] buffer)
  - void write(char[] buffer, int offset, int length)
  - void close(): fecha o stream (essencial)
  - void flush(): esvazia o buffer

Leitura e gravação de texto com buffer

- *A maneira mais eficiente de ler um arquivo de texto é usar FileReader decorado por um BufferedReader. Para gravar, use um PrintWriter decorando o FileWriter*

```
BufferedReader in = new BufferedReader(
    new FileReader("arquivo.txt"));
StringBuffer sb =
    new StringBuffer(new File("arquivo.txt").length());
String linha = in.readLine();
while( linha != null ) {
    sb.append(linha).append('\n');
    linha = in.readLine();
}
in.close();
String textoLido = sb.toString();
// (...)
PrintWriter out = new PrintWriter(
    new FileWriter("ARQUIVO.TXT"));
out.print(textoLido.toUpperCase());
out.close();
```

Leitura da entrada padrão e memória

- A entrada padrão (System.in) é representada por um objeto do tipo InputStream.
- O exemplo abaixo lê uma linha de texto digitado na entrada padrão e grava em uma String. Em seguida lê seqüencialmente a String e imprime uma palavra por linha

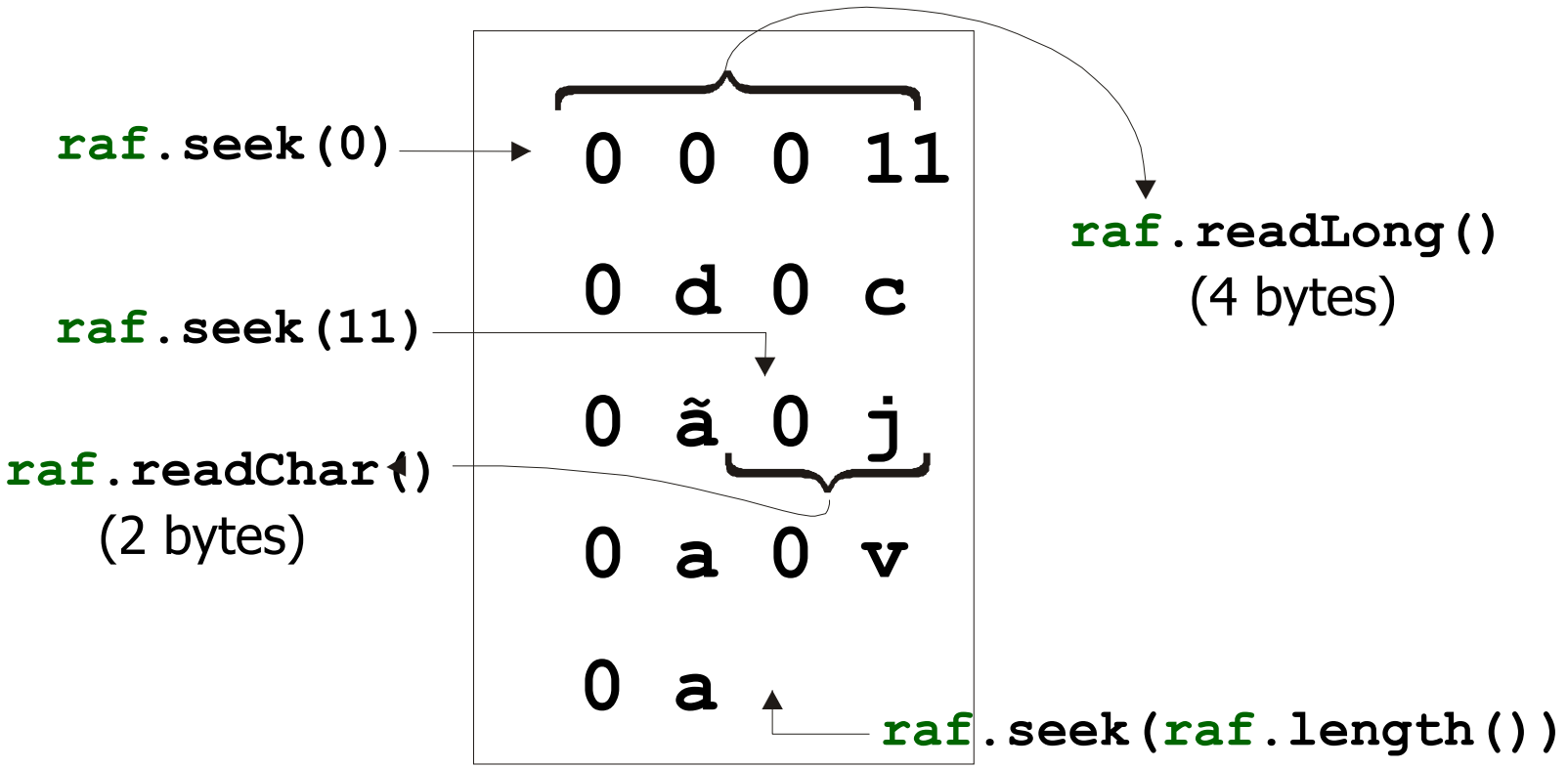
```
BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Digite uma linha:");
String linha = stdin.readLine());

StringReader rawIn = new StringReader(linha);
int c;
while((c = rawIn.read()) != -1)
  if ( c == ' ') System.out.println();
  else System.out.print((char)c);
}
```

RandomAccessFile

- Classe "alienígena": não faz parte da hierarquia de fluxos de dados do java.io
  - Implementa interfaces DataOutput e DataInput
  - Mistura de File com streams: não deve ser usado com outras classes (streams) do java.io
- Oferece acesso aleatório a um arquivo através de um ponteiro
- Métodos (DataOutput e DataInput) tratam da leitura e escrita de Strings e tipos primitivos
  - void seek(long)
  - readInt(), readBytes(), readUTF(), ...
  - writeInt(), writeBytes(), writeUTF(), ...

RandomAccessFile

```
RandomAccessFile raf =
        new RandomAccessFile ("arquivo.dat", "rw");
```



Exceptions

- *A maior parte das operações de E/S provoca exceções que correspondem ou são subclasses de IOException*
  - *EOFException*
  - *FileNotFoundException*
  - *StreamCorruptedException*
- *Para executar operações de E/S é preciso, portanto, ou capturar IOException ou repassar a exceção através de declarações throws nos métodos causadores*
- *Nos exemplos mostrados o tratamento de exceções foi omitido. Tipicamente, as instruções close() ocorrem em um bloco try-catch dentro de um bloco finally*

```
try { ... } finally {
    try { stream.close(); } catch (IOException e) {}
}
```

Não adianta saber se o fechamendo to stream falhou

Serialização

- Java permite a gravação direta de objetos em disco ou seu envio através da rede
  - Para isto, o objeto deve declarar implementar java.io Serializable
- Um objeto Serializable poderá, então
  - Ser gravado em qualquer stream usando o método writeObject() de ObjectOutputStream
  - Ser recuperado de qualquer stream usando o método readObject() de ObjectInputStream
- Um objeto serializado é um grafo que inclui dados da classe e todas as suas dependências
  - Se a classe ou suas dependências mudar, o formato usado na serialização mudará e os novos objetos serão incompatíveis com os antigos (não será mais possível recuperar arquivos gravados com a versão antiga)

Exemplo: gravação e leitura de objetos

```
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream(armario)
);

Arco a = new Arco();
Flecha f = new Flecha();

// grava objeto Arco em armario
out.writeObject(a);

// grava objeto flecha em armario
out.writeObject(f);
```

Gravação de objetos

Leitura de objetos na mesma ordem

```
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream(armario)
);

// recupera os dois objetos
// método retorna Object (requer cast)
Arco primeiro = (Arco)in.readObject();
Flecha segundo = (Flecha)in.readObject();
```

ZIP e JAR

- Os pacotes *java.util.zip* e *java.util.jar* permitem comprimir dados e colecionar arquivos mantendo intactas as estruturas de diretórios
- Vantagens
  - Menor tamanho: maior eficiência de E/S e menor espaço em disco
  - Menos arquivos para transferir pela rede (também maior eficiência de E/S)
- Use classes de ZIP e JAR para coleções de arquivos
  - ZipEntry, ZipFile, ZipInputStream, etc.
- Use streams GZIP para arquivos individuais e para reduzir tamanho de dados enviados pela rede

Exemplo com GZIP streams

- *GZIP usa o mesmo algoritmo usado em ZIP e JAR mas não agrupa coleções de arquivos*
  - *GZIPOutputStream comprime dados na gravação*
  - *GZIPInputStream expande dados durante a leitura*
- *Para usá-los, basta incluí-los na cadeia de streams:*

```
ObjectOutputStream out = new ObjectOutputStream(
                new java.util.zip.GZIPOutputStream(
                   new FileOutputStream(armario) ) );

Objeto gravado = new Objeto();
out.writeObject(gravado);

// (...)

ObjectInputStream in = new ObjectInputStream(
                new java.util.zip.GZIPInputStream(
                   new FileInputStream(armario) ) );

Objeto recuperado = (Objeto)in.readObject();
```

FileChannel

- *Novidade no J2SDK 1.4*
- *Permite ler e gravar arquivos, mapeando memória e bloqueando acesso (afeta performance)*
  - *Mapeamento permite abrir o arquivo como se fosse um vetor, usando a classe java.nio.ByteBuffer. Ideal para ler arquivos consistindo de registros de tamanho fixo.*
  - *É preciso importar java.nio.* e java.nio.channels.*;*
- *Exemplo: ler registro de arquivo de registros fixos*

```
FileInputStream stream = new FileInputStream("a.txt");
FileChannel in = stream.getChannel();
int len = (int) in.size();
ByteBuffer map = in.map(FileChannel.MapMode.READ_ONLY, 0, len);
final int TAM = 80; // tamanho de cada registro: 80 bytes
byte[] registro = new byte[TAM]; //array p/ guardar 1 registro
map.position( 5 * TAM ); // posiciona antes do 5o. registro
map.get( registro ); // preenche array com dados encontrados
```

Logging

- Recurso introduzido no J2SDK 1.4
- Oferece um serviço que gera relatórios durante a execução de uma aplicação. Os relatórios são formados por eventos escolhidos pelo programador e podem ter como destino:
  - A tela (console), um arquivo, uma conexão de rede, etc.
- Os dados também podem ser formatados de diversas formas
  - Texto, XML e filtros
- As mensagens são classificadas de acordo com a severidade, em sete níveis diferentes. O usuário da aplicação pode configurá-la para exibir mais ou menos mensagens de acordo com o nível desejado
- Principais classes
  - java.util.logging.Logger e java.util.logging.Level

Logger

- *Para criar um Logger, é preciso usar seu construtor estático:*

```
Logger logger = Logger.getLogger("com.meupacote");
```

- *Os principais métodos de Logger encapsulam os diferentes níveis de detalhamento (severidade) ou tipos de informação. Métodos log() são genéricos e aceitam qualquer nível*
  - `config(String msg)`
  - `entering(String class, String method)`
  - `exiting(String class, String method)`
  - `fine(String msg)`
  - `finer(String msg)`
  - `finest(String msg)`
  - `info(String msg)`
  - `log(Level level, String msg)`
  - `severe(String msg)`
  - `throwing(String class, String method, Throwable e)`
  - `warning(String msg)`

Exemplo de Logging

- *Exemplo da documentação da Sun [J2SDK14]*

```
package com.wombat;
public class Nose{
    // Obtain a suitable logger.
    private static Logger logger =
        Logger.getLogger("com.wombat.nose");

    public static void main(String argv[]){
      // Log a FINE tracing message
      logger.fine("doing stuff");
      try{
         Wombat.sneeze();
      } catch (Error ex){ // Log the error
        logger.log(Level.WARNING,"trouble sneezing",ex);
      }
      logger.fine("done");
    }
}
```

Níveis de severidade

- As seguintes constantes da classe Level devem ser usadas para indicar o nível das mensagens gravadas
  - Level.OFF (não imprime nada no log)
  - Level.SEVERE (maior valor - imprime mensagens graves)
  - Level.WARNING
  - Level.INFO
  - Level.CONFIG
  - Level.FINE
  - Level.FINER
  - Level.FINEST (menor valor - imprime detalhamento)
  - Level.ALL (imprime tudo no log)
- Quanto maior o valor escolhido pelo usuário, menos mensagens são gravadas.

