Modulo Ia
Introdução ao Java

UniverCidade - Prof. Ismael H F Santos

Ementa

- Modulo I - Introdução a Linguagem JAVA
  - Paradigma OO
  - Características da linguagem
  - Plataformas Java
  - Introdução Prática
  - Fundamentos da linguagem
  - Interfaces e Classes Abstratas
  - Classes parametrizadas (não é Generics !)

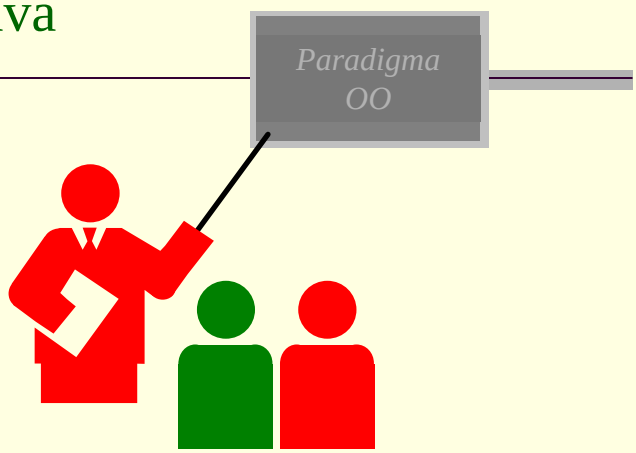
Bibliografia

- *Linguagem de Programação JAVA*
  - *Ismael H. F. Santos, Apostila UniverCidade, 2002*
- *The Java Tutorial: A practical guide for programmers*
  - *Tutorial on-line: http://java.sun.com/docs/books/tutorial*
- *Java in a Nutshell*
  - *David Flanagan, O´Reilly & Associates*
- *Just Java 2*
  - *Mark C. Chan, Steven W. Griffith e Anthony F. Iasi, Makron Books.*
- *Java 1.2*
  - *Laura Lemay & Rogers Cadenhead, Editora Campos*

Livros

- **Core Java 2**, Cay S. Horstmann, Gary Cornell
  - Volume 1 (Fundamentos)
  - Volume 2 (Características Avançadas)
- **Java: Como Programar**, Deitel & Deitel
- **Thinking in Patterns with JAVA**, Bruce Eckel
  - **Gratuito**. http://www.mindview.net/Books/TIJ/

POO-Java



Paradigma OO

- Paradigmas de Programação
  - Programação Procedural ou Imperativo
  - Programação Funcional
  - Programação Lógico
  - Programação Orientada por Objetos

Paradigma OO

- Cenário Exemplo

"João deseja enviar flores para Maria mas ela mora em outra cidade. João vai, então, até a floricultura e pede a José, o floricultor, para que ele envie um bouquet de rosas ao endereço de Maria. José, por sua vez, liga para uma outra floricultura, da cidade de Maria, e pede para que as flores sejam entregues."

Paradigma OO

- Colocação do Problema - nomenclatura

- João precisa resolver um problema;
- Então, ele procura um *agente*, José, e lhe passa uma *mensagem* contendo sua *requisição*: enviar rosas para Maria;
- José tem a *responsabilidade* de, através de algum *método*, cumprir a *requisição*;
- O *método* utilizado por José pode estar *oculto* de João.

Paradigma OO

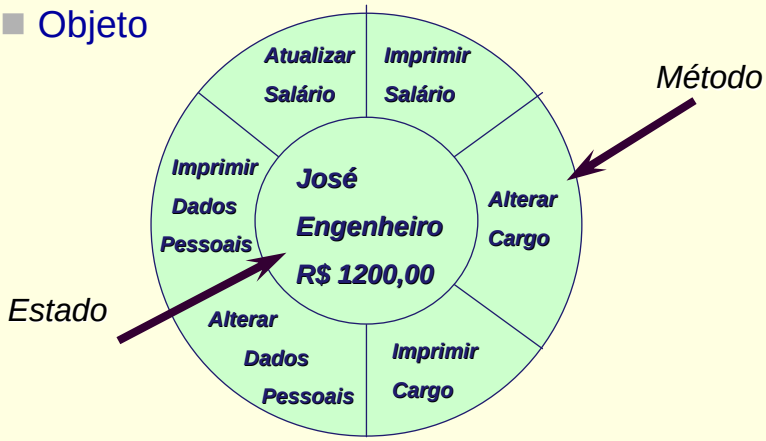
- Modelo OO
  - Uma ação se inicia através do envio de uma mensagem para um agente (um objeto) responsável por essa ação.
  - A mensagem carrega uma requisição, além de toda a informação necessária, isto é os argumentos, para que a ação seja executada.
  - Se o agente receptor da mensagem a aceita, ele tem a responsabilidade de executar um método para cumprir a requisição.

Paradigma OO

- Objetos
  - Estão preparados para cumprir um determinado conjunto de requisições.
  - Recebem essas requisições através de mensagens.
  - Possuem a responsabilidade de executar um método que cumpra a requisição.
  - Possuem um estado representado por informações internas.

Paradigma OO

- Objeto



Paradigma OO

- Classes
  - O conjunto de requisições que um objeto pode cumprir é determinado pela sua classe.
  - A classe também determina que método será executado para cumprir uma requisição.
  - A classe especifica que informações um objeto armazena internamente.
  - Objetos são instâncias de classes.
  - Classes podem ser compostas em hierarquias, através de herança.

Paradigma OO

- Classe

Paradigma OO

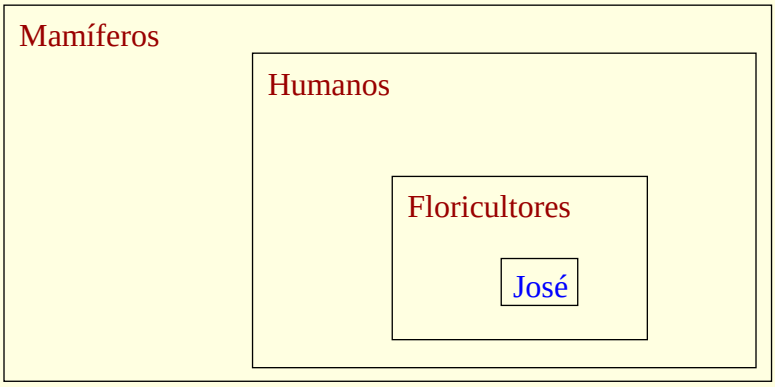
- Encapsulamento
  - As classes possuem uma parte invisível, que é a sua implementação, e uma parte visível que é a sua interface;
  - As operações da interface possibilitam o acesso aos objetos.

Paradigma OO

- Herança
  - **Compartilhamento de atributos e serviços entre classes;**
  - **A definição de classes em termos de outras classes constitui uma hierarquia:**
    - **superclasses (classes ancestrais ou bases)**
    - **subclasses (classes derivadas)**
  - **Recurso utilizado para especializar ou estender classes;**