Exercícios

- *1. Escreva um programa que leia um arquivo de texto para dentro de uma janela de aplicação gráfica (TextArea)*
- *2. Faça um programa que leia um arquivo XML ou HTML e arranque todos os tags. Imprima na saída padrão.*
- *3. Aplicação da Biblioteca: Crie uma classe RepositorioDadosArquivo que implemente RepositorioDados mantenha arquivos armazenados em dois diretorios:*
  - *agentes/*
  - *publicacoes/*

  Cada diretório deverá armazenar um arquivo por registro. O nome do arquivo deve ser o codigo do registro e os dados devem estar guardados um em cada linha.
  - *Pode-se usar BufferedReader.readLine() para lê-los.*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

16 Classes internas

Helder da Rocha
www.argonavis.com.br

Classes internas

- Classes podem ser membros de classes, de objetos ou locais a métodos. Podem até serem criadas sem nome, apenas com corpo no momento em que instanciam um objeto
- Há poucas situações onde classes internas podem ou devem ser usadas. Devido à complexidade do código que as utiliza, deve-se evitar usos não convencionais
- Usos típicos incluem tratamento de eventos em GUIs, criação de threads, manipulação de coleções e soquetes
- Classes internas podem ser classificadas em quatro tipos
  - Classes dentro de instruções (classes anônimas)
  - Classes dentro de métodos (classes locais)
  - Classes dentro de objetos (membros de instância)
  - Classes internas estáticas (membros de classe)

Tipos de classes internas

- São sempre classes dentro de classes. Exemplo:

```
class Externa {
   private class Interna {
      public int campo;
      public void metodoInterno() {...}
   }
   public void metodoExterno() {...}
}
```

- Podem ser private, protected, public ou package-private
  - Exceto as que aparecem dentro de métodos, que são locais
- Podem ser estáticas:
  - E chamadas usando a notação `Externa.Interna`
- Podem ser de instância, e depender da existência de objetos:
  - `Externa e = new Externa();`
    `Externa.Interna ei = e.new Externa.Interna();`
- Podem ser locais (dentro de métodos)
  - E nas suas instruções podem não ter nome (anônimas)

Classes estáticas (internal classes)

- *Declaradas como static*
  - *Idênticas às classes externas, mas não têm campos static*
  - *Classe externa age como um pacote para várias classes internas estáticas:* **`Externa.Coisa`**, **`Externa.InternaUm`**
  - *Compilador gera arquivo* **`Externa$InternaUm.class`**

```
class Externa {
   private static class InternaUm {
      public int campo;
      public void metodoInterno() {...}
   }
   public static class InternaDois
                        extends InternaUm {
      public int campo2;
      public void metodoInterno() {...}
   }
   public static interface Coisa {
      void existe();
   }
   public void metodoExterno() {...}
}
```

Classes de instância (embedded classes)

- São membros do objeto, como métodos e campos de dados
- Requerem que objeto exista antes que possam ser usadas.
  - Externamente use `referencia.new` para criar objetos
- Variáveis de mesmo nome sempre se referem à classe externa
  - Use `NomeDaClasse.this` para acessar campos internos

```
class Externa {
   public int campoUm;
   private class Interna {
      public int campoUm;
      public int campoDois;
      public void metodoInterno() {
         this.campoUm = 10; // Externa.campoUm
         Interna.this.campoUm = 15;
      }
   }
   public static void main(String[] args){
      Interna e = (new Externa()).new Interna();
   }
}
```

Classes dentro de métodos (embedded)

- *Servem para tarefas "descartáveis" já que deixam de existir quando o método acaba*
  - *Têm o escopo de variáveis locais. Objetos criados, porém, podem persistir além do escopo do método, se retornados*
  - *Se usa variáveis locais do método essas variáveis devem ser constantes (declaradas final), pois assim podem persistir após a conclusão do método.*

```
public Multiplicavel calcular(final int a, final int b) {
    class Interna implements Multiplicavel {
        public int produto() {
            return a * b; // usa a e b, que são constantes
        }
    }
    return new Interna();
}
public static void main(String[] args){
    Multiplicavel mul = (new Externa()).calcular(3,4);
    int prod = mul.produto();
}
```

Classes anônimas (dentro de instruções)

- *Classes usadas dentro de métodos freqüentemente servem apenas para criar um objeto uma única vez*
  - *A classe abaixo estende ou implementa SuperClasse, que pode ser uma interface ou classe abstrata (o new, neste caso, indica a criação da classe entre chaves, não da SuperClasse)*

    `Object i = new SuperClasse() { implementação };`
  - *Compilador gera arquivo* **`Externa$1.class`**, **`Externa$2.class`**,

```
public Multiplicavel calcular(final int a, final int b) {
    return new Multiplicavel()  {
        public int produto() {
            return a * b;
        }
    };
}
public static void main(String[] args){
    Multiplicavel mul = (new Externa()).calcular(3,4);
    int prod = mul.produto();
}
```

Compare com parte em preto e vermelho do slide anterior!

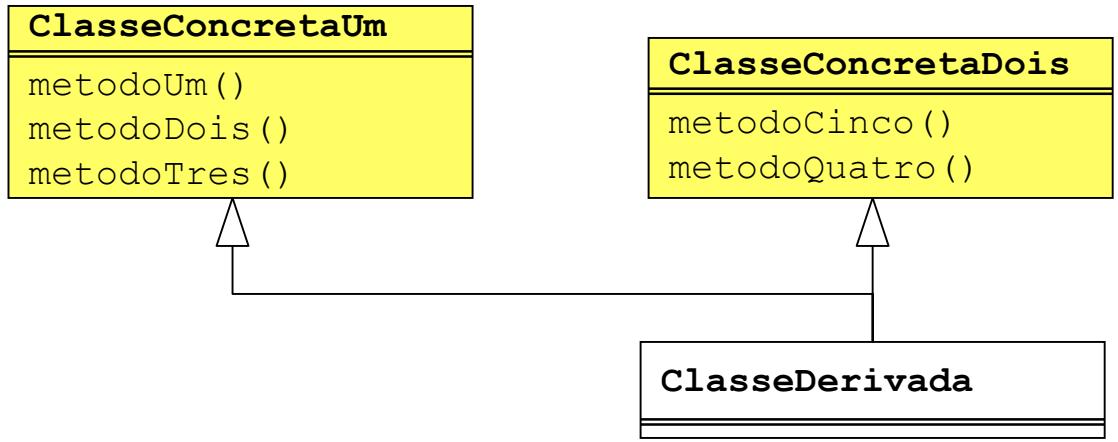
A classe está dentro da instrução: preste atenção no ponto-e-vírgula!

Para que servem classes internas?

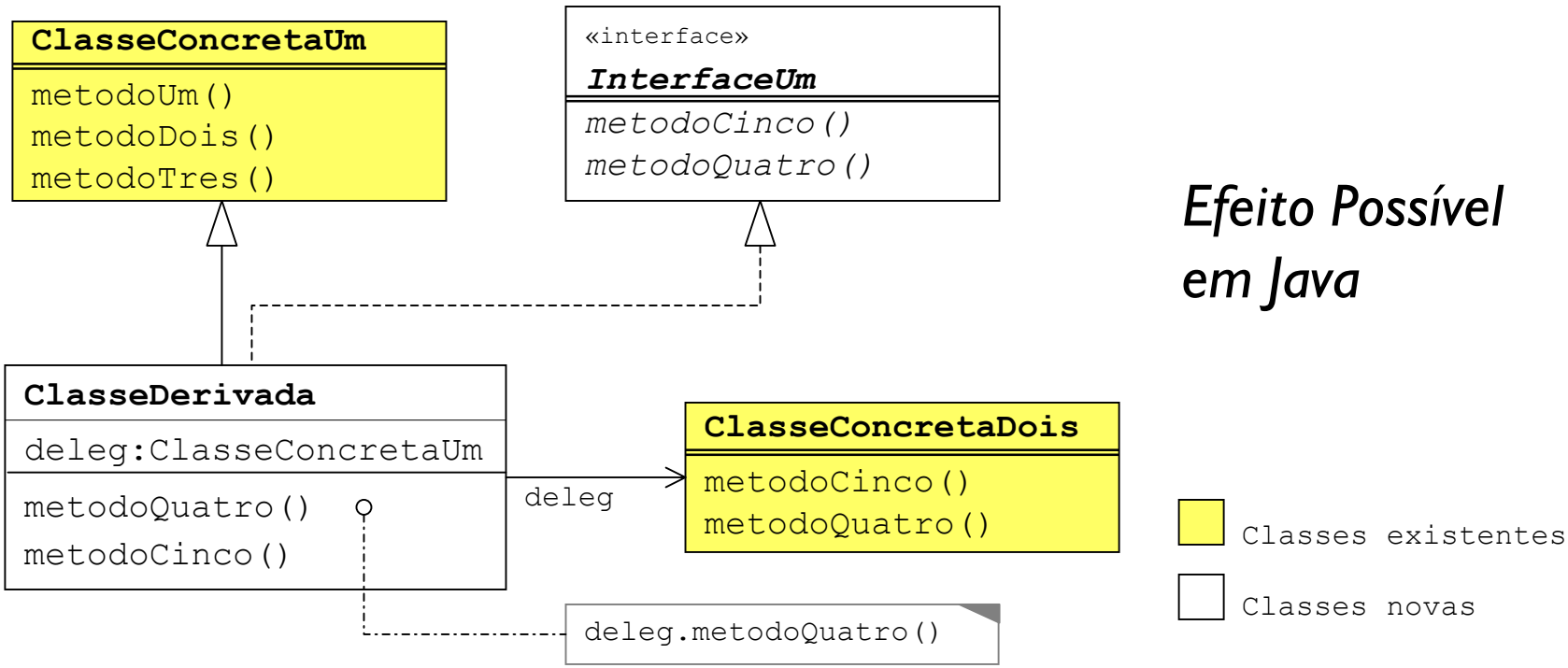
- *Mais reutilização*
  - *Recurso poderoso quando combinado com interfaces e herança - facilita implementação de delegação: tipo de herança de implementação que combinando composição com herança de interfaces (simula herança múltipla)*
  - *"Ponteiros seguros" apontando para métodos localizados em classes internas*
  - *Flexibilidade para desenvolver objetos descartáveis*
- *Riscos*
  - *Aumenta significativamente a complexidade do código*
  - *Dificulta o trabalho de depuração (erros de compilador são mais confusos em classes internas)*
- *Evite fugir do convencional ao usar classes internas*

Como delegação simula herança múltipla

Efeito Desejado
(não permitido em Java)



Efeito Possível em Java



Exercícios

- *1. Escreva uma aplicação que chame o método imprimir() de cada uma das classes do arquivo Internas.java (cap15)*
- *2. Implemente a classe InMethod de Internas.java como uma classe anônima.*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Como construir aplicações gráficas e applets

Helder da Rocha
www.argonavis.com.br

AWT e Swing

- AWT ou Abstract Window Toolkit é o antigo conjunto de ferramentas para interfaces gráficas do Java
  - Serve para oferecer infraestrutura mínima de interface gráfica (nivela por baixo)
  - Componentes têm aparência dependente de plataforma
  - Limitado em recursos devido a depender de suporte de cada plataforma para os componentes oferecidos
  - Bugs e incompatibilidades entre plataformas
- JFC (Java Foundation Classes) oferece uma interface muito mais rica
  - Swing é o nome dado à coleção de componentes
  - É preciso importar java.awt e javax.swing para usar JFC

História do AWT

- *Interface gráfica: componentes, layout, eventos*
- *Java 1.0*
  - *Interface que roda de forma medíocre em todas as plataformas ("Abominable" Window Toolkit)*
  - *Modelo de eventos arcaico*
- *Java 1.1*
  - *Melhora do modelo de eventos: por delegação usando design pattern Observer*
- *Java 1.2*
  - *JFC/Swing substitui totalmente componentes AWT*
  - *Mantém e estende a interface de eventos e layout*

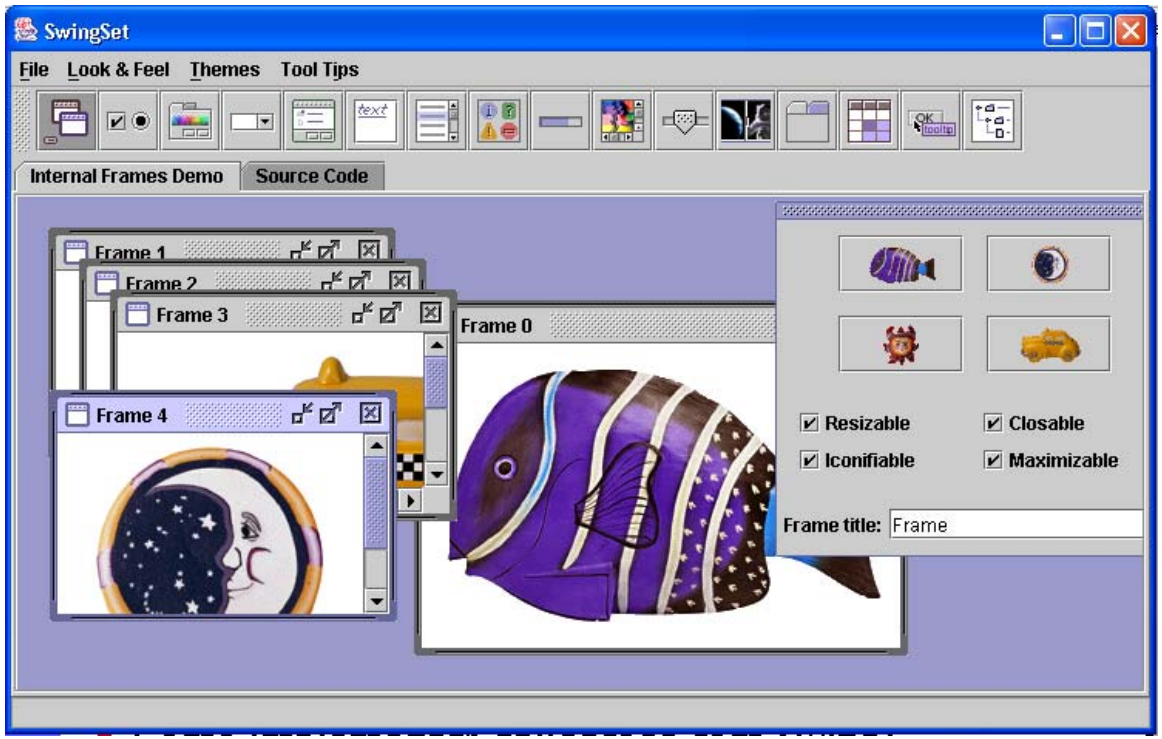
Java Foundation Classes

- Parte do J2SE desde Java SDK 1.2. Consiste de:
- 1. Swing: componentes leves, que não dependem de implementação nativa (veja Java Tutorial)
  - Uma das mais completas bibliotecas gráficas já criadas
  - Baseada em JavaBeans: ferramentas GUI conseguem gerar código legível e reutilizável
- 2. "Look & Feel": Drag & drop, cut & paste, undo/redo, i18n, texto estilizado
- Biblioteca de componentes (apenas o Swing) é compatível com JDK 1.1.4
  - Pode ser baixada separadamente e usada com versões limitadas do Java como J# da Microsoft e MacOS 9

JFC/Swing

- *Veja demo em* `$JAVA_HOME/demo/jfc/SwingSet2/`

```
> java -jar SwingSet2.jar SwingSet2
```



- *Como implementar aplicações com Swing?*
  - *Java Tutorial: Swing "trail" possui guias passo-a-passo para uso de cada componente e recurso do JFC e Swing (www.java.sun.com/tutorial)*

Tipos de aplicações

- Há dois tipos de aplicações gráficas em Java
  - Componentes iniciados via browser (applets)
  - Aplicações standalone iniciadas via sistema operacional
- Ambas capturam eventos do sistema e desenham-se sobre um contexto gráfico fornecido pelo sistema
- Applets são aplicações especiais que rodam a partir de um browser
  - São componentes que executam em um container (ambiente operacional) fornecido pelo browser
  - Browser é quem controla seu ciclo de vida (início, fim, etc.)
  - Geralmente ocupam parte da janela do browser mas podem abrir janelas extras
  - Possuem restrições de segurança

java.awt.Component

- Raiz da hierarquia de componentes gráficos
  - Componentes Swing herdam de javax.swing.JComponent, que é "neto" de Component
- Há um Component por trás de tudo que pode ser pintado na tela
- Principais métodos (chamados pelo sistema):
  - void paint (java.awt.Graphics g)
  - void repaint()
  - void update(java.awt.Graphics g)
- O objeto passado como argumento durante a execução (contexto gráfico) é, na verdade, um java.awt.Graphics2D (subclasse de Graphics)

Componentes AWT

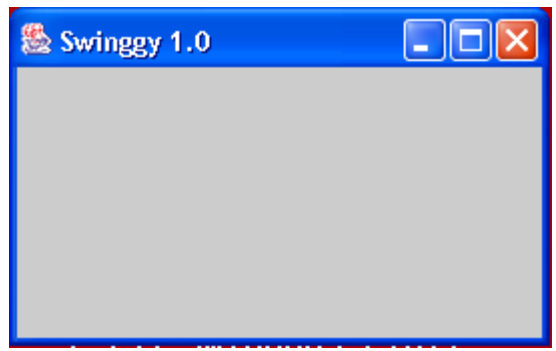
- *Há dois tipos importantes de componentes:*
- *1) descendentes diretos de java.awt.Component*
  - *"Apenas" componentes: descendentes da classe Component que não são descendentes de Container (todos os componentes da AWT)*
- *2) descendentes de java.awt.Container*
  - *Subclasse de java.awt.Component*
  - *São "recipientes." Podem conter outros componentes.*
  - *São descendentes da classe Container: Frame, Panel, Applet e JComponent (raiz da hierarquia dos componentes Swing)*

Containers essenciais

- Frame (AWT) e JFrame (Swing)
  - Servem de base para qualquer aplicação gráfica
- Panel e JPanel
  - Container de propósito geral
  - Serve para agrupar outros componentes e permitir layout em camadas
- Applet e JApplet
  - Tipo de Panel (JPanel) que serve de base para aplicações que rodam dentro de browsers
  - Pode ser inserido dentro de uma página HTML e ocupar o contexto gráfico do browser

Exemplo de JFrame

```
import java.awt.*;
import javax.swing.*;

public class Swinggy extends JFrame {

    public Swinggy(String nome) {
        super(nome);

        this.setSize(400,350);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new Swinggy("Swinggy 1.0");
    }

}
```



AWT thread

- Thread que é responsável pela atualização do contexto gráfico
  - Chama update() (método de Component) e passa referência para o contexto gráfico como argumento sempre que for necessário redesenhá-lo.
- Método update(Graphics g)
  1. Limpa a área a ser redesenhada (contexto gráfico)
  2. Chama paint(g)
- Métodos update() e paint() nunca devem ser chamados diretamente a partir do thread principal
  - Use repaint(), que faz o agendamento de uma chamada a update() através do AWT thread
  - Sobreponha update() se desejar

java.awt.Graphics

- Representa o contexto gráfico de cada componente
- Passado pelo sistema quando chama update()
- Programador pode desenhar no componente usando referência recebida via paint() ao sobrepor o método:

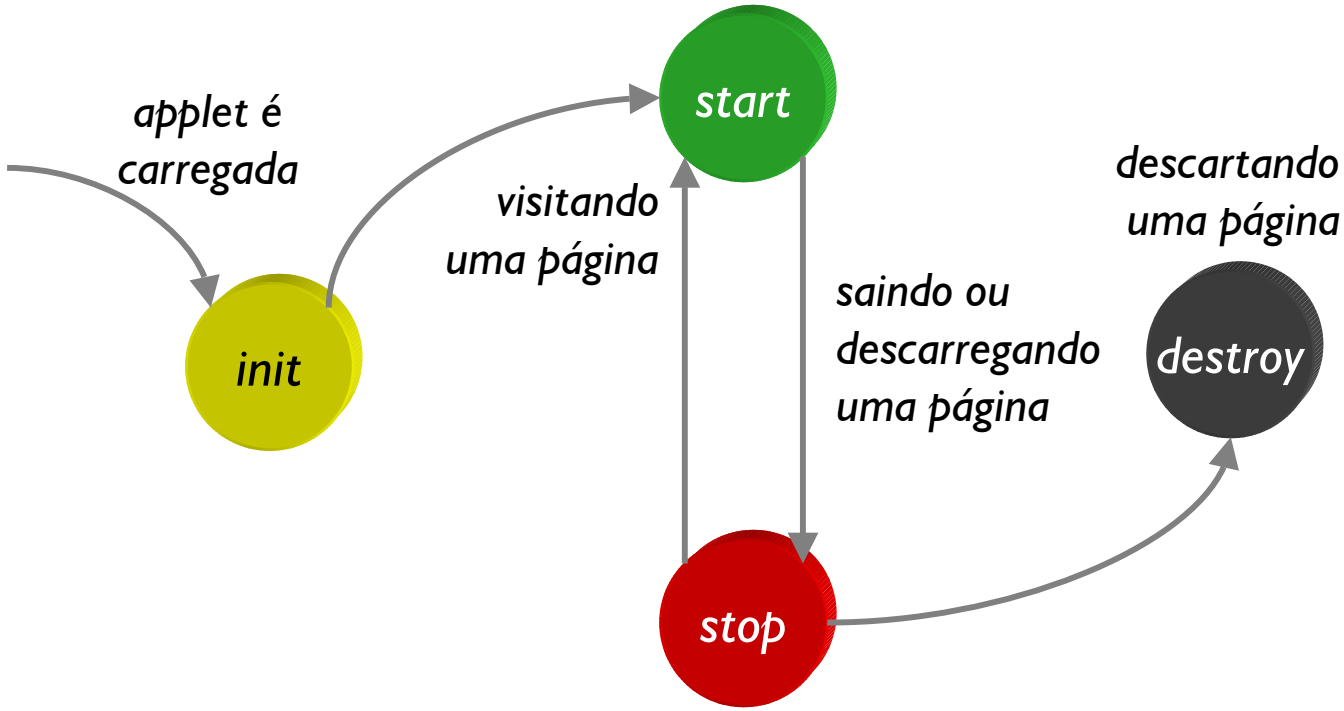
```
public void paint(Graphics g) {
      Graphics2D g2 = (Graphics2D) g;
      Shape s = new Ellipse2D.Double();
      g2.setColor(Color.red);
      g2.draw(s);
}
```

- Para definir o que será desenhado em determinado componente, sobreponha seu método paint()
  - Use Graphics2D! Mais recursos!

Applets

- Aplicação gráfica que roda em browser
  - Toda a infraestrutura herdada da classe javax.swing.JApplet (ou java.applet.Applet)
  - É um componente de um framework que executa em ambiente de execução (container) no browser
- Métodos de JApplet, chamados automaticamente, devem ser sobrepostos. Ciclo de vida:
  - init() - inicialização dos componentes do applet
  - start() - o que fazer quando applet iniciar
  - stop() - o que fazer antes de applet parar
  - destroy() - o que fazer quando applet terminar
  - paint() - o que desenhar no contexto gráfico

Ciclo de vida



- *paint()* é outro método que é chamado automaticamente, mas não faz parte do ciclo de vida do Applet, especificamente
  - Faz parte do ciclo de vida de qualquer aplicação gráfica
  - No Applet, é chamado depois do start() e sempre que o contexto gráfico do applet for precisar ser atualizado (redimensionamento da janela do browser, ocultação do applet por outra janela, etc.)

Como construir applets

- *Applet mínimo*

```
import javax.swing.*;
import java.awt.*;


/*
 * <applet code="HelloApplet" height="50"
 *         width="200"></applet>
 */
public class HelloApplet extends JApplet {
   public void init() {
      Container pane = this.getContentPane();
      JLabel msg = new JLabel("Hello Web");
      pane.add(msg);
   }
}
```

Comentário usado pelo appletviewer para exibir Applet:
> appletviewer HelloApplet.java

Como usar applets

- *Até Java 1.1: para incluir um applet na página Web usava-se*

```
<applet code="pacote.Classe"
        height="100" width="100">
    <param name="parametro1" value="valor">
    <param name="parametro2" value="valor">
</applet>
```

- *Java 2 (J2SDK 1.2 em diante) usa HTML 4.0 (que tornou o tag <applet> obsoleto)*

```
<object classid="XXX-XXX" ...> ... </object>
```

- *Para gerar <object> a partir de <applet> use o HTML Converter, distribuído com o SDK:*

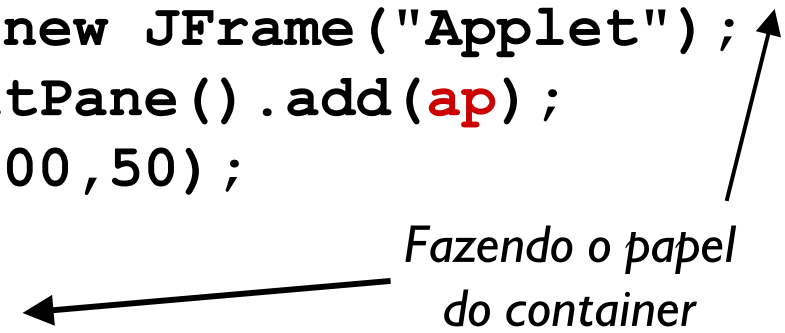
```
java -jar $JAVA_HOME/lib/htmlconverter.jar -gui
```

Applet/Aplicação

- *JApplets podem ser incluídos em JFrames e Applets podem ser incluídos em Frames*
- *Para criar um programa que roda tanto como applet como aplicação*
  1. *Escreva o applet da forma convencional, implementando init(), start(), etc. (métodos do framework, que o container usa para controlar o ciclo de vida do applet)*
  2. *Crie um método main, e nele*
     - *Crie um novo JFrame e uma instância do applet*
     - *Adicione o applet no novo JFrame*
     - *Torne o JFrame visível*
     - *Chame init() e start() do applet*

Exemplo

```
public class HelloApplet extends JApplet {
    public void init() {
        (...)
    }

    public static void main(String[] args) {
        HelloApplet ap = new HelloApplet();
        JFrame f = new JFrame("Applet");
        f.getContentPane().add(ap);
        f.setSize(200,50);
        ap.init();
        ap.start();
        f.setVisible(true);
    }

}
```



Restrições dos applets

- Há várias coisas que aplicações comuns podem e que um applet não pode fazer:
  - Não pode carregar bibliotecas ou definir métodos nativos
  - Não pode ler ou escrever arquivos na máquina cliente
  - Não pode fazer conexões de rede a não ser para a máquina de onde veio
  - Não pode iniciar a execução de nenhum programa na máquina do cliente
  - Não tem acesso à maior parte das propriedades do sistema
  - Janelas abertas sempre têm aviso de segurança
- Várias restrições podem ser flexibilizadas se o applet for assinado.

Applets: vantagens / desvantagens

- Desvantagens
  - Restrições
  - Dependência de plug-in e incompatibilidade em browsers
  - Tempo de download
- Vantagens
  - Facilidade para realizar comunicação em rede
  - Possibilidade de abrir janelas externas
  - Capacidade de estender o browser em recursos de segurança, protocolos de rede, capacidade gráfica
  - Aplicação sempre atualizada
  - Capacidade de interagir com a página via JavaScript
- Alternativa (1.4)
  - Java Web Start: aplicações "normais" instaladas via rede

Java Web Start

- Aplicação distribuída (cliente e servidor) que viabiliza a distribuição e instalação de aplicações via rede
  - Aplicação instalada via Web Start é uma aplicação Java "normal" com possibilidade de ter acesso irrestrito ao sistema (usuário deve autorizar esse acesso na instalação)
  - Checa, sempre que é inicializada, se houve atualização caso a rede esteja disponível
- Use o cliente Java Web Start para localizar aplicações remotas e instalá-las na sua máquina
  - Em máquinas Windows, cliente é instalado junto com J2SDK
- Configure seu servidor Web para suportar JNLP (Java Network Launch Protocol) e distribuir aplicações via Java Web Start
  - Configuração básica consiste da criação de alguns arquivos XML e definição de novo MIME type no servidor
  - Veja links para info sobre Java Web Start na documentação do J2SDK

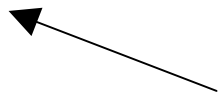
Recursos gráficos básicos: Fontes e Cores

- *Qualquer componente pode mudar a sua fonte e cor*
  - *A mudança afeta todos os componentes contidos no componente afetado*
- *Cores*
  - *instância da classe java.awt.Color*

```
componente.setBackground(new Color(255,0,0));
componente.setForeGround(Color.yellow);
```

- *Fontes*
  - *instância da classe java.awt.Font*

```
Font f = new Font("SansSerif", Font.BOLD, 24);
componente.setFont(f);
```



Há maneiras mais sofisticadas de lidar com fontes (esta é compatível com AWT)

Posicionamento de componentes: Layouts

- *Há duas formas de acrescentar componentes em um container*
  - *Usar um algoritmo de posicionamento (layout manager) para dimensionar e posicionar os componentes (esta é a maneira recomendada e default)*
  - *Desligar o algoritmo de layout e posicionar e redimensionar os componentes diretamente (pixels)*
- *Todo container tem um algoritmo de layout default*
  - *Frame e JFrame: BorderLayout (layout "geográfico")*
  - *Outros Containers: FlowLayout (layout seqüencial)*

JFrame

- Para acrescentar objetos em um JFrame ou JApplet, é preciso obter uma interface opaca chamada ContentPane
  - O ContentPane é uma área independente de plataforma que cobre a área útil do JFrame
  - O layout é definido no ContentPane
  - Objetos são adicionados no ContentPane
  - Cores e fontes devem ser definidas a partir do ContentPane
- Para obter o ContentPane use

```
Container pane = frame.getContentPane();
```

- Para definir um layout (diferente de BorderLayout)

```
pane.setLayout(referência_para_layout);
```

FlowLayout, JButton, Icon

- *FlowLayout*
  - *É o layout mais simples*
  - *Dispõe os objetos um depois do outro como se fossem letras digitadas em um editor de texto*
  - *Estilo default é centralizado (pode ser alterado)*

```
pane.setLayout(new FlowLayout());
```

- *JButton*
  - *Botões simples*
  - *Aceitam textos ou imagens (através da interface Icon)*

```
JButton b1 = new Button("texto");
JButton b2 = new Button("texto", icone);
```