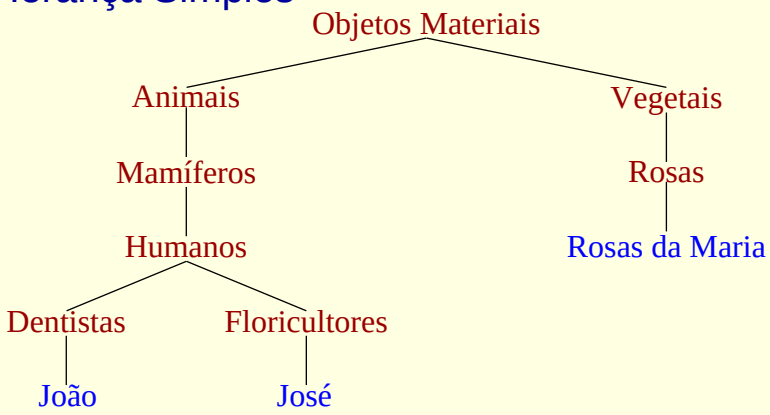
Paradigma OO

- Herança - Hierarquia de Classes



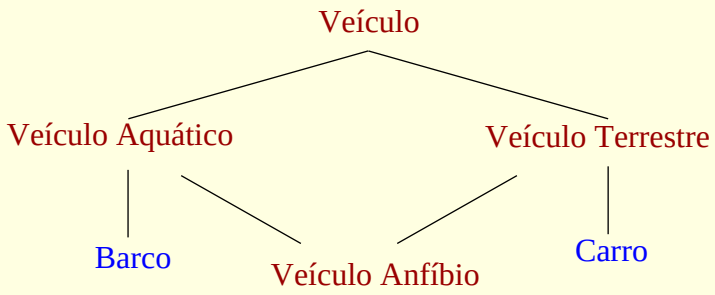
Paradigma OO

- Herança Simples



Paradigma OO

- Herança Múltipla

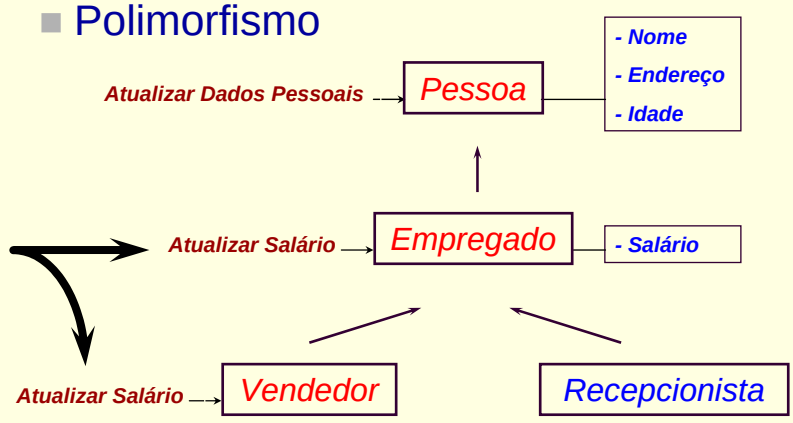


Paradigma OO

- Polimorfismo
  - **Implica na redefinição de um método ou serviço da superclasse. Desta forma, pode haver métodos com diferentes implementações em uma hierarquia;**
  - **Métodos com várias formas, implica em métodos com várias implementações.**

Paradigma OO

- Polimorfismo



Paradigma OO - Sumário

- **Agentes são objetos;**
- **Ações são executadas através da troca de mensagens entre objetos;**
- **Todo objeto é uma instância de uma classe;**
- **Uma classe define uma interface e um comportamento;**
- **Classes podem estender outras classes através de herança.**

Paradigma OO - Sumário

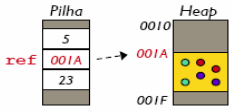
- *Objetos são **conceitos** que têm*
  - *identidade,*
  - *estado e*
  - *comportamento*
- *Características de Smalltalk, resumidas por Allan Kay:*
  - *Tudo (em um programa OO) são objetos*
  - *Um programa é um monte de objetos enviando mensagens uns aos outros*
  - *O espaço (na memória) ocupado 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*

Paradigma OO - Sumário

- *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*

Paradigma OO - Sumário

- *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*

Paradigma OO - Sumário

- *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*

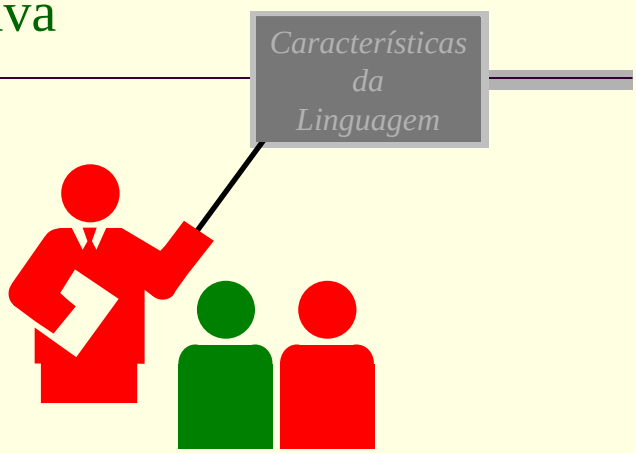
Paradigma OO

- Por que OO ?

Porque promove o desenvolvimento de software com qualidade

- **Correção**
- **Robusto**
- **Extensibilidade**
- **Reusabilidade**
- **Compatibilidade**

POO-Java



Características da Linguagem

- Java é uma Linguagem OO, contendo uma coleção de APIs para o desenvolvimento de aplicações multiplataforma

- 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, 5.0)*

- Java segundo a Sun Microsystems

Simples	object-oriented	distribuída
interpretada	robusta	segura
architecture-neutral	portável	alta performance
multithreaded	dinâmica	

Características da Linguagem

- **Simplicidade**

  - Poucos comandos, ausência de goto (apesar de suportar break para label), mecanismos de tratamento de exceção, não permite sobrecarga de operadores, não suporta herança múltipla, não suporta templates, não utiliza ponteiros, não trabalha com unions ou estruturas; somente classes. Vetores e strings são objetos. Java só trabalha com referências !

Características da Linguagem

- Orientada a Objeto

  - Suporte completo a OOP. Define uma extensa biblioteca de classes, disponibilizadas em pacotes (packages).

    java.lang => classes para definição básica da linguagem (core language)
    java.applet => classes para implementação de Applets
    java.awt => classes para gráficos, texto, janelas, GUI
    java.awt.image => classes para processamento de imagens
    java.awt.event => classes para tratamento de eventos na interface GUI
    java.awt.swing => classes para interface GUI extensão do AWT
    java.awt.peer => classes para interface GUI independente de plataforma
    java.io => classes para input / output
    java.net => classes para network computing
    java.util => classes para tipos de dados úteis (arrays, listas, etc)

  - Diferentemente de C++, a maioria dos tipos em Java são objetos, com exceção dos tipos: numéricos, caracter e boolean.

Características da Linguagem

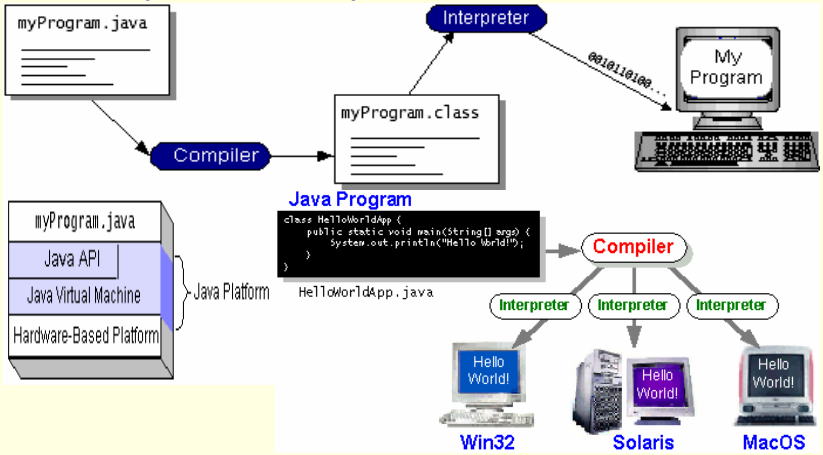
- Distribuída
  - Java suporta aplicações Cliente/Servidor por ser uma linguagem distribuída. Suporta vários níveis de conectividade através de classes presentes no java.net.package. Suporta conexão através de sockets (socket class).
- Robusta
  - Java é uma linguagem fortemente tipada (mais que C++). Requer declaração explícita de métodos (idem ANSI-C). Não suporta ponteiros, eliminando assim a possibilidade de invasões de área e manipulação errônea com aritmética de ponteiros.

Características da Linguagem

- Interpretada
  - O compilador Java gera Bytecodes ao invés de código nativo de cada máquina. Para executar um programa Java é preciso se interpretar os Bytecodes, o que é feito pelo interpretador Java. Através dos Bytecodes, Java provê um formato de arquivo objeto neutro (independente de plataforma), o que permite a sua migração entre plataformas de maneira eficiente e segura. Um programa em Java pode rodar em qualquer plataforma que tenha o interpretador Java instalado e as bibliotecas de run-time.