- *Ícones de imagem*

```
Icon icone = new ImageIcon("caminho");
```

  - *Caminho deve ser relativo (de preferência) e usar sempre "/" como separador*

Componentes de texto

- *JTextField*
  - *campo de entrada de dados simples*
- *JPasswordField*
  - *campo para entrada de dados ocultos*
- *JTextArea*
  - *campo de entrada de texto multilinha*
- *JEditorPane*
  - *editor que entende HTML e RTF*
- *JTextPane*
  - *editor sofisticado com vários recursos*

Principais métodos

- *getText()*: *recupera o texto contido no componente*
- *setText(valor)*: *substitui o texto com outro*

- *Veja documentação para mais detalhes*

Exemplo

```
public class Swinggy2 extends JFrame {

  public Swinggy2(String nome) {
     super(nome);

    Container ct = this.getContentPane();
    ct.setLayout(new FlowLayout());

    Icon icone = new ImageIcon("jet.gif");
    JButton b1 = new JButton("Sair");
    JButton b2 = new JButton("Viajar", icone);

    ct.add(b1);
    ct.add(b2);

    this.setSize(400,350);
    this.setVisible(true);
  }
}
```

Layout **null**

- *Algoritmos de layout podem ser combinados para obter qualquer configuração*
  - *Mais fáceis de manter e reutilizar*
  - *Layout em camadas e "orientado a objetos"*
  - *Controlam posicionamento e dimensão de componentes*
- *Algoritmos de layout podem ser criados implementando interface LayoutManager (e LayoutManager2)*
- *Para desligar layouts*

```
pane.setLayout(null);
```

- *Agora é preciso definir posição e tamanho de cada componente*

```
componente.setBounds(x, y, larg, alt);
```

Exemplo

```
private JButton b1, b2, b3;
public Swinggy3(String nome) {
  Container ct = this.getContentPane();
  ct.setLayout(null);

  Icon pataverm = new ImageIcon("redpaw.gif");
  Icon pataverd = new ImageIcon("greenpaw.gif");
  Icon pataazul = new ImageIcon("bluepaw.gif");
  b1 = new JButton("Vermelha", pataverm);
  b2 = new JButton("Verde", pataverd);
  b3 = new JButton("Azul", pataazul);
  b1.setBounds(10,10,150,40);
  b2.setBounds(10,60,150,40);
  b3.setBounds(10,110,150,40);
  ct.add(b1);
  ct.add(b2);
  ct.add(b3);
}
```

Aplicação fica mais simples com vetores

```
private JButton[] b;
private String[] txt = {"Roda", "Para", "Pausa", "Sai",
                        "Olha", "Urgh!"};
private String[] img = { "greenpaw.gif", "redpaw.gif",
                         "bluepaw.gif",  "jet.gif",
                         "eye.gif",      "barata.gif"};
public Swinggy4(String nome) {
   Container ct = this.getContentPane();
   ct.setLayout(null);
   b = new JButton[txt.length];
   for (int i = 0; i < b.length; i++) {
     b[i] = new JButton(txt[i],
              new ImageIcon(img[i]));
     b[i].setBounds(10,10+(i*50),150,40);
     ct.add(b[i]);
   } // (...)
}
```

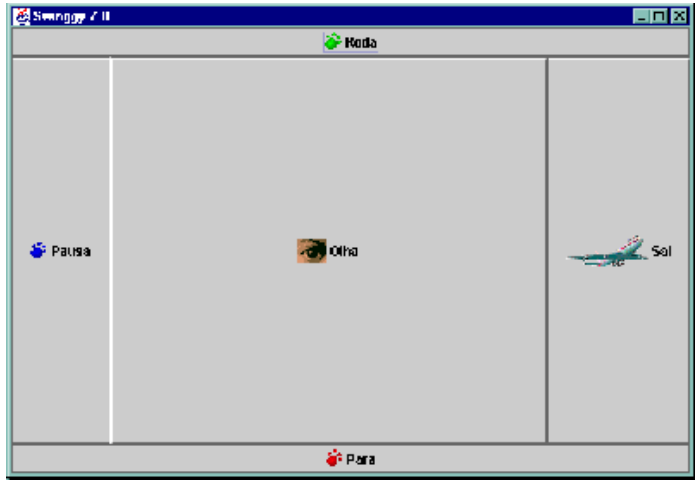
À medida em que a complexidade desta solução aumentar, você poderá querer encapsular a lógica em um LayoutManager!

Outros Layout Managers

- *GridLayout (linhas, colunas)*
  - *Layout que posiciona os elementos como elementos de uma tabela*
- *BorderLayout*
  - *Layout que posiciona elementos em quatro posições "cardeais" e no centro*
  - *Norte e Sul têm prioridade sobre Leste e Oeste que têm prioridade sobre Centro*
  - *Constantes BorderLayout.CENTER, BorderLayout.WEST, BorderLayout.NORTH, etc.*
- *BoxLayout* e *GridBagLayout*
  - *Permitem layouts sofisticados com amplo controle*

Exemplo com BorderLayout

```
(...)

ct.setLayout(new BorderLayout());
b = new JButton[txt.length];
String[] pos = {BorderLayout.NORTH,
                BorderLayout.SOUTH,
                BorderLayout.WEST,
                BorderLayout.EAST,
                BorderLayout.CENTER};
for (int i=0; i < Math.min(b.length, pos.length); i++) {
   b[i] = new JButton(txt[i], new ImageIcon(img[i]));
        ct.add(pos[i], b[i]);
}

(...)
```



Veja restante do código (vetores b, txt e img) em slides anteriores

Preferred Size dos componentes

- Layout Managers são "tiranos"
  - É impossível controlar tamanho e posição de componentes se um LayoutManager estiver sob controle
  - Para flexibilizar regras de posicionamento, configure o Layout Manager usado através de seus construtores e métodos
  - Para flexibilizar regras de dimensionamento, altere o preferred size dos seus componentes
- Tamanhos preferidos dos componentes
  - setPreferredSize(), disponível em alguns componentes, permite definir o seu tamanho ideal
  - getPreferredSize(), disponível em todos os componentes pode ser sobreposto em subclasses e será chamado pelos Layout Managers que o respeitam (Flow, Border)

Regras de BorderLayout

- As áreas de Border Layout só aceitam um componente
  - Se for necessário ter mais de um componente no NORTH, por exemplo, é preciso primeiro adicioná-los dentro de um único componente que será adicionado (um Panel, por exemplo)
- Regras de ocupação de espaço
  - NORTH E SOUTH têm prioridade sobre a ocupação da largura (usam todo o espaço disponível) mas têm altura limitada pelo preferred size do componente
  - EAST e WEST tem altura limitada apenas pela existência ou não de componentes no NORTH e/ou SOUTH e tem largura limitada pelo preferred size do componente
  - CENTER ignora preferred size e ocupa todo o espaço que puder, mas é limitado pela existência de NORTH, SOUTH, EAST ou WEST
- Construtores de BorderLayout permitem controle detalhado de espaçamento e outros detalhes

Exemplo com GridLayout

```
(...)
ct.setLayout(new GridLayout(3, 2));

b = new JButton[txt.length];
for (int i = 0; i < b.length; i++) {
   b[i] = new JButton(txt[i],
                      new ImageIcon(img[i]));
       ct.add(b[i]);
}
(...)
```

Redimensione a janela e veja o resultado

Regras de GridLayout

- Cada célula aceita um componente
  - Se houver mais células que componentes, células não preenchidas ficarão em branco
  - Se houver mais componentes que células, estes não serão mostrados
- Regras de ocupação de espaço
  - Qualquer componente adicionado ocupa toda a célula
  - GridLayout ignora preferred size dos componentes: aumento do frame estica todos os componentes
  - Para manter o preferred size de um componente, pode-se adicioná-lo em um componente que o respeita (por exemplo, um que use FlowLayout) e adicionar este componente em GridLayout

Mesmo exemplo com FlowLayout

```
(...)
public Swinggy5(String nome) {
   super(nome);
   Container ct = this.getContentPane();
   ct.setLayout(new FlowLayout());
   b = new JButton[txt.length];
   for (int i = 0; i < b.length; i++) {
      b[i] = new JButton(txt[i],
                         new ImageIcon(img[i]));
      ct.add(b[i]);
   }
   this.setSize(400,350);
   this.setVisible(true);
} (...)
```

Redimensione a janela e veja o resultado

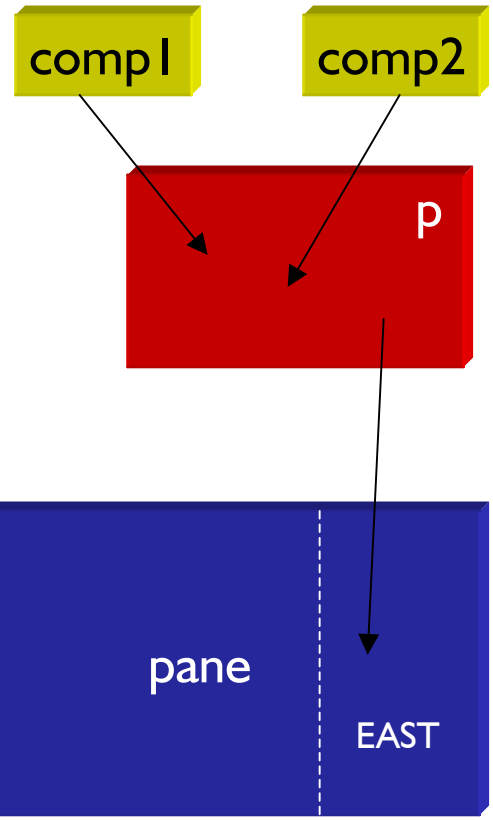
Regras de FlowLayout

- Elementos são adicionados em seqüência
  - Comportamento default é posicionar elementos lado a lado até que a linha em que estão colocados não mais couber nenhum
  - Elementos são centralizados por default
- Construtores permitem controle sobre espaçamento e alinhamento dos componentes
  - Espaço entre componentes é alterado para todos os componentes
  - Alinhamento pode ser pela direita ou pela esquerda, além de default (pelo centro)
- Preferred size é respeitado para todos os componentes

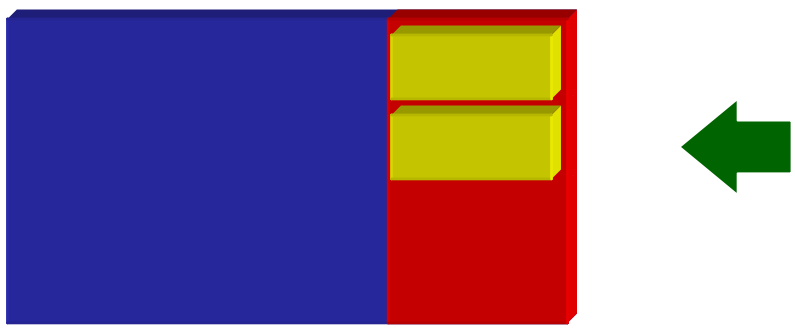
Combinação de Layouts

- *Componentes podem ser combinados em recipientes (como JPanel) para serem tratados como um conjunto*

```
JPanel p = new JPanel();
p.setLayout(layout do JPanel);
p.add(comp1);
p.add(comp2);
pane.add(BorderLayout.EAST, p);
```



- *Possibilita a criação de layouts complexos que consistem de várias camadas*
  - *Cada JPanel é uma camada*



Exemplo (1/2)

```
import javax.swing.*;
import java.awt.*;

public class Swinggy8 extends JFrame {

  public Swinggy8(String nome) {
    super(nome);

    Container framePane = this.getContentPane();
    framePane.setLayout(new BorderLayout());

    JPanel botoes = new JPanel();
    botoes.setBackground(Color.yellow);
    botoes.setLayout(new GridLayout(3,1));
    botoes.add(new JButton("Um"));
    botoes.add(new JButton("Dois"));
    botoes.add(new JButton("Três"));
    JPanel lateral = new JPanel();
    lateral.add(botoes);
```

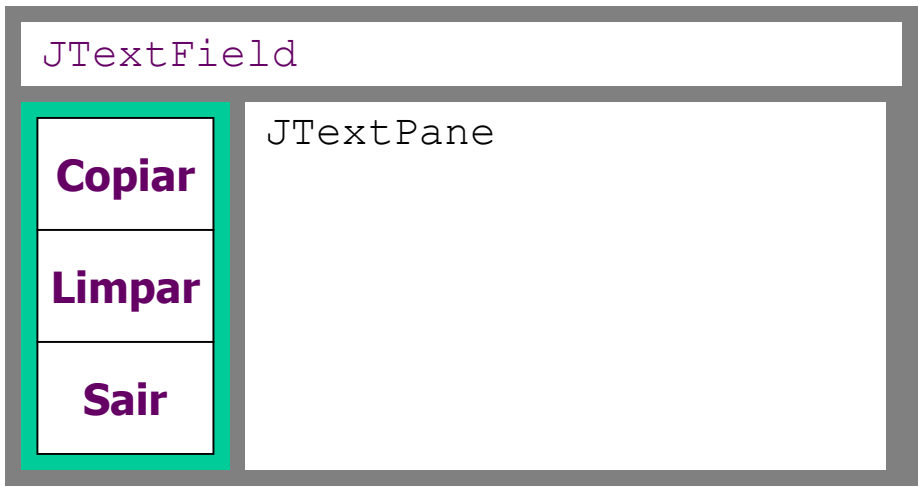
Exemplo (2/2)

```
    JInternalFrame if1 =
     new JInternalFrame("Um", true, true, true);
    JInternalFrame if2 =
     new JInternalFrame("Dois", true, true, true);
    if1.getContentPane().add(new JEditorPane());
    if2.getContentPane().add(new JEditorPane());
    if1.setBounds(20,20, 250,200);
    if2.setBounds(70,70, 250,200);
    if1.setVisible(true);
    if2.setVisible(true);

    JDesktopPane dtp = new JDesktopPane();
    dtp.add(if1); dtp.add(if2);
    framePane.add(BorderLayout.CENTER, dtp);
    framePane.add(BorderLayout.WEST, lateral);
    this.setSize(400,350);
    this.setVisible(true);
  }
}
```

Exemplo: resultado

Exercício

- *1. Construa uma aplicação gráfica que contenha três botões (JButton), um JTextField e um JTextPane distribuídos da seguinte forma*



Use BorderLayout para distribuir os componentes JTextField, JEditorPane e JPanel

Use GridLayout para distribuir os botões

- *2. Experimente usar outros componentes e outros layouts (veja SwingSet demo)*

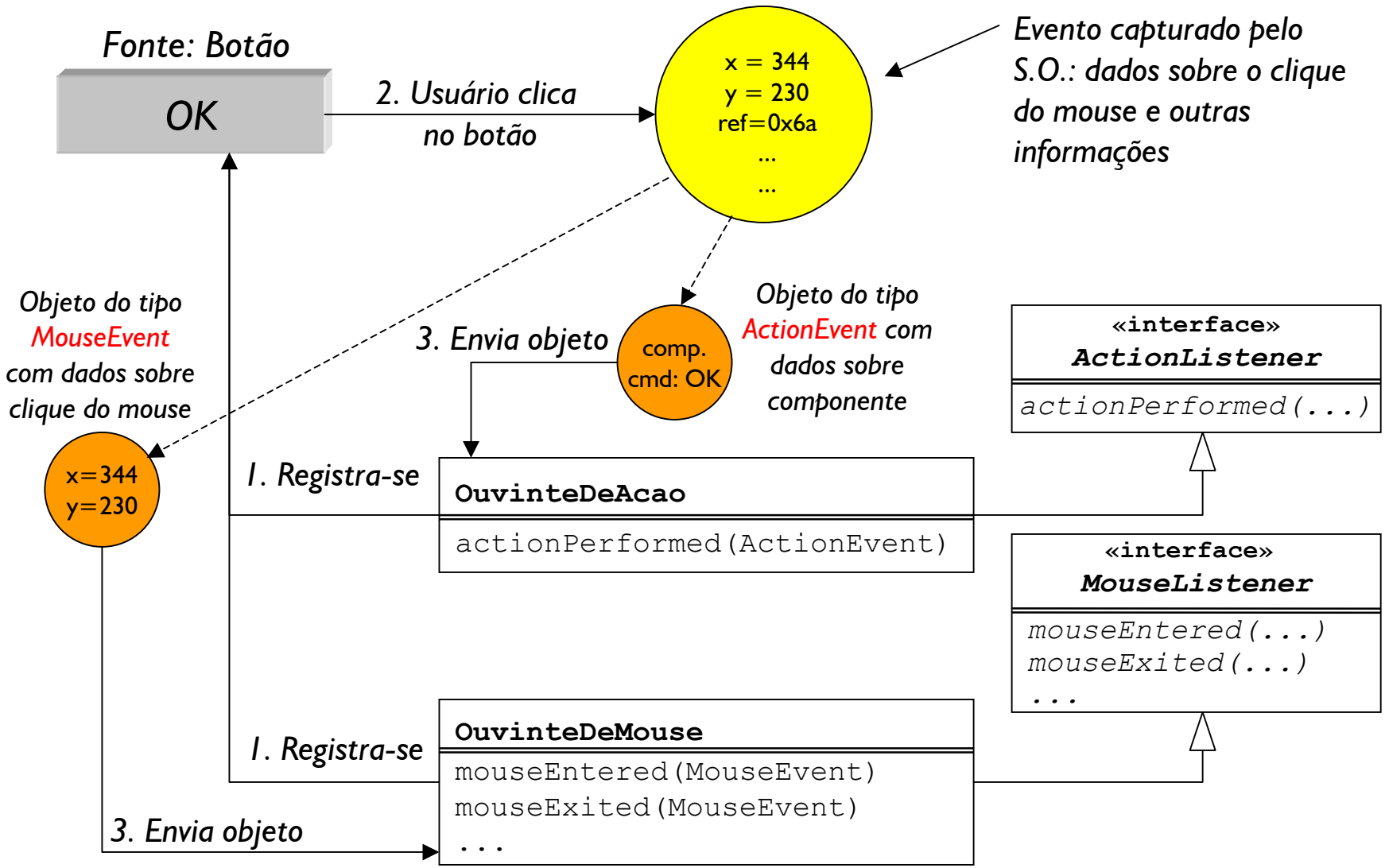
Eventos

- *Eventos em Java são objetos*
  - *Subclasses de java.util.EventObject*
- *Todo evento tem um objeto que é sua fonte*
  - `Object fonte = evento.getSource();`
- *Métodos de ouvintes (listeners) que desejam tratar eventos, recebem eventos como argumento*

```
public void eventoOcorreu(EventObject evento) {
  Object fonte = evento.getSource();
  System.out.println("" +evento+ " em " +fonte);
}
```

- *Ouvintes precisam ser registrados nas fontes*
  - *Quando ocorre um evento, um método de todos os ouvintes registrados é chamado e evento é passado como argumento*

Fontes, Eventos, Ouvintes



Eventos da Interface Gráfica

- Descendentes de *java.awt.event.AWTEvent*
- Divididos em categorias (*java.awt.event*)
  - *ActionEvent* (fonte: componentes de ação)
  - *MouseEvent* (fonte: componentes afetados pelo mouse)
  - *ItemEvent* (fonte: checkboxes e similares)
  - *AdjustmentEvent* (fonte: scrollbars)
  - *TextEvent* (fonte: componentes de texto)
  - *WindowEvent* (fonte: janelas)
  - *FocusEvent* (fonte: componentes em geral)
  - *KeyEvent* (fonte: componentes afetados pelo teclado)
  - ...

Listeners

- *Cada evento tem uma interface Listener correspondente que possui métodos padrão para tratá-los*
  - *ActionEvent: ActionListener*
  - *MouseEvent: MouseListener e MouseMotionListener*
  - *ItemEvent: ItemListener*
  - *AdjustmentEvent: AdjustmentListener*
  - *TextEvent: TextListener*
  - *WindowEvent: WindowListener*
  - *FocusEvent: FocusListener*
  - *KeyEvent: KeyListener*

    ...
  - *XXXEvent: XXXListener*

Como ligar a fonte ao listener

- *Na ocorrência de um evento, em uma fonte, todos os listeners registrados serão notificados*
  - *É preciso antes cadastrar os listeners na fonte*

```
fonte.add<Listener>(referência_para_listener);
```

- *Exemplo:*
  -
```
JButton button = new JButton("Fonte");
ActionListener ouvinte1 = new OuvinteDoBotao();
MouseListener ouvinte2 = new OuvinteDeCliques();
button.addActionListener(ouvinte1);
button.addMouseListener(ouvinte2);
```
- *O mesmo objeto que é fonte às vezes também é listener, se implementar as interfaces*
  - *Ainda assim, é necessário registrar a fonte ao listener (o objeto não adivinha que ele mesmo tem que capturar seus eventos)*

```
this.addWindowListener(this);
```

Fonte *Listener*

Como implementar um listener

- *Crie uma nova classe que declare implementar o(s) listener(s) desejado(s)*

```
public class MeuListener implements
                ActionListener, ItemListener { ...}
```

- *Implemente cada um dos métodos da(s) interface(s)*

```
public void actionPerformed(ActionEvent e) { ...}

public void itemStateChanged(ItemEvent e) {...}
```

- *Veja a documentação sobre o listener usado e o evento correspondente (para saber que métodos usar para obter suas informações)*
  - *Todos os métodos são public void*
  - *Todos recebem o tipo de evento correspondente ao tipo do listener como argumento*

Quais os listeners, métodos, eventos?

- *Consulte a documentação no pacote* **`java.awt.event`**
- *Veja em cada listener a assinatura dos métodos que você deve implementar*

```
public void actionPerformed(ActionEvent evt)
```

- *Veja em cada evento os métodos que você pode chamar dentro do listener para obter as informações desejadas (por exemplo textos, coordenadas, teclas apertadas, etc.)*

```
String comando = actionEvent.getActionCommand();
```

- *Veja em cada componente-fonte os métodos que você pode chamar para obter informações sobre o componente:*

```
Object fonte = evento.getSource();
if (fonte instanceof JButton)
   JButton b = (JButton) fonte;
   String label = b.getLabel();
```

Alguns Eventos, Listeners e Métodos

`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent)`
`ItemEvent`	`ItemListener`	`itemStateChanged(ItemEvent)`

`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`

`MouseEvent`	`MouseListener`	`mouseClicked(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`

	`MouseMotionListener`	`mouseDragged(MouseEvent)` `mouseMoved(MouseEvent)`
`TextEvent`	`TextListener`	`textValueChanged(TextEvent)`

`WindowEvent`	`WindowListener`	`windowActivated(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowClosing(WindowEvent)`
		`windowDeactivated(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowOpened(WindowEvent)`

Adapters

- Alguns listeners possuem uma classe Adapter que implementa todos os métodos, sem instruções
  - Implementação vazia: {}
  - Só existe para listeners que têm mais de um método
- São úteis quando um Ouvinte precisa implementar apenas um dentre vários métodos de um Listener
  - Pode sobrepor a implementação desejada do método do Adapter e não precisa se preocupar com os outros
  - Não são úteis em ouvintes que já estendem outras classes ou quando implementam diferentes listeners
- O nome do adapter é semelhante ao do Listener
  - MouseListener: MouseAdapter,
  - WindowListener: WindowAdapter, ...

Exemplo

```
import java.awt.event.*;

public class Swinggy9 extends JFrame {
   public Swinggy9(String nome) {
      (...)
        JButton b1 = new JButton("Sair");
        JButton b2 = new JButton("Viajar", icone);
        JTextField tf = new JTextField(10);
        b1.setActionCommand("Saindo");
        b2.setActionCommand("Viajando");

        ActionListener listener = new Eco();
        b1.addActionListener(listener);
        b2.addActionListener(listener);
      (...)
   }

   // Classe interna!!
   class Eco implements ActionListener {
       public void actionPerformed(ActionEvent e) {
          tf.setText(e.getActionCommand());
       }
   }
}
```

Tratamento de eventos com classes internas

- *É comum utilizar-se classes internas, mais especificamente, classes anônimas no tratamento de eventos de uma GUI*
- *Vantagens incluem a possibilidade de enxergar os componentes que geralmente são atributos private*
- *O exemplo anterior usa uma classe interna de instância (também chamada de classe aninhada, ou embedded)*
- *O exemplo abaixo usa classes anônimas (compare os dois!):*

```
b1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    tf.setText(e.getActionCommand());
  }
});

b2.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    tf.setText(e.getActionCommand());
  }
});      (...)
```

Exercício

- *3. Implemente os eventos para a aplicação*
  - *Copiar deve acrescentar o texto do JTextField no JEditorPane e limpar o JTextField*
  - *Limpar deve limpar o JTextField*
  - *Sair deve sair do programa*
- *4. Implemente os botões como itens do menu "Operações"*
  - *Use JMenuBar, JMenu e JMenuItem*
- *5. Implemente um JToggleButton "desenhar/escrever" que troque o JTextPane por um JCanvas e permita rabiscar com o mouse (use MouseEvent)*
  - *Veja aplicação-exemplo no CD*

Exercício

- *6. Implemente uma interface gráfica para a aplicação da biblioteca (capítulo 7).*
  - *Use o JTabbedPane para criar diferentes painéis: um para entrada de Agentes (autores, editores), outro para entrada de Publicações (livros, revistas, artigos) e outro para buscas.*
  - *Use um combo-box para selecionar cada tipo de agente ou publicação*
  - *Desabilite campos não utilizados.*
  - *Implemente os eventos chamando os métodos da fachada (Biblioteca)*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Helder da Rocha
www.argonavis.com.br

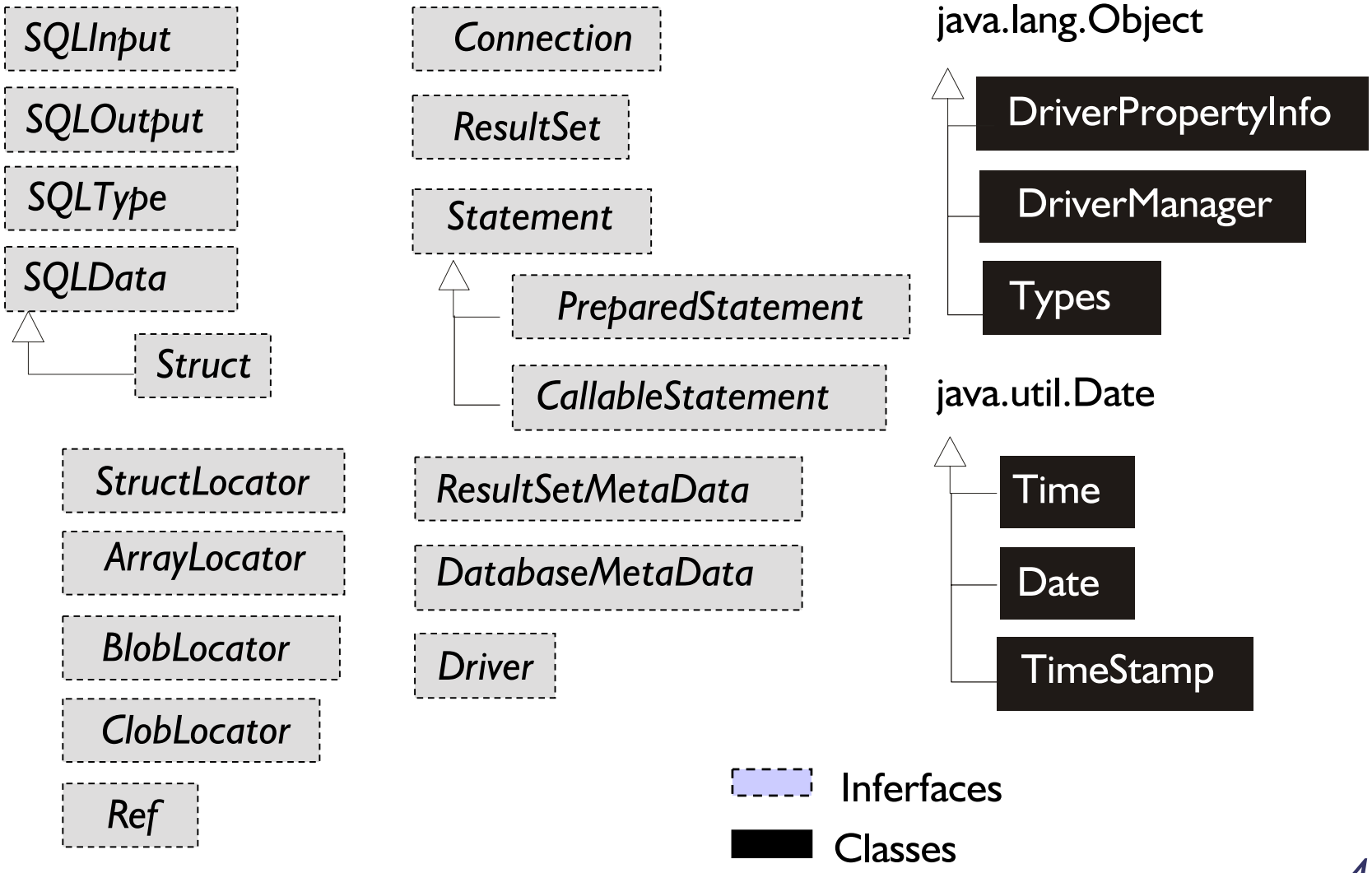
JDBC

- *JDBC é uma interface baseada em Java para acesso a bancos de dados através de SQL.*
  - *Pacote Java padrão:* ***java.sql***
  - *Baseada em ODBC*
- *Usando JDBC, pode-se obter acesso direto a bancos de dados através de applets e outras aplicações Java*
- *Este módulo apresenta uma introdução superficial do JDBC mas suficiente para integrar aplicações Java com bancos de dados relacionais que possuam drivers JDBC*
  - *Não são abordados Connection Pools nem DataSources*