Características da Linguagem

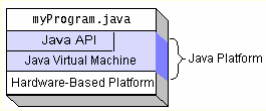
- Compilação e Interpretação no Java

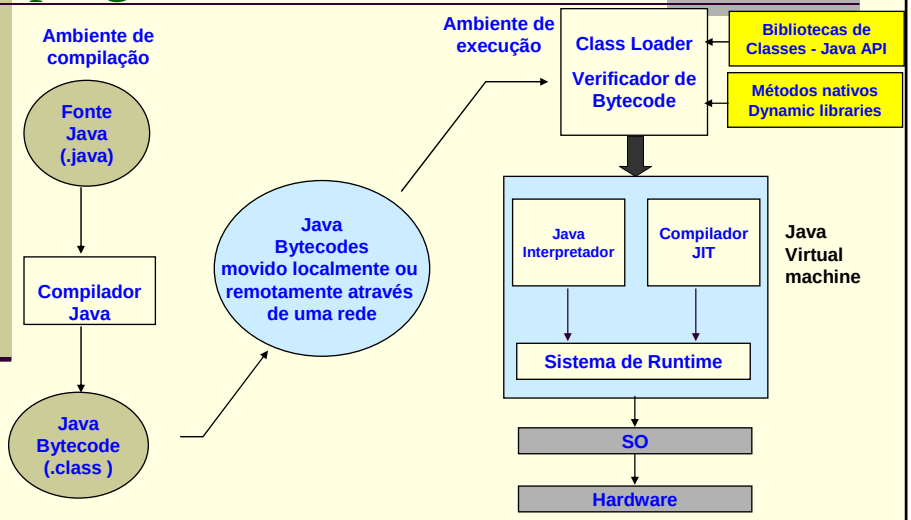


JVM – Java Virtual Machine

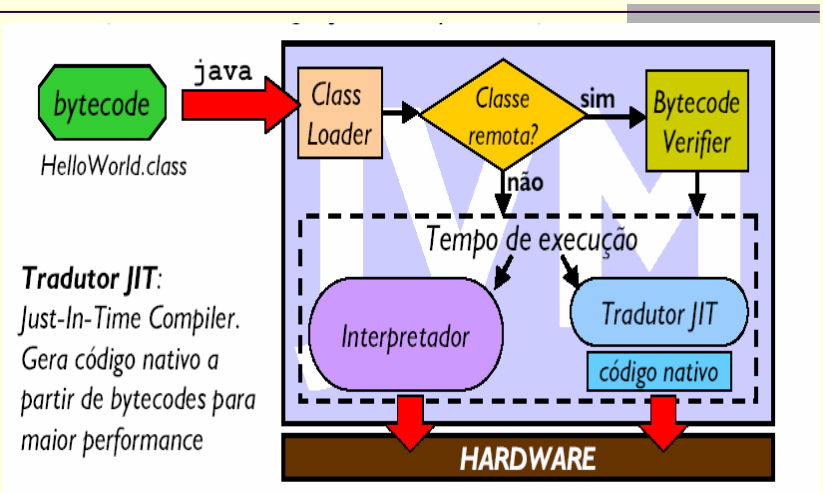
Windows 2K Macintosh Solaris Windows NT

Ciclo de vida de um programa em Java





JVM – Class Loader



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

J2SDK – Java System Devel. Kit

- *O JSDK (Java System Development Kit) é o ambiente padrão distribuído pela Sun para desenvolvimento de aplicações Java*
- *O JSDK 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*

JSDK - Ambiente de 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 Byte-code

- ***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

7

Bytecode Verifier

Segurança

Referências à memória são resolvidas pelo compilador e traduzidas durante a execução pelo interpretador. Durante a execução a maquina virtual Java (JVM) executa um processo de verificação dos Bytecodes do programa para assegurar que a classe carregada a partir da rede não tenha sido adulterada (no caso de Applets).

Verificação de Bytecodes

Etapa que antecede a execução do código em classes carregadas através da rede

- ***Class Loader distingue classes locais (seguras) de classes remotas (potencialmente inseguras)***

Bytecode Verifier

- Verificação de Bytecodes (cont)
  - 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***

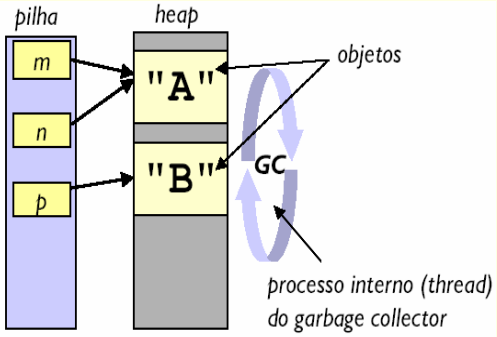
Características da Linguagem Java

- Arquitetura Neutra
  - Programas Java são compilados, conforme já vimos, para um formato neutro (independente de plataforma). A primeira vantagem é a possibilidade de executar este programa em qualquer HW que suporte um JVM. Além disso, Java define uma biblioteca gráfica padrão para GUI (java.awt–Abstract Windowing Toolkit) de forma que a aplicação terá sempre o mesmo comportamento e aparência em qualquer plataforma
- 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 dados do objeto***
    - ***2. Inicializar o objeto (via construtor)***
    - ***3. Atribuir endereço de memória a uma variável (referência)***

Características da Linguagem Java

- Coleta de Lixo (cont.)
  - Mais de uma referência pode apontar para o mesmo objeto, conforme o exemplo abaixo:

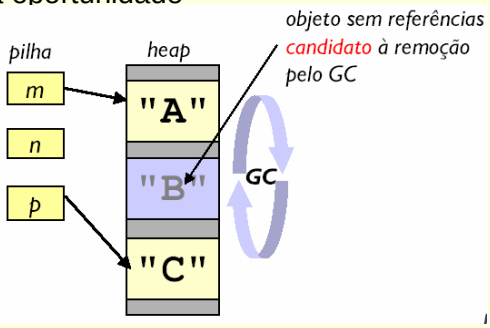
```
Mensagem m, n, p;
m = new Mensagem("A");
n = m;
p = new Mensagem("B");
```



Características da Linguagem Java

- Coleta de Lixo (cont.)
  - Quando um objeto não tem mais referências apontando para ele, seus dados não mais podem ser usados, e a memória pode ser liberada. O coletor de lixo irá liberar a memória na primeira oportunidade

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

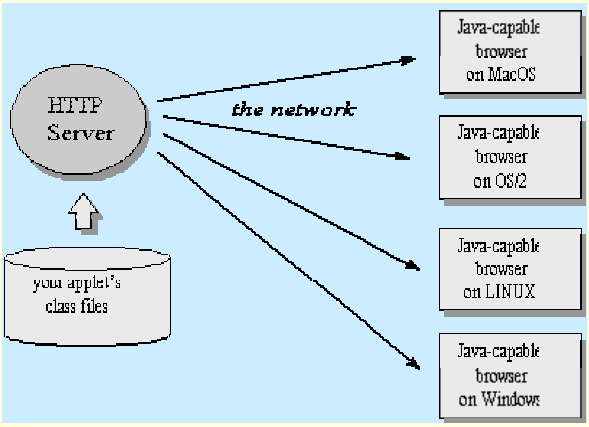


Características da Linguagem Java

- Portabilidade
  - Arquitetura Neutra é apenas uma parte do processo para se obter a portabilidade! Java se preocupa em não deixar nenhum aspecto da linguagem sendo dependente de implementação. Por isso o tamanho dos tipos de dados de Java são definidos independentes de plataforma (veremos isso adiante)
- Dinâmica
  - Java é mais dinâmica que C/C++. Bibliotecas podem livremente somar novos métodos e instâncias de variáveis sem nenhum efeito em seus clientes. Em Java descobrir o tipo de uma instância em tempo de execução é algo extremamente simples.
  - Além disso Java apresenta suporte para a mobilidade de código via rede como é o caso de applets.

Características da Linguagem Java

- Java Applet



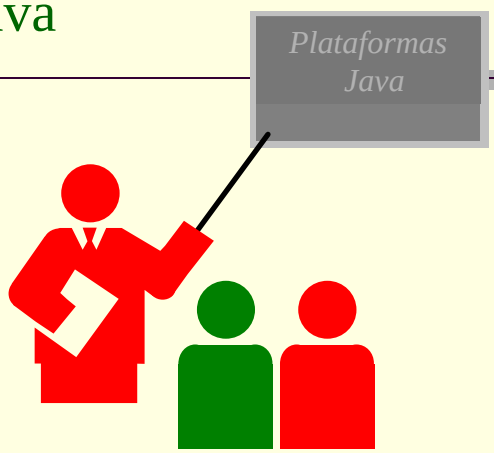
Características da Linguagem Java

- Alta Performance
  - Antigamente Java era aproximadamente 10 a 20 vezes mais lenta que o código gerado em C/C++. Atualmente após diversas pesquisas na area de compiladores com tecnologia JIT (Just In Time Compiler) já e' possível a tradução de Bytecodes direto para código de máquina da CPU durante a execução.
  - Mais recentemente a tecnologia HotSpot tem se mostrado bastante eficiente e já se consegue executar programas com performance semelhante a de programas em C ou ate mesmo Fortran. Alem disso conversores de Java para C/C++ já estão disponíveis e podem ser usados nos casos onde a necessidade de performance seja crítica.