JDBC

- JDBC é uma interface de nível de código
  - Código SQL é usado explicitamente dentro do código Java
  - O pacote java.sql consiste de um conjunto de classes e interfaces que permitem embutir código SQL em métodos.
- Com JDBC é possível construir uma aplicação Java para acesso a qualquer banco de dados SQL.
  - O banco deve ter pelo menos um driver ODBC, se não tiver driver JDBC
- Para usar JDBC é preciso ter um driver JDBC
  - O J2SE distribui um driver ODBC que permite o acesso a bancos que não suportam JDBC mas suportam ODBC

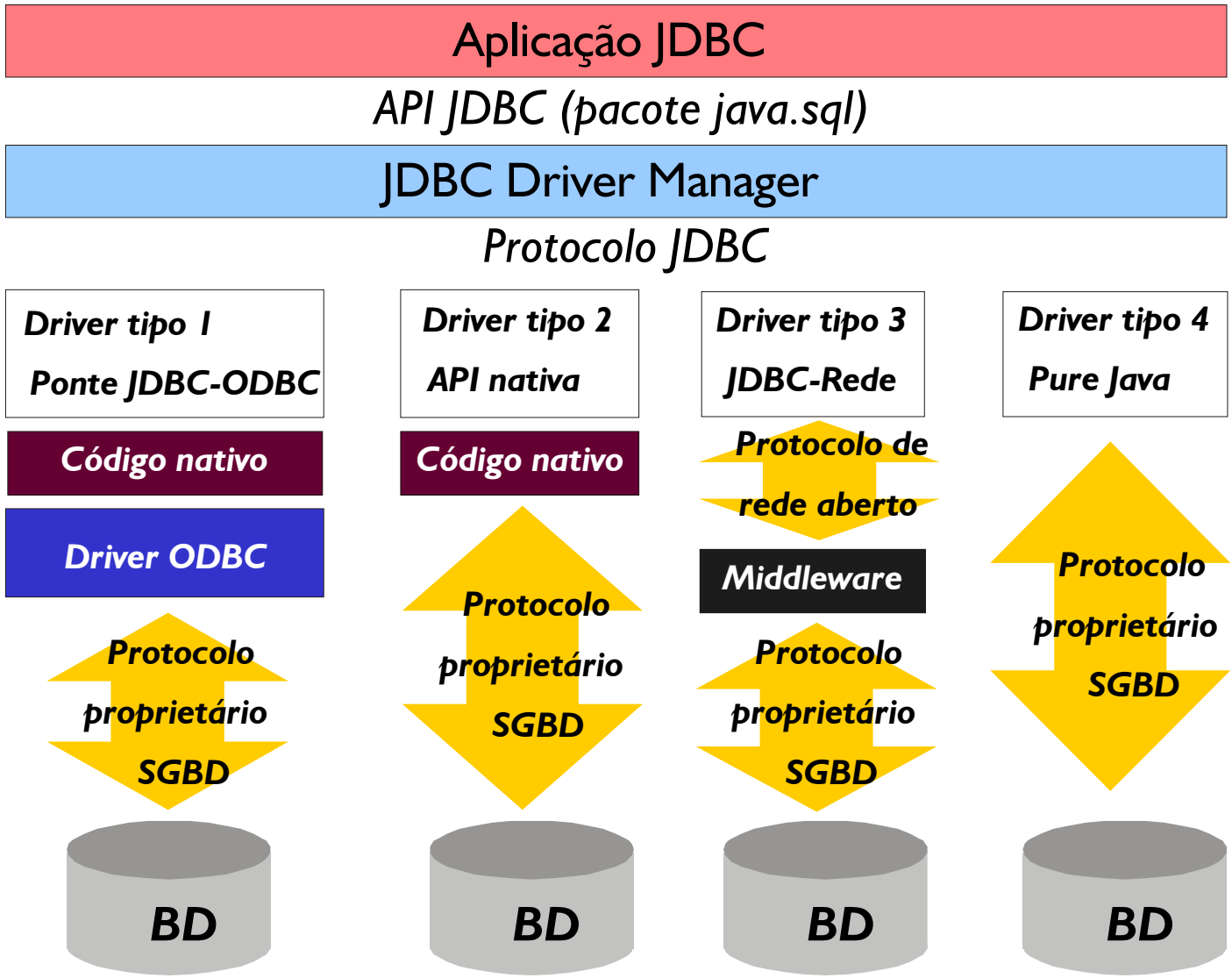
Pacote java.sql



Tipos de Drivers JDBC

- Tipo 1: ponte ODBC-JDBC
  - Usam uma ponte para ter acesso a um banco de dados. Este tipo de solução requer a instalação de software do lado do cliente.
- Tipo 2: solução com código nativo
  - Usam uma API nativa. Esses drivers contém métodos Java implementados em C ou C++. Requer software no cliente.
- Tipo 3: solução 100% Java no cliente
  - Oferecem uma API de rede via middleware que traduz requisições para API do driver desejado. Não requer software no cliente.
- Tipo 4: solução 100% Java
  - Drivers que se comunicam diretamente com o banco de dados usando soquetes de rede. É uma solução puro Java. Não requer código adicional do lado do cliente.

Arquitetura JDBC



URL JDBC

- *Uma aplicação JDBC pode carregar ao mesmo tempo diversos drivers.*
- *Para determinar qual driver será usado usa-se uma URL:*

  jdbc:<subprotocolo>:<dsn>

  - *A aplicação usa o subprotocolo para identificar e selecionar o driver a ser instanciado.*
  - *O dsn é o nome que o subprotocolo utilizará para localizar um determinado servidor ou base de dados.*
  - *Sintaxe dependente do fabricante. Veja alguns exemplos:*

    ```
    jdbc:odbc:anuncios
    jdbc:oracle:thin:@200.206.192.216:1521:exemplo
    jdbc:mysql://alnitak.orion.org/clientes
    jdbc:cloudscape:rmi://host:1098/MyDB;create=true
    ```

DriverManager e Driver

- A interface Driver é utilizada apenas pelas implementações de drivers JDBC
  - É preciso carregar a classe do driver na aplicação que irá utilizá-lo. Isto pode ser feito com Class.forName():

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- A classe DriverManager manipula objetos do tipo Driver.
  - Possui métodos para registrar drivers, removê-los ou listá-los.
  - É usado para retornar Connection, que representa uma conexão a um banco de dados, a partir de uma URL JDBC recebida como parâmetro

```
Connection con =
   DriverManager.getConnection
                          ("jdbc:odbc:dados",
                           "nome","senha");
```

Connection, ResultSet e Statement

- Interfaces que contém métodos implementados em todos os drivers JDBC.
- Connection
  - Representa uma conexão ao banco de dados, que é retornada pelo DriverManager na forma de um objeto.
- Statement
  - Oferece meios de passar instruções SQL para o sistema de bancos de dados.
- ResultSet
  - É um cursor para os dados recebidos.

Statement

- *Obtendo-se um objeto Connection, chama-se sobre ele o método createStatement() para obter um objeto do tipo Statement:*

```
Statement stmt = con.createStatement()
```

que poderá usar métodos como execute(), executeQuery(), executeBatch() e executeUpdate() para enviar instruções SQL ao BD.

- *Subinterfaces:*
  - *PreparedStatement e CallableStatement*

```
PreparedStatement pstmt =
                        con.prepareStatement(...);
CallableStatement cstmt = con.prepareCall(...);
```

Enviando instruções

- *Exemplo de uso de Statement*

```
stmt.execute("CREATE TABLE dinossauros "
             + "(codigo INT PRIMARY KEY, "
             + "genero CHAR(20), "
             + "especie CHAR(20));");

int linhasModificadas =
  stmt.executeUpdate("INSERT INTO dinossauros "
           + "(codigo, genero, especie) VALUES "
           + "(499,'Fernandosaurus','brasiliensis')");

ResultSet cursor =
  stmt.executeQuery("SELECT genero, especie " +
                    " FROM dinossauros " +
                    " WHERE codigo = 355");
```

ResultSet

- *O método executeQuery(), da interface Statement, retorna um objeto ResultSet.*
  - *Cursor para as linhas de uma tabela.*
  - *Pode-se navegar pelas linhas da tabela recuperar as informações armazenadas nas colunas*
- *Os métodos de navegação são*
  - `next()`, `previous()`, `absolute()`, `first()` e `last()`
- *Métodos para obtenção de dados:*
  - `getInt()`
  - `getString()`
  - `getDate()`
  - `getXXX()`, ...

Tipos JDBC e métodos getXXX()

Método de ResultSet	*Tipo de dados SQL92*
`getInt()`	`INTEGER`
`getLong()`	`BIG INT`
`getFloat()`	`REAL`
`getDouble()`	`FLOAT`
`getBignum()`	`DECIMAL`
`getBoolean()`	`BIT`
`getString()`	`CHAR, VARCHAR`
`getDate()`	`DATE`
`getTime()`	`TIME`
`getTimestamp()`	`TIME STAMP`
`getObject()`	*Qualquer tipo (Blob)*

ResultSet

- *Exemplo de uso de ResultSet*

```
ResultSet rs =
   stmt.executeQuery("SELECT Numero, Texto, "
                     + " Data FROM Anuncios");

while (rs.next()) {
     int x = rs.getInt("Numero");
     String s = rs.getString("Texto");
     java.sql.Date d = rs.getDate("Data");
     // faça algo com os valores obtidos...
}
```

Transações

- *Permite a execução atômica de comandos enviados ao banco. Implementada através dos métodos de Connection*
  - *commit()*
  - *rollback()*
  - *setAutoCommit(boolean autoCommit): default é true.*
- *Por default, as informações são processadas a medida em que são recebidas. Para mudar:*

```
con.setAutoCommit(false);
```

- *Agora várias instruções podem ser acumuladas.*
  Para processar:

```
con.commit();
```

- *Se houver algum erro e todo o processo necessitar ser desfeito, pode-se emitir um ROLLBACK usando:*

```
con.rollback();
```

PreparedStatement

- *Statement pré-compilado que é mais eficiente quando várias queries similares são enviadas com parâmetros diferentes*
- *String com instrução SQL é preparado previamente, deixando-se "?" no lugar dos parâmetros*
- *Parâmetros são inseridos em ordem, com setXXX() onde XXX é um tipo igual aos retornados pelos métodos de ResultSet*

```
String sql = "INSERT INTO Livros VALUES(?, ?, ?)";
PreparedStatement cstmt = con.prepareStatement(sql);
cstmt.setInt(1, 18943);
cstmt.setString(2, "Lima Barreto");
cstmt.setString(3, "O Homem que Sabia Javanês");
cstmt.executeUpdate();
...
```

Stored Procedures

- *Procedimentos desenvolvidos em linguagem proprietária do SGBD (stored procedures) podem ser chamados através de objetos CallableStatement*
- *Parâmetros são passados da mesma forma que em instruções PreparedStatement*
- *Sintaxe*
  - `con.prepareCall("{ call proc_update(?, ?, ...) }");`
  - `con.prepareCall("{ ? = call proc_select(?, ?, ...) }");`

```
CallableStatement cstmt =
   con.prepareCall("{? = call sp_porAssunto(?)}";
cstmt.setString(2, "520.92");
ResultSet rs = cstmt.executeQuery();
...
```

Fechar conexão e Exceções

- *Após o uso, os objetos Connection, Statement e ResultSet devem ser fechados. Isto pode ser feito com o método close():*
  - `con.close();`
  - `stmt.close();`
  - `rs.close();`
- *A exceção SQLException é a principal exceção a ser observada em aplicações JDBC*

Metadados

- *Classe DatabaseMetaData: permite obter informações relacionadas ao banco de dados*

```
Connection con; (...)
DatabaseMetaData dbdata = con.getMetaData();
String nomeDoSoftwareDoBanco =
                    dbdata.getDatabaseProductName();
```

- *Classe ResultSetMetaData: permite obter informações sobre o ResultSet, como quantas colunas e quantas linhas existem na tabela de resultados, qual o nome das colunas, etc.*

```
ResultSet rs; (...)
ResultSetMetaData meta = rs.getMetaData();
int colunas = meta.getColumnCount();
String[] nomesColunas = new String[colunas];
for (int i = 0; i < colunas; i++) {
    nomesColunas[i] = meta.getColumnName(i);
}
```

Resources (javax.sql)

- O pacote javax.sql, usado em aplicações J2EE, contém outras classes e pacotes que permitem o uso de conexões JDBC de forma mais eficiente e portável
- javax.sql.DataSource: obtém uma conexão a partir de um sistema de nomes JNDI (previamente registrada)

```
Context ctx = new InitialContext();
DataSource ds =
   (DataSource)ctx.lookup("jdbc/EmployeeDB");
Connection con = ds.getConnection();
```

  - DataSource é uma alternativa mais eficiente que DriverManager: possui pool de conexões embutido
- javax.sql.RowSet e suas implementações
  - Extensão de ResultSet
  - Permite manipulação customizada de ResultSet