Características da Linguagem Java

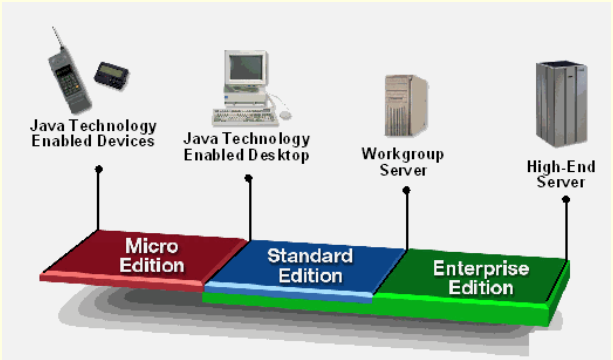
- MultiThreaded
  - Aplicações gráficas e distribuídas devem possibilitar a execução de várias tarefas de forma concorrente (Web Browser). Java é uma linguagem multithread, suportando múltiplas threads para a execução de várias tarefas em paralelo. Escrever código com múltiplas threads é extremamente difícil em C e/ou C++ já que estas linguagens não foram projetadas com este intuito.
  - O pacote java.lang define a classe Thread e inclui suporte a primitivas de sincronização de threads. Estas primitivas estão baseadas no modelo de monitor e variáveis de condição definidas por C.A.R. Hoare. Além disso, Threads em Java têm a capacidade de usar os recursos de sistemas com múltiplos processadores.

POO-Java

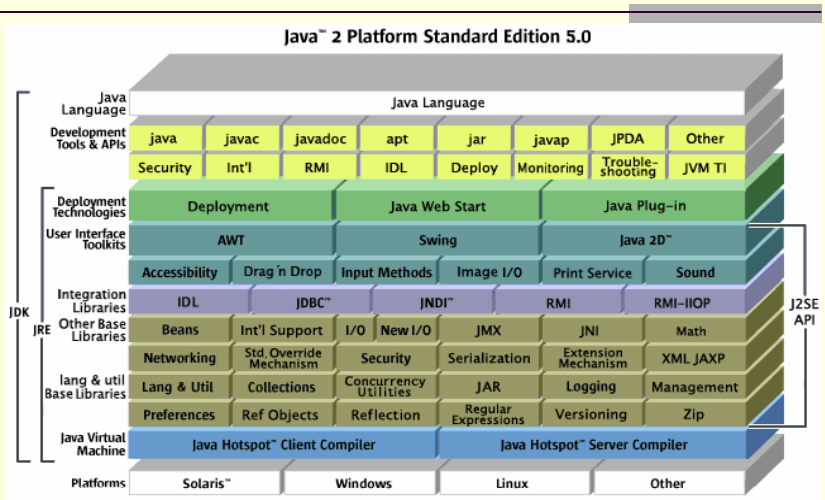


The Java Plataforms

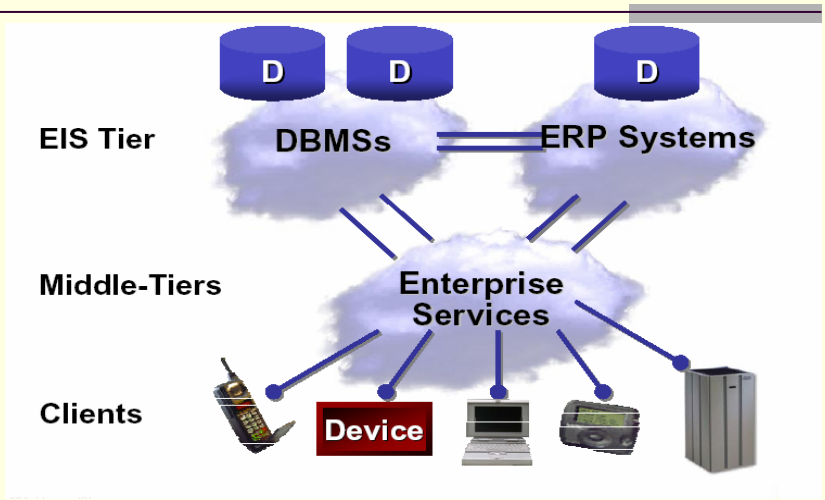
- Java 2 Plataform, Standard Edition (J2SE)
- Java 2 Plataform, Enterprise Edition (J2EE)
- Java 2 Plataform, Micro Edition (J2ME)



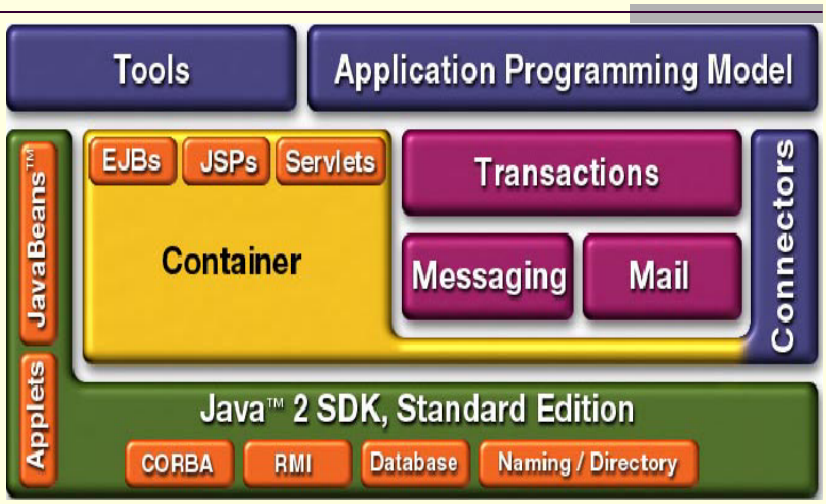
Java 2 Plataform SE



Global Enterprise



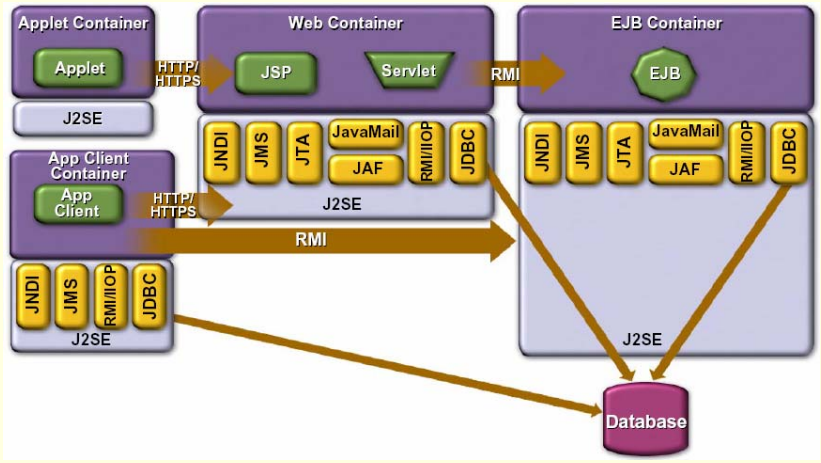
Java Plataform EE



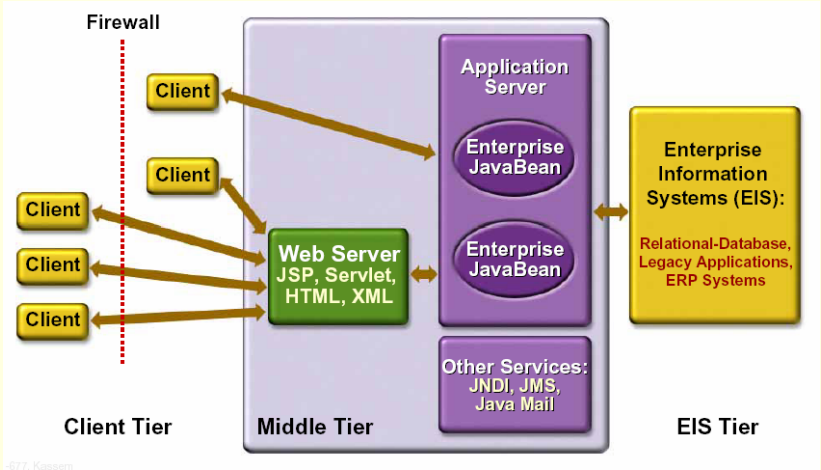
JEE Container

- Funções
  - Concorrência na execução multi-usuarios
  - Consistência, suporte a transações distribuídas
  - Segurança
  - Disponibilidade
  - Escalabilidade
  - Suporte a distribuição da aplicações
  - Integração de diversas aplicações
  - Facilidades de administração

O Ambiente JEE



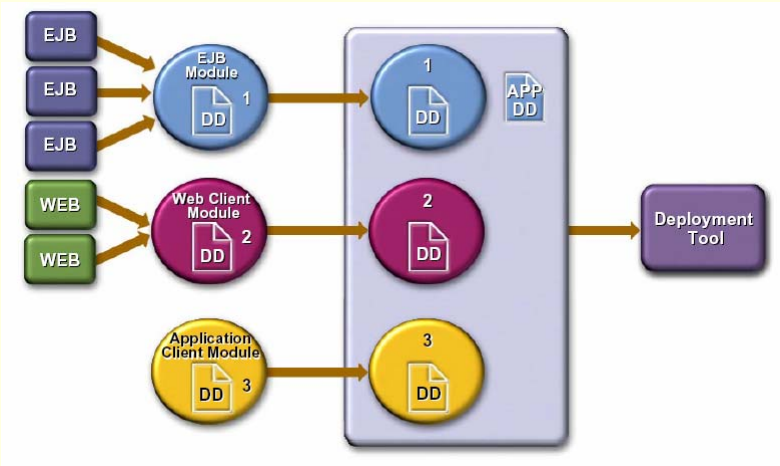
O Ambiente JEE



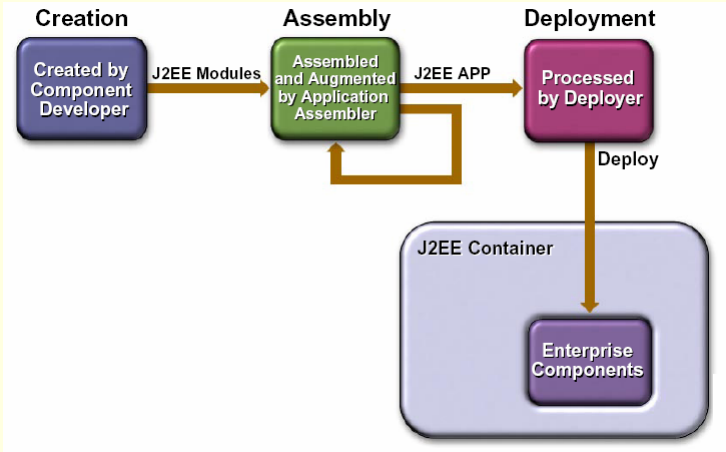
J2EE Containers x Components

- Containers
  - O container é o carro
  - Representa o suporte dado pela plataforma, realizando seu trabalho de forma transparente para o usuario.
  - Se encarregam do gerenciamento da comunicação entre as camadas de Apresentação, Negócios e Persistência.
- Components
  - O componente é o motorista
  - Aplicação do cliente é mais facilmente codificada através do uso dos componentes disponibilizados pela plataforma

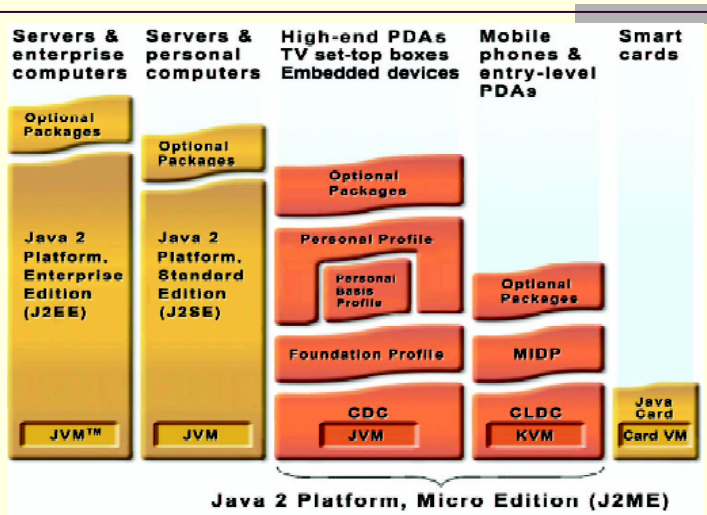
O deployment da Aplicação



Ciclo de Vida da Aplicação



Java 2 Plataforms ME



A especificação J2ME

- **O J2ME e dividido em Configurações e Perfis**
  - **Uma Configuração define uma plataforma Java para uma faixa de equipamentos. Especifica uma JVM e uma API que podem ser portadas facilmente entre uma larga faixa de dispositivos.**
    - **CLDC** (Connected Limited Device Configuration)
      - Mínimo**: 512 Kb ROM, 256 Kb RAM e algum tipo de conexão, possivelmente persistente, e com alta taxa de transmissão.**
    - **CLDC** (Connected Device Configuration)
      - Mínimo**: 128 Kb ROM, 32 Kb RAM, GUI restrita, conexão de rede wireless com baixa taxa de transmissão e acesso intermitente.**

A especificação J2ME

- **O J2ME Perfis**
  - **O Perfil é o suplemento da configuração para dispositivos específicos, contem bibliotecas de classes para a criação de aplicações para uma categoria restrita de dispositivos**
    - **MIDP 2.0** (Mobile Information Device Profile), define aspectos de seguranca, rede(HTTP, HTTPS, RS232, Sockets, Datagramas), gráficos, tecnologia Push, sons
    - **FP** (Foundation Profile), **JGP** (JavaGame Profile)
    - **PP** (Personal Profile), **PDAP** (PDA Profile)

Exemplo J2ME - HelloWorldMIDlet

```
/*
 * @(#)HelloWorldMIDlet.java - 1.0 03/03/05
 */
package example;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
/**
 * An example HelloWorldMIDlet shows the values of the system properties.
 */
public class HelloWorldMIDlet extends MIDlet implements CommandListener {
  private Display display;
  private Command exitCommand, backCommand, aboutCommand, propsCommand;
  private TextBox textBox;
  private Alert alert;
  private List list;
  private Form props;
  private StringBuffer propbuf = new StringBuffer(50);
  private boolean firstTime = true;
  /*
   * Construct a new HelloWorldMIDlet.
   */
  public HelloWorldMIDlet() {
    // Display
    display = Display.getDisplay(this);
```


Exemplo J2ME (cont.)

```
  // Sintax: Command("Name", Type, Priority)
  exitCommand  = new Command("Exit", Command.EXIT, 1);
  backCommand  = new Command("Back", Command.BACK, 1);
  aboutCommand = new Command("About", Command.SCREEN, 2);
  propsCommand = new Command("SysProps", Command.SCREEN, 2);

  // First Display shows List options
  String  stringElements[] = { "About", "System Properties", "Alert" };
  list = new List("Título", List.IMPLICIT, stringElements, null);
  list.addCommand(exitCommand);

  // Sintax: TextBox("Title", "Initial text", NrChars, Validations)
  textBox = new TextBox("HelloWorldMIDlet", "Hello World MIDlet Example ...",256,0);
  textBox.addCommand(exitCommand); textBox.addCommand(propsCommand);
  textBox.addCommand(backCommand);

  // Form info
  props = new Form("System Properties");
  props.addCommand(exitCommand); props.addCommand(aboutCommand);
  props.addCommand(backCommand);

  // Alert Message
  alert = new Alert("Hello Alert", "Testing Alert ...", null, AlertType.INFO);
}
```