Padrões de Projeto implementados em JDBC

- Drivers JDBC implementam vários padrões de projeto. Os principais são
  - **Bridge**: define uma solução para que uma implementação (o driver que permite a persistência dos objetos) seja independente de sua abstração (a hierarquia de objetos)
  - **Abstract Factory**: permite que hierarquias de classes sejam plugadas e objetos diferentes, de mesma interface, sejam produzidos (uma createStatement() cria um objeto Statement com a implementação do driver instalado.)
  - **Factory Method**: é a implementação dos métodos getConnection(), createStatement(), etc. que devolvem um objeto sem que a sua implementação seja conhecida.
  - **Iterator**: o ResultSet e seu método next()

Exercícios

- *1. Construa uma aplicação Java simples que permita que o usuário envie comandos SQL para um banco de dados e tenha os resultados listados na tela.*
  - *Veja detalhes em README.txt no diretório cap18/*
- *2. Crie uma aplicação com o mesmo objetivo que a aplicação do exercício 1, mas, desta vez, faça com que os dados sejam exibidos dentro de vários JTextField (um para cada campo) organizados lado a lado como uma planilha.*
  - *Use JScrollPane() para que os dados caibam na tela*
  - *Use um JPanel que posicione os JTextField (de tamanho fixo) lado a lado e outro, que possa crescer dinamicamente, para os registros (coleções de JTextField)*
- *3. Descubra como usar o JTable (são várias classes) e refaça o exercício 2.*

Exercícios 2

- *4. Crie uma classe RepositorioDadosBD que implemente RepositorioDados da aplicação biblioteca*
  - *a) Crie tabelas autor, editor, livro, revista e artigo*
  - *b) Crie tabelas de relacionamentos: publicacao_autor, palavras_chave, artigo_revista*
  - *b) Implemente os métodos usando SQL e JDBC*
  - *c) Teste a aplicação*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Helder da Rocha
www.argonavis.com.br

Sobre este módulo

- Este módulo pretende apenas cobrir conceitos essenciais sobre programação em rede com Java
  - Como criar um servidor e um cliente TCP/IP
  - Como abrir uma conexão TCP/IP
  - Como ler de uma conexão
  - Como escrever para uma conexão
- Classes abordadas do pacote java.net
  - Socket e ServerSocket
  - InetAddress
  - URL
- Para maiores detalhes, consulte as referências no final do capítulo

Pacote java.net

- *O pacote java.net contém classes para implementar comunicação através da rede*
- *Fáceis de usar. Semelhante à criação de arquivos:*

```
Socket sock = new Socket("www.x.com", 80);                    (1)
PrintWriter os = new PrintWriter(
   new OutputStreamWriter(
      sock.getOutputStream()));
BufferedReader is = new BufferedReader(
   new InputStreamReader(
      sock.getInputStream());
os.println("GET / HTTP/1.0\n\n");                             (2)
String linha = "";
while ((linha = is.readLine()) != null) {
   System.out.println(linha);                                 (3)
} // ... feche o socket com sock.close();
```

(1) Abre socket para servidor Web, (2) envia comando e (3) imprime resposta

TCP/IP no pacote java.net

- A comunicação via protocolo **TCP** (Transfer Control Protocol), confiável, é suportada pelas classes
  - Socket (soquete de dados)
  - ServerSocket (soquete do servidor).
- A comunicação via **UDP** (Unreliable Datagram Protocol), não-confiável, é suportada pelas classes
  - DatagramSocket (soquete de dados UDP),
  - DatagramPacket (pacote UDP)
  - MulticastSocket (soquete UDP para difusão).
- Endereçamento
  - InetAddress (representa um endereço na Internet)
  - URL (representa uma URL)

URL

- Representa uma **URL**
- Principais métodos
  - openStream() obtém um InputStream para os dados
  - openConnection(): retorna um objeto URLConnection que contém métodos para ler o cabeçalho dos dados
  - getContent(): retorna os dados diretamente como Object se conteúdo for conhecido (texto, imagens, etc.)
- Para imprimir a página HTML de um site

```
try {
   URL url = new URL("http://www.site.com");
   InputStreamReader reader =
      new InputStreamReader(url.openStream());
   BufferedReader br = new BufferedReader(reader);
   String linha = "";
   while ( (linha = br.readLine()) != null) {
      System.out.println(linha);
   }
} catch (MalformedURLException e) { ... }
```

InetAddress

- Representa um endereço Internet
- Principais métodos estáticos construtores
  - getLocalHost() retorna InetAddress
  - getByName(String host) retorna InetAddress
- Principais métodos de instância
  - getHostAddress() retorna String com IP do InetAddress
  - getHostName() retorna String com nome no InetAddress
- Para descobrir o IP e nome da máquina local:

```
InetAddress address = InetAddress.getLocalHost();
String ip = address.getHostAddress();
String nome = address.getHostName();
```

Socket

- *Um dos lados de uma conexão bidirecional TCP*
- *Principais métodos servem para obter fluxos de entrada e saída*
  - *getInputStream()*
  - *getOutputStream()*
  - *close()*
- *Exemplo*

```
InetAddress end =
      InetAddress.getByName("info.acme.com");
Socket con = new Socket(end, 80);
InputStream dados = con.getInputStream();
OutputStream comandos =
                        con.getOutputStream();
```

- *Depois de obtido os fluxos, basta ler ou enviar dados*

Socket (2)

- *Para ler ou gravar caracteres ao invés de bytes, pode-se decorar os fluxos obtidos de um socket com as classes Reader e Writer:*

```
Socket con = new
        Socket("maquina.com.br", 4444);

Reader r = new InputStreamReader(
                    con.getInputStream());

Writer w = new OutputStreamWriter(
                    con.getOutputStream());

// Use aqui os fluxos de dados

con.close();
```

ServerSocket

- *Com ServerSocket pode-se implementar um servidor que fica escutando uma porta a espera de um cliente*
- *Principal método*
  - *accept(): aceita a conexão e retorna o seu socket*
- *Exemplo de servidor dedicado*

```
ServerSocket escuta = new ServerSocket(80);
while(true) {
   Socket cliente = escuta.accept();  // espera
   InputStream comandos =
                        cliente.getInputStream();
   OutputStream dados =
                        cliente.getOutputStream();
   // ... use os dados
   cliente.close();
}
```

Exceções de rede

- Várias exceções podem ocorrer em um ambiente de rede
  - O programa deve tomar medidas para reduzir o impacto das exceções inevitáveis, como rede fora do ar ou conexão recusada
  - O compilador irá informar, durante o desenvolvimento, as exceções que precisam ser declaradas ou tratadas
- As exceções mais comuns do pacote java.net são
  - SocketException
  - MalformedURLException
  - UnknownHostException
  - ProtocolException
- Operações de timeout, liberação de threads, sincronização, transações, etc. devem ser implementados pelo programador em aplicações de rede
  - Não há exceções tratando esses problemas automaticamente

Exercícios

- 1. Escreva um programa que descubra e imprima o número IP da sua máquina
- 2. Escreva um programa que
  - Conecte-se na porta HTTP (geralmente 80) de um servidor conhecido
  - Envie o comando: "`GET / HTTP/1.0\n\n`"
  - Grave o resultado em um arquivo resultado.html
- 3. Servidor dedicado: escreva um servidor simples que responda a comandos da forma "GET arquivo".
  - Localize o arquivo e imprima-o no OutputStream
  - Escreva um cliente que receba o arquivo e grave-o localmente

Exercícios (2)

- 4. Servidor multithreaded: Escreva um programa que use um ServerSocket para aguardar conexões de um cliente. O programa deverá ter duas partes:
  - (1) uma classe principal (Servidor) que fica escutando a porta escolhida (número acima de 1024) e
  - (2) uma classe que estende Thread (Conexao) e que irá tratar as requisições do cliente.

  O servidor deverá imprimir na tela todos os comandos enviados por todos os clientes.
  - Os clientes enviam mensagens de texto, como um chat.
- 5. Crie um cliente para a aplicação acima e teste-o em várias máquinas diferentes.

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br

Fundamentos de Objetos Remotos

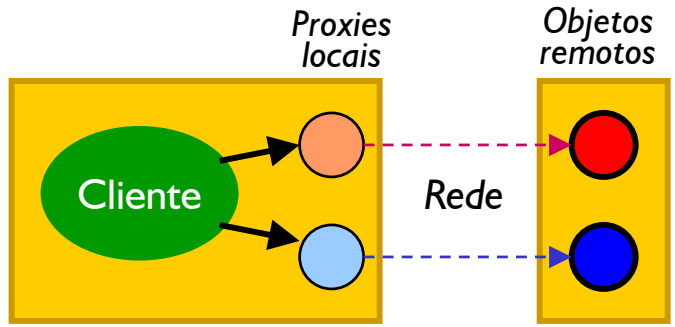
Helder da Rocha
www.argonavis.com.br

Sobre este módulo

- *Este módulo tem como objetivo dar uma visão geral, porém prática, da criação e uso de objetos remotos em Java com RMI*
- *Para permitir a demonstração de uma aplicação simples, vários conceitos importantes, essenciais em aplicações RMI reais foram omitidos, como*
  - *Necessidade de implantar um gerente de segurança*
  - *Necessidade de divulgar um codebase (Classpath distribuído)*
- *Para maiores detalhes, consulte as fontes de referência para este curso*

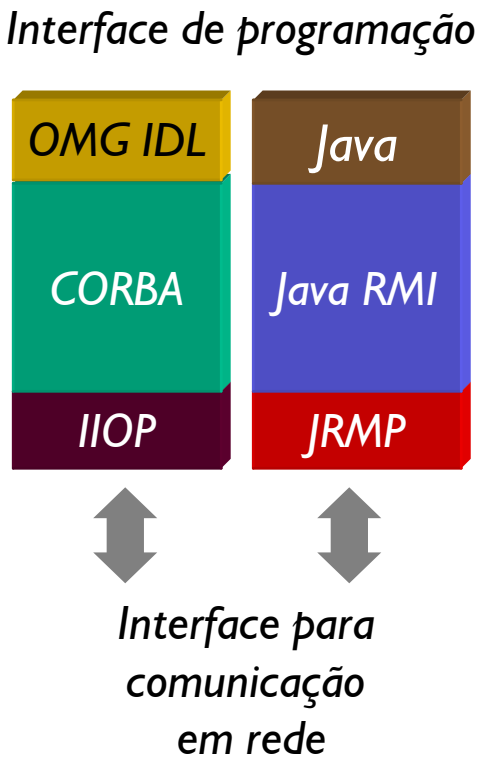
O que são objetos remotos

- Objetos remotos são objetos cujos métodos podem ser chamados remotamente, como se fossem locais
  - Cliente precisa, de alguma forma, localizar e obter uma instância do objeto remoto (geralmente através de um proxy intermediário gerado automaticamente)
  - Depois que o cliente tem a referência, faz chamadas nos métodos do objeto remoto (através do proxy) usando a mesma sintaxe que usaria se o objeto fosse local
- Objetos remotos abstraem toda a complexidade da comunicação em rede
  - Estende o paradigma OO além do domínio local
  - Torna a rede transparente



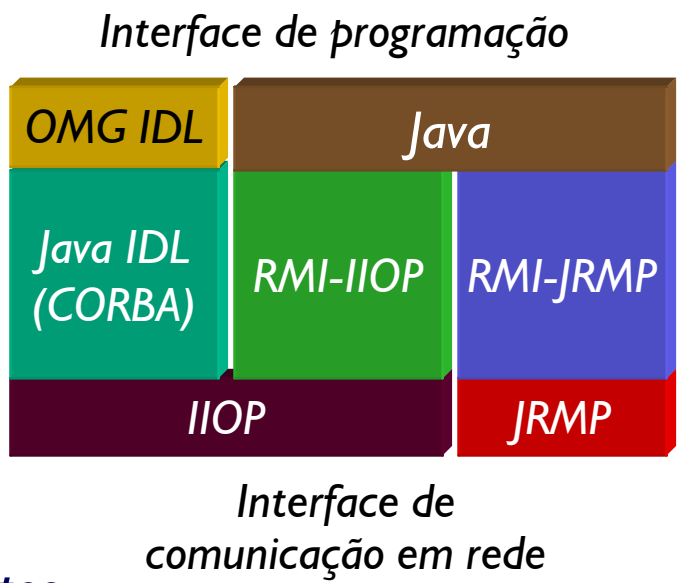
Como implementar objetos remotos

- Para usar objetos remotos, é preciso ter uma infraestrutura que cuide da geração das classes, comunicação e outros detalhes
- Há duas soluções disponíveis para implementar objetos remotos em Java
  - OMG **CORBA**: requer o uso, além de Java, da linguagem genérica OMG IDL, mas suporta integração com outras linguagens
  - Java **RMI**: para soluções 100% Java
- As duas soluções diferem principalmente na forma de implementação
- Em ambas, um programa cliente poderá chamar um método em um objeto remoto da mesma maneira como faz com um método de um objeto local



Objetos remotos com Java RMI

- Java RMI (Remote Method Invocation) pode ser implementado usando protocolos e infraestrutura próprios do Java (JRMP e RMI Registry) ou usando IIOP e ORBs, próprios do CORBA
- JRMP - Java Remote Method Protocol
  - Pacote `java.rmi` - RMI básico
  - Ideal para aplicações 100% Java.
- IIOP - Internet Inter-ORB Protocol
  - Pacote `javax.rmi` - RMI sobre IIOP
  - Ideal para ambientes heterogêneos.
- A forma de desenvolvimento é similar
  - Há pequenas diferenças na geração da infraestrutura (proxies) e registro de objetos



- RMI sobre IIOP permite programação Java RMI e comunicação em CORBA, viabilizando integração entre Java e outras linguagens sem a necessidade de aprender OMG IDL

RMI: funcionamento básico

- Um objeto remoto previamente registrado é obtido, através de servidor de nomes especial: RMI Registry.
  - Permite que os objetos publicamente acessíveis através da rede sejam referenciados através de um nome.
- Serviço de nomes: classe java.rmi.Naming
  - Método Naming.lookup() consulta um servidor de nomes RMI e obtém uma instância de um objeto remoto
- Exemplo (jogo de batalha naval):