Exemplo J2ME (cont.)

```
/**
   * Show the value of the properties
   */
  public void startApp() {
    Runtime runtime = Runtime.getRuntime();
    runtime.gc();
    long free = runtime.freeMemory();
    if( firstTime ) {
      long total = runtime.totalMemory();
      props.append("Free Memory = " + free + "\n");
      props.append("Total Memory = " + total + "\n");
      props.append(showProp("microedition.configuration"));
      props.append(showProp("microedition.profiles"));
      props.append(showProp("microedition.platform"));
      props.append(showProp("microedition.locale"));
      props.append(showProp("microedition.encoding"));

      firstTime = false;
      list.setCommandListener(this);
      display.setCurrent(list);
    } else {
      props.set(0, new StringItem("", "Free Memory = " + free + "\n"));
    }
  }
```


Exemplo J2ME (cont.)

```
/**
   * Handle events
   */
  public void commandAction(Command c, Displayable s) {
    if ( c == exitCommand ) {
      destroyApp(false); notifyDestroyed();
    } else if ( c == backCommand ) {
      list.setCommandListener(this); display.setCurrent(list);
    } else if ( c == aboutCommand ) {
      textBox.setCommandListener(this); display.setCurrent(textBox);
    } else if ( c == propsCommand ) {
      props.setCommandListener(this); display.setCurrent(props);
    } else {
      if ( list.getSelectedIndex() == 0 ) {
        textBox.setCommandListener(this); display.setCurrent(textBox);
      } else if ( list.getSelectedIndex() == 1 ) {
        props.setCommandListener(this); display.setCurrent(props);
      } else {
        display.setCurrent(alert);
      }
    }
  }
```

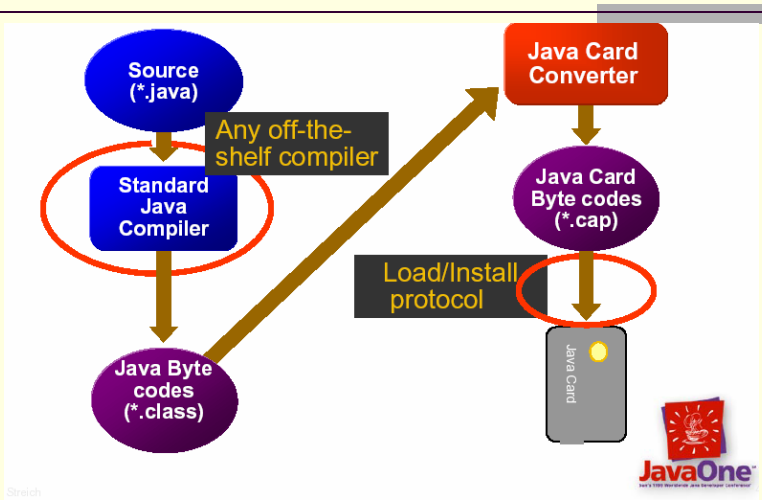
Exemplo J2ME (cont.)

```
/**
   * Show a property.
   */
  String showProp(String prop) {
    String value = System.getProperty(prop);
    propbuf.setLength(0);
    propbuf.append(prop); propbuf.append(" = ");
    if (value == null) {
      propbuf.append("<undefined>");
    } else {
      propbuf.append("\""); propbuf.append(value); propbuf.append("\"");
    }
    propbuf.append("\n");
    return propbuf.toString();
  }
  /**
   * Time to pause, free any space we don't need right now.
   */
  public void pauseApp() {
  }
  /**
   * Destroy must cleanup everything.
   */
  public void destroyApp(boolean unconditional) {
  }
}
```


Tecnologia JavaCard

- Disponibiliza Java para “smart cards”
- JavaCard define um subset da linguagem Java e da Java Virtual Machine para executar em “smart cards”
  - OOP para “smart cards” mais simples e poderosa do que programação em C
  - Modelo Web browser de funcionamento
  - Core and extensions JavaCard API
  - JavaCard Runtime Environment (JCRE)

Desenvolvendo aplicação JavaCard



POO-Java

Introdução Prática

Introdução Prática

- Programa Exemplo
  - Crie o programa “AloMundo.java” (case-sensitive)

```
public class AloMundo {
 public static void main(String[] args){
   System.out.println(“Alô Mundo!”);
 }
}
```

Exercício - Questão 1

Introdução Prática - Applet

- Applet Exemplo – “AloMundoApplet.java”
  - Applets são programas que podem ser executados em Browsers compatíveis com a linguagem Java

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorld extends Applet {
   public void paint(Graphics g) {
     g.drawString("Hello world!", 50, 25);
   }
}
```

Exercício - Questão 2

Como Compilar

- *Use o **java** compiler (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*

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`

Como Executar

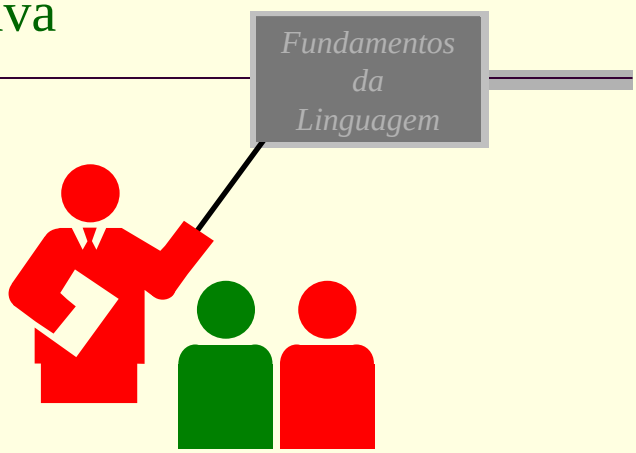
- *Use o interpretador **java** (faz parte do JRE)**
  - `java NomeDaClasse`
  - `java pacote.subpacote.NomeDaClasse`
  - `java -classpath c:\classes;c:\bin;. pacote.Classe`
  - `java -cp c:\classes;c:\bin;. pacote.Classe`
  - `java -cp %CLASSPATH%;c:\mais pacote.Classe`
  - `java -cp biblioteca.jar 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*

Outras ferramentas do JSDK

- *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*

POO-Java



Tipos Abstratos de Dados

- Modela uma estrutura de dados através de sua funcionalidade.
- Define a interface de acesso à estrutura.
- Não faz qualquer consideração com relação à implementação.

Exemplo de TAD: Pilha

- Funcionalidade: armazenagem LIFO
- Interface:

```
boolean isEmpty()
```
verifica se a pilha está vazia

```
push(int n)
```
empilha o número fornecido

```
int pop()
```
desempilha o número do topo e o retorna

```
int top()
```
retorna o número do topo

TAD × Classes

- Uma determinada implementação de um TAD pode ser realizada por meio de uma classe.
- A classe deve prover todos os métodos definidos na interface do TAD.
- Um objeto dessa classe implementa uma instância do TAD.

Fundamentos da Linguagem

- Classes e Objetos
  - Em Java, a declaração de novas classes é feita através da construção class.
  - Podemos criar uma classe Point para representar um ponto (omitindo sua implementação) da seguinte forma:

```
class Point {
  ...
}
```


Fundamentos da Linguagem

- Campos - Atributos ou Métodos
  - Como dito, classes definem dados que suas instâncias conterão.
  - A classe Point precisa armazenar as coordenadas do ponto sendo representado de alguma forma.

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

Fundamentos da Linguagem

Instanciação ou Criação

- Uma vez definida uma classe, uma nova instância (objeto) pode ser criada através do comando new.
- Podemos criar uma instância da classe Point da seguinte forma:

```
Point p = new Point();
```


Fundamentos da Linguagem

Referências para Objetos

- Em Java, nós sempre fazemos referência ao objeto. Dessa forma, duas variáveis podem se referenciar ao mesmo ponto.

```
Point p1 = new Point();
Point p2 = p1;
p2.x = 2;
p2.y = 3;
// p1 e p2 representam o ponto (2,3)
```

Fundamentos da Linguagem

Acessando Campos

- Os campos de uma instância de Point podem ser manipulados diretamente.

```
Point p1 = new Point();
p1.x = 1; p1.y = 2;
// p1 representa o ponto (1,2)
Point p2 = new Point();
p2.x = 0; p2.y = 0;
// e p2 o ponto (0,0)
```


Fundamentos da Linguagem

Métodos

- Além de atributos, uma classe deve definir os métodos que irá disponibilizar, isto é, a sua interface.

- A classe Point pode, por exemplo, prover um método para mover o ponto de um dado deslocamento.

Fundamentos da Linguagem

- Declaração de Método

  - Para mover um ponto, precisamos saber quanto deslocar em x e em y. Esse método não tem um valor de retorno pois seu efeito é mudar o estado do objeto.