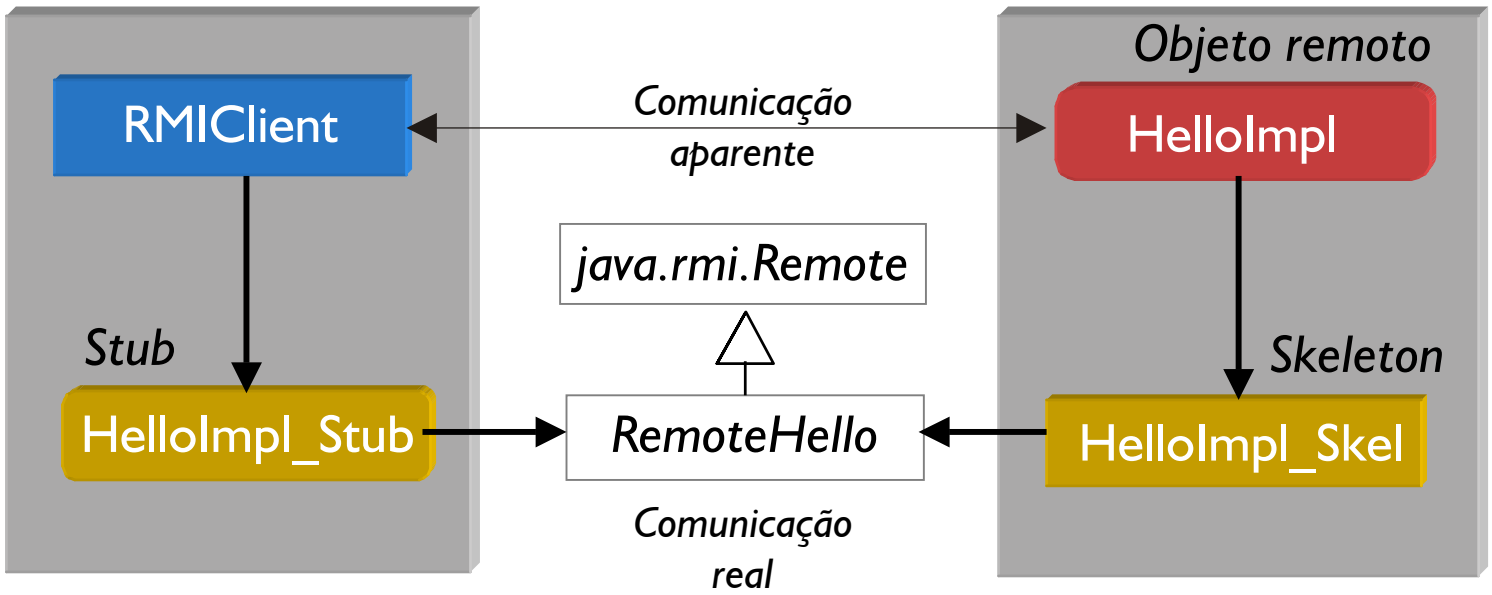
```
Territorio mar =
  (Territorio)Naming.lookup("rmi://gamma/caspio");
```

- Agora é possível chamar métodos remotos de mar:

```
tentativa[i] = mar.atira("C", 9);
```

Arquitetura RMI

- *Uma aplicação distribuída com RMI tem acesso transparente ao objeto remoto através de sua Interface Remota*
  - *A "Interface Remota" é uma interface que estende java.rmi.Remote*
  - *A partir da Interface Remota e implementação do objeto remoto o sistema gera objetos (proxies) que realizam todas as tarefas necessárias para viabilizar a comunicação em rede*



Padrões de Projeto: ***Proxy***

- *A implementação RMI é um exemplo do padrão de projeto chamado* ***Proxy***
- *Proxy é uma solução para situações onde o objeto de interesse está inacessível diretamente, mas o cliente precisa operar em uma interface idêntica*
  - *A solução oferecida por Proxy é criar uma classe que tenha a mesma interface que o objeto de interesse (implemente a mesma interface Java) e que implemente, em seus métodos, a lógica de comunicação com o objeto inacessível.*
  - *Em RMI, o proxy é o Stub gerado automaticamente pelo ambiente de desenvolvimento (rmic)*

Como usar RMI em 10 passos

- O objetivo deste módulo é oferecer apenas uma introdução básica a Java RMI. Isto será feito através de um exemplo simples
  - 1. Definir a interface
  - 2. Implementar os objetos remotos
  - 3. Implementar um servidor para os objetos
  - 4. Compilar os objetos remotos
  - 5. Gerar stubs e skeletons com rmic
  - 6. Escrever, compilar e instalar o(s) cliente(s)
  - 7. Instalar o stub no(s) cliente(s)
  - 8. Iniciar o RMI Registry no servidor
  - 9. Iniciar o servidor de objetos
  - 10. Iniciar os clientes informando o endereço do servidor.

1. Definir a interface remota

- *Declare todos os métodos que serão acessíveis remotamente em uma interface Java que estenda java.rmi.Remote.*
  - *Todos os métodos devem declarar throws java.rmi.RemoteException.*
- *Isto deve ser feito para cada objeto que será acessível através da rede.*

```
import java.rmi.*;
public interface Mensagem extends Remote {
   public String getMensagem()
                        throws RemoteException;
   public void setMensagem(String msg)
                        throws RemoteException;
}
```

2. Implementar os objetos remotos

- *Cada objeto remoto é uma classe que estende a classe java.rmi.server.UnicastRemoteObject e que implementa a interface remota criada no passo 1.*
- *Todos os métodos declaram causar java.rmi.RemoteException inclusive o construtor, mesmo que seja vazio.*

```
import java.rmi.server.*;
import java.rmi.*;

public class MensagemImpl extends UnicastRemoteObject
                                    implements Mensagem {
   private String mensagem = "Inicial";
   public MensagemImpl() throws RemoteException {}
   public String getMensagem() throws RemoteException {
      return mensagem;
   }
   public void setMensagem(String msg) throws RemoteException {
      mensagem = msg;
   }
}
```

3. Estabelecer um servidor

- *Crie uma classe que*
  - *a) Crie uma instância do objeto a ser servido e*
  - *b) Registre-a (bind ou rebind) no serviço de nomes.*

```
import java.rmi.*;
public class MensagemServer {

    public static void main(String[] args)
                                throws RemoteException {
            Mensagem mens = new MensagemImpl();
            Naming.rebind("mensagens", mens);
            System.out.println("Servidor no ar. "+
                        " Nome do objeto servido: '"
                                + "mensagens" +"'");
    }
}
```

4. Compilar os objetos remotos

- *Compile todas as interfaces e classes utilizadas para implementar as interfaces Remote*
  - `javac Mensagem.java MensagemImpl.java`

5. Gerar stubs e skeletons

- *Use a ferramenta do J2SDK:* ***rmic***
- *Será gerado um arquivo stub*
  - `MensagemImpl_stub.class`

  e um arquivo skeleton
  - `MensagemImpl_skel.class`

  para cada objeto remoto (neste caso, apenas um)
- *RMIC =* ***RMI Compiler***
  - *Use opção -keep se quiser manter código-fonte*
  - *Execute o rmic sobre as implementações do objeto remoto já compiladas:*

```
> rmic MensagemImpl
```

6. Compilar e instalar o(s) cliente(s)

- *Escreva uma classe cliente que localize o(s) objeto(s) no serviço de nomes (java.rmi.Naming)*
  - *a) Obtenha uma instância remota de cada objeto*
  - *b) Use o objeto, chamando seus métodos*

```
import java.rmi.*;
public class MensagemClient {
    public static void main(String[] args)
                                        throws Exception {
          String hostname = args[0];
          String objeto = args[1];
          Object obj =
             Naming.lookup("rmi://" + hostname + "/"
                                        + objeto);
          Mensagem mens = (Mensagem) obj;
          System.out.println("Mensagem recebida: "
                                  + mens.getMensagem());
          mens.setMensagem("Fulano esteve aqui!");
    }
}
```

7. Instalar o stub no(s) cliente(s)

- Distribua o cliente para as máquinas-cliente. A distribuição deve conter
  - Classe(s) que implementa(m) o cliente (HelloMensagem.class)
  - Os stubs (HelloMensagem_stub.class)
  - As interfaces Remote (Mensagem.class)

- Em aplicações reais, os stubs podem ser mantidos no servidor
  - O cliente faz download do stub quando quiser usá-lo
  - Para isto é preciso definir algumas propriedades adicionais (omitidas no nosso exemplo simples) como Codebase (CLASSPATH distribuído), SecurityManager e políticas de segurança (Policy)

8. Iniciar o RMI Registry no servidor

- *No Windows*

```
> start rmiregistry
```

- *No Unix*

```
> rmiregistry &
```

(O RMI Registry fica "calado" quando está rodando)

- *Neste exemplo será preciso iniciar o RMIRegistry no diretório onde estão os stubs e interface Remote*
  - *Isto é para que o RMIRegistry veja o mesmo CLASSPATH que o resto da aplicação*
  - *Em aplicações RMI reais isto não é necessário, mas é preciso definir a propriedade java.rmi.server.codebase contendo os caminhos onde se pode localizar o código*

9. Iniciar o servidor de objetos

- *O servidor é uma aplicação executável que registra os objetos no RMIRegistry. Rode a aplicação:*

```
> java MensagemServer
  Servidor no ar. Nome do objeto servido: mensagens
```

- *Neste exemplo será preciso iniciar o servidor no diretório onde estão os stubs e interface Remote*
  - *Isto é para que o RMIRegistry veja o mesmo CLASSPATH que o resto da aplicação*
  - *Em aplicações RMI reais isto não é necessário, mas é preciso definir a propriedade java.rmi.server.codebase contendo os caminhos onde se pode localizar o código*

10. Iniciar os clientes

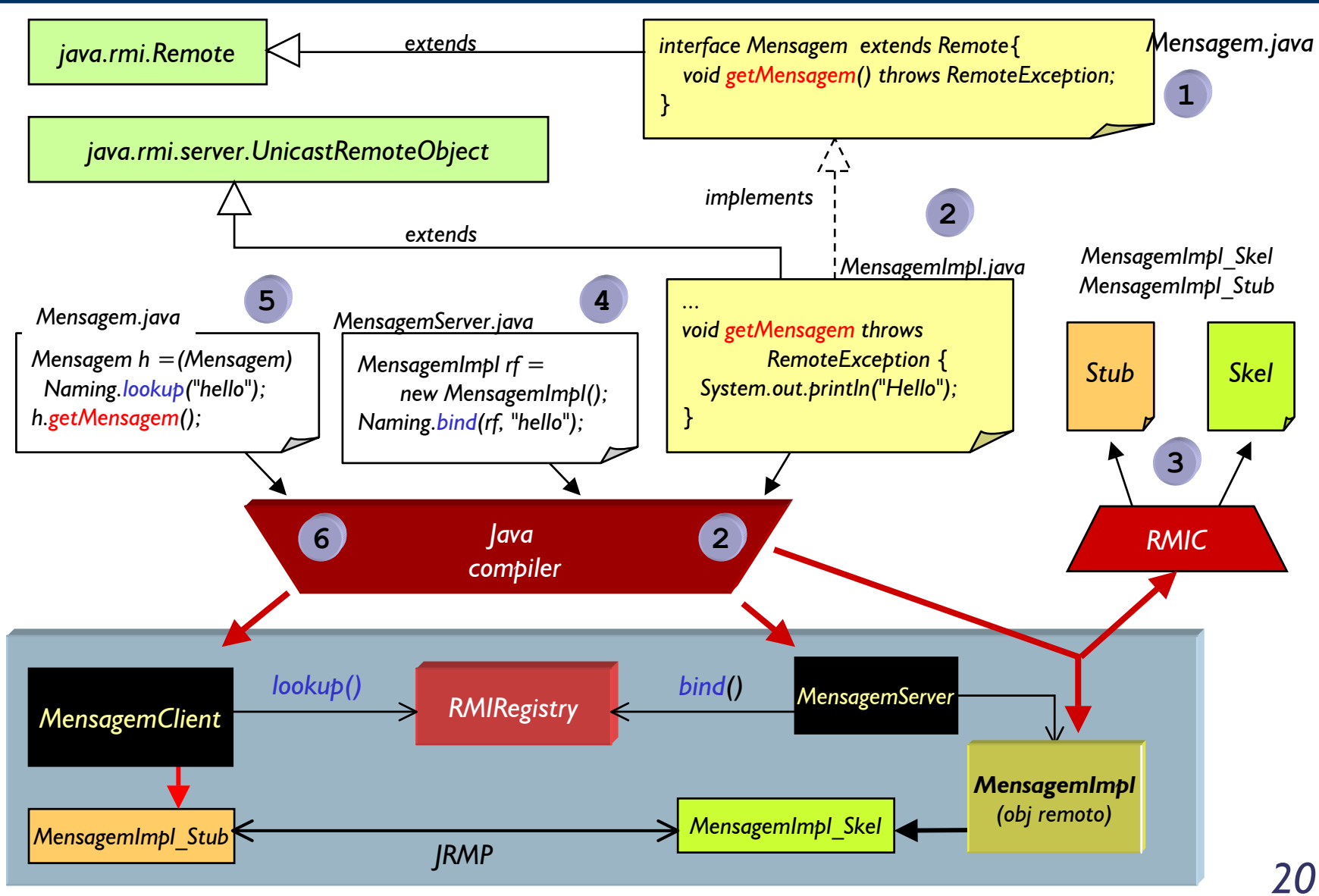
- *Rode o cliente*
  - *Informe o endereço do servidor e objetos a utilizar*

```
> java MensagemClient maquina.com.br mensagens
```

Exercício

Implemente os exemplos mostrados, inventando uma mensagem diferente para seu objeto remoto. Use o seu cliente para acessar os objetos remotos registrados nas máquinas de seus colegas.

Resumo


java.rmi.Remote
extends
interface Mensagem extends Remote{
void getMensagem() throws RemoteException;
}
Mensagem.java
1
java.rmi.server.UnicastRemoteObject
implements
2
extends
MensagemImpl.java
MensagemImpl_Skel
MensagemImpl_Stub
5
Mensagem.java
Mensagem h =(Mensagem)
Naming.lookup("hello");
h.getMensagem();
4
MensagemServer.java
MensagemImpl rf =
new MensagemImpl();
Naming.bind(rf, "hello");
...
void getMensagem throws
RemoteException {
System.out.println("Hello");
}
Stub
Skel
3
RMIC
6
Java
compiler
2
MensagemClient
lookup()
RMIRegistry
bind()
MensagemServer
MensagemImpl
(obj remoto)
MensagemImpl_Stub
JRMP
MensagemImpl_Skel

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br