```
class Point {
  int x, y;
  void move(int dx, int dy) {
    x += dx; y += dy;
  }
}
```


Fundamentos da Linguagem

- Envio de Mensagens: Chamadas de Método

  - Em Java, o envio de uma mensagem é feito através de uma chamada de método com passagem de parâmetros.

  - Por exemplo, a mensagem que dispara a ação de deslocar um ponto é a chamada de seu método move.

```
p1.move(2,2);
// p1 está deslocado de duas unidades
// no sentido positivo, nos dois eixos.
```

Fundamentos da Linguagem

- this
  - Dentro de um método, o objeto pode precisar de sua própria referência.
  - Em Java, a palavra reservada this significa essa referência ao próprio objeto.

```
class Point {
  int x, y;
  void move(int dx, int dy) {
    this.x += dx; this.y += dy;
  }
}
```


Fundamentos da Linguagem

- Inicializações
  - Em várias circunstâncias, é interessante inicializar um objeto.
  - Por exemplo, poderíamos querer que todo ponto recém criado estivesse em (0,0).
  - Esse tipo de inicialização se resume a determinar valores iniciais para os campos

Fundamentos da Linguagem

Inicialização de Campos

- Por exemplo, a classe Point poderia declarar:

```
class Point {
  int x = 0;    int y = 0;
  void move(int dx, int dy) {
    this.x += dx;
    this.y += dy;
  }
}
```


Fundamentos da Linguagem

Construtores

- Ao invés de criar pontos sempre em (0,0), poderíamos querer especificar a posição do ponto no momento de sua criação.
- O uso de construtores permite isso.
- Construtores são mais genéricos do que simples atribuições de valores iniciais aos campos: podem receber parâmetros e fazer um processamento qualquer.

Fundamentos da Linguagem

Declaração de Construtores

- O construtor citado para a classe Point pode ser definido da seguinte forma:

```
class Point {
  int x = 0;
  int y = 0;
  Point(int x, int y) {
    this.x = x; this.y = y;
  }
  ...
}
```


Fundamentos da Linguagem

Usando Construtores

- Como o construtor é um método de inicialização do objeto, devemos utilizá-lo no momento da instanciação.

```
Point p1 = new Point(1,2);
// p1 é o   ponto (1,2)
Point p2 = new Point(0,0);
// p2 é o ponto (0,0)
```

Fundamentos da Linguagem

Construtor Padrão

- Quando não especificamos nenhum construtor, a linguagem Java declara, implicitamente, um construtor padrão, vazio, que não recebe parâmetros.
- Se declararmos algum construtor, esse construtor padrão não será mais declarado.

Fundamentos da Linguagem

Finalizações

- Pode ser necessário executar alguma ação antes que um objeto deixe de existir.
- Para isso são utilizados os destrutores.
- Destrutores são métodos que são chamados automaticamente quando um objeto deixa de existir.
- Em Java, destrutores são chamados de finalizadores.

Fundamentos da Linguagem

Gerência de Memória – Garbage Collection

- Java possui uma gerência automática de memória, isto é, quando um objeto não é mais referenciado pelo programa, ele é automaticamente coletado (destruído).
- A esse processo chamamos "coleta de lixo".
- Nem todas as linguagens OO fazem coleta de lixo e, nesse caso, o programador deve destruir o objeto explicitamente.

Fundamentos da Linguagem

Finalizadores em Java

- Quando um objeto Java vai ser coletado, ele tem seu método finalize chamado.
- Esse método deve efetuar qualquer procedimento de finalização que seja necessário antes da coleta do objeto.

Fundamentos da Linguagem

Membros de Classe

- Classes podem declarar membros (campos e métodos) que sejam comuns a todas as instâncias, ou seja, membros compartilhados por todos os objetos da classe.
- Tais membros são comumente chamados de ‘membros de classe’ (versus ‘de objetos’).
- Em Java, declaramos um membro de classe usando o qualificador static. Daí, o nome ‘membros estáticos’ usado em Java.

Fundamentos da Linguagem

Membros de Classe: Motivação

- Considere uma classe que precise atribuir identificadores unívocos para cada objeto.
- Cada objeto, ao ser criado, recebe o seu identificador.
- O identificador pode ser um número gerado seqüencialmente, de tal forma que cada objeto guarde o seu mas o próximo número a ser usado deve ser armazenado na classe.

Fundamentos da Linguagem

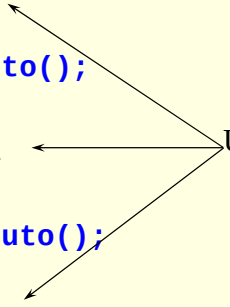
- Membros de Classe: Um Exemplo
  - Podemos criar uma classe que modele produtos que são produzidos em uma fábrica.
  - Cada produto deve ter um código único de identificação.

```
class Produto {
  static int próximo_id = 0;
  int id;
  Produto() {
    id = próximo_id;  próximo_id++;
  }  ...
}
```


Fundamentos da Linguagem

- Membros de Classe: Análise do Exemplo

```
// Considere que ainda não há nenhum produto.
// Produto.próximo_id = 0

Produto lápis = new Produto();
  // lápis.id = 0
  // lápis.próximo_id = 1

Produto caneta = new Produto();
  // caneta.id = 1
  // caneta.próximo_id = 2
```



Fundamentos da Linguagem

Membros de Classe: Acesso Direto

- Como os membros estáticos são da classe, não precisamos de um objeto para acessá-los: podemos fazê-lo diretamente sobre a classe.

```
Produto.próximo_id = 200;
// O próximo produto criado terá
// id = 200.
```

- Java, sendo uma linguagem puramente orientada por objetos, possui apenas declarações de classes: não é possível escrever código fora de uma classe.

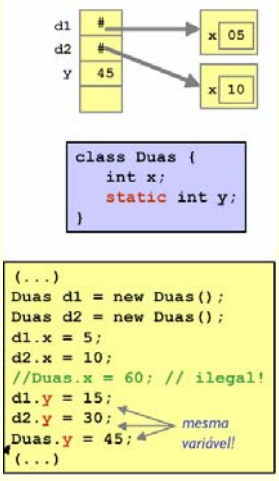
Fundamentos da Linguagem

Membros de Classe

- Para prover uma biblioteca matemática, Java declara uma classe, Math, que contém apenas métodos estáticos.
- Exemplo1: calcular a distância entre dois pontos.

```
float dx = p1.x - p2.x;
float dy = p1.y - p2.y;
float d = Math.sqrt(dx*dx+dy*dy);
```

- Exemplo2:
  - observe na figura ao lado que a a variavel de classe y não esta presente em nenhum objeto !



Fundamentos da Linguagem

- this revisitado
  - Vimos que um método estático pode ser chamado diretamente sobre a classe. Ou seja, não é necessário que haja uma instância para chamarmos um método estático. Dessa forma, não faz sentido que o this exista dentro de um método estático.

Fundamentos da Linguagem

- Noção de Programa
  - Uma vez que tudo o que se escreve em Java são declarações de classes, o conceito de programa também está relacionado a classes: a execução de um programa é, na verdade, a execução de uma classe.
  - Executar uma classe significa executar seu método estático main. Para ser executado, o método main deve possuir uma assinatura bem determinada.

Fundamentos da Linguagem

- Executando uma Classe
  - Para que a classe possa ser executada, seu método main deve possuir a seguinte assinatura:

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

- Estrutura de um Programa Java

Unidade de compilação (arquivo .java)
Declarações do pacote
Declarações de importação
Interfaces
Classes

Fundamentos da Linguagem

- Programa Exemplo – AloMundo.java
  - Usando o método main e um atributo estático da classe que modela o sistema, podemos escrever nosso primeiro programa:

```
public class AloMundo {
 public static void main(String[] args){
   System.out.println("Alô Mundo!");
 }
}
```

  - Uma classe deve ser declarada em um arquivo homônimo (case-sensitive) com extensão .java.

Exercício - Questão 1

Fundamentos da Linguagem

- Léxico
  - Suporte a Unicode
  - Comentários em uma linha
    // ...
  - Comentários em múltiplas linhas
    /*
    ...
    */
  - Comentários de documentação (javadoc)
    /** ... */

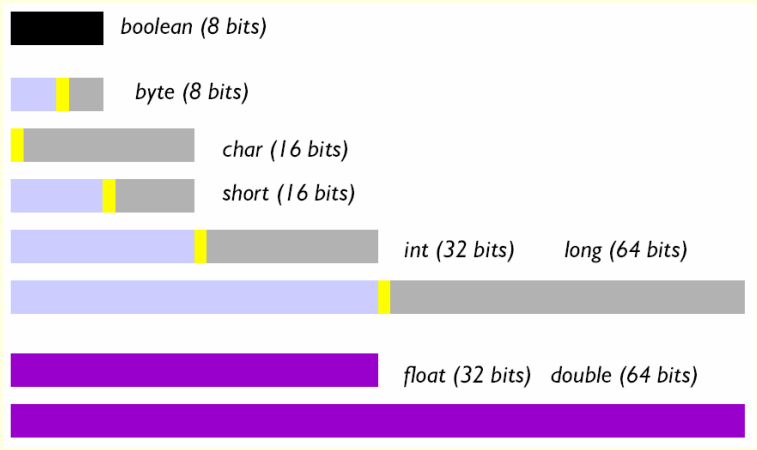
Fundamentos da Linguagem

- Tipos Básicos ou Primitivos de Java

Type	Conteúdo	Valor Default	Tamanho	Menor Valor	Maior Valor
boolean	True ou False	False	1 bit	-	-
char	Unicode caracter	\u0000	16 bits	\u0000	\uFFFF
byte	Signed integer	0	8 bits	-128	127
short	Signed integer	0	16 bits	-32768	32767
int	Signed integer	0	32 bits	-2147483648	2147483647
long	Signed integer	0	64 bits	-2^{63}	$2^{63} - 1$
float	IEEE75A floating point	0.0	32 bits	3.40×10^{38}	1.40×10^{-45}
double	IEEE75A floating point	0.0	64 bits	1.79×10^{308}	4.94×10^{-324}

Fundamentos da Linguagem

- Formato dos Tipos de Dados



Fundamentos da Linguagem

- Tipos de Dados de Referência – Reference Data Types

Todos os tipos não primitivos são tratados como objetos ou arrays, e, por isso, são acessados pelo Java por "referência".

Por isso chamamos os tipos de dados de Java não primitivos de "Reference Data Types".

Fundamentos da Linguagem

- Exemplo

```
Button a,b;
a=new Button(); //a refere-se a um objeto Button
b=a;           // b refere-se ao mesmo objeto que a
b.setLabel("Ok");
String s = b.getLabel(); // s será igual a "Ok"
```

Observe que isso não acontece quando utilizamos os tipos primitivos:

```
int i = 3;  int j = i; i = 2;
System.out.println("i="+i+"-"+"j="+j); //Exibe: i=3-j=2
```


Fundamentos da Linguagem

- A classe String
  - Strings são objetos da classe java.lang.String
  - Apesar de String não ser um tipo primitivo, ele pode ser visto como um, pois pode ser construído diretamente, sem o uso de *new*. *Strings são imutáveis !*

    ```
    String s1 = "Sou uma string";
    ```
  - A forma normal de criação de objetos também é permitida para strings (embora raramente usada):

    ```
    String s2 = new String("Também sou uma string");
    ```

Fundamentos da Linguagem

- A classe String (cont.)
  - O operador + pode ser utilizado para concatenar strings.

```
String result = s1 + s2;
```

  - Em Java, a expressão s1 == s2 testa se s1 e s2 são o mesmo objeto e não se eles contem os mesmos strings !

- String principais métodos:
  - `public char charAt(int index)`
  - `public int compareTo(String comparison)`
  - `public int compareTo(Object object)`
  - `public boolean endsWith(String suffix)`

Fundamentos da Linguagem

- String principais métodos (cont.)
  - `public String concat(String suffix)`

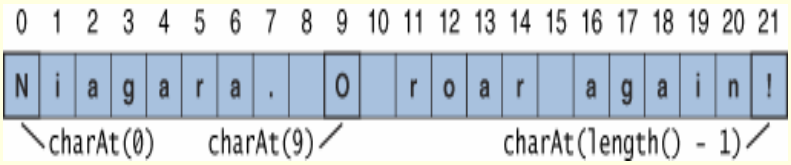
```
String result = someString.concat(otherString);
```

  - `public boolean equalsIgnoreCase(String comparison)`
  - `public int indexOf(int character)`
  - `public int indexOf(int character, int startIndex)`
  - `public int indexOf(String subString)`
  - `public int indexOf(String subString, int startIndex)`
  - `public int length()`
  - `public String replace(char oldChar, char newChar)`

Fundamentos da Linguagem

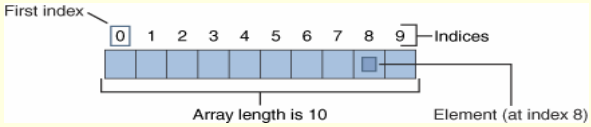
- Exemplo

```
public class StringsDemo {
  public static void main(String[] args) {
    String palindrome = "Niagara. O roar again";
    int len = palindrome.length();
    StringBuffer dest = new StringBuffer(len);
    for (int i=len-1; i >= 0; i--)
      dest.append(palindrome.charAt(i));
    System.out.println(dest.toString());
  }
}
```



Fundamentos da Linguagem

- Arrays
  - Seu comprimento pode ser determinado pelo método length
  - Pode ser associados a variáveis do tipo Object, assim como a variáveis de seu tipo específico.
  - Arrays podem ser passados para métodos.
  - Indexação de seus elementos começa por zero.
- A declaração de um array de um determinado tipo é indicada pelo uso de [ ] para cada dimensão do array.
- É necessária também a indicação do tamanho do array para a sua utilização.



Fundamentos da Linguagem

- Exemplos: Declaração de Arrays

```
int[ ] arrayDeInts;           // 1-dim
char[ ][ ] matrixDeChars;     // 2-dim
Object[ ] matrixDeObjetos[ ]; // 2-dim
UmaClasseX[ ] arrayDeUmaClasseX;
```

- Alocação de memória

```
arrayDeInts = new int [7];
matrixDeChars = new char[3][4];
```

- Declaração e Alocação

```
boolean[] answers = { true, false, true, true };
String[][] nomes = { {“Maria”, “Jose”, “Joao”},
                     { “Lucia”,“Arthur”, “Carol”} };
```


Fundamentos da Linguagem

- O atributo **length**

```
Carro[] carros = new Carro[3]; // array de 3 carros
carros[0] = new Carro(“BMW”, 60000);
carros[1] = new Carro(“Vectra”, 40000);
carros[2] = new Carro(“Gol”, 20000);

int[] precos = new int[3];

for (int i = 0; i < carros.length; i ++)  {
  precos[i] = carros[i].getPreco(); // copia os precos
  System.out.println(“Preço: ” + precos[i]);
}
```

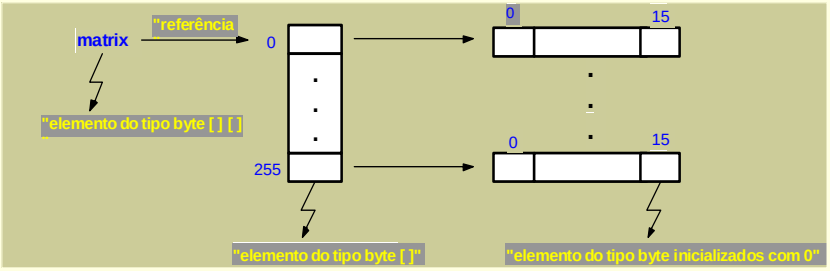
Outro exemplo:

```
int[] list = {1, 2, 3, 4, 5};
for( int i=0; i<list.length; ++i )
  System.out.println(“list[“+i+”]=”+list[i]);
```

Fundamentos da Linguagem

- Arrays Multidimensionais
  - Array multidimensionais em Java são tratados como arrays de arrays exatamente como em C/C++

byte matrix[ ][ ] = new byte [256][16];



Fundamentos da Linguagem

- Arrays Multidimensionais – Exemplos

```
String texto[][][][] = new String [5][3][][]; // é válida
double temperatura[][][] = new double [100][][10] // não !

String parametros[ ][ ] = {
            { “foreground”, “color”, “block” },
            { “background”, “color”, “gray” },
            { “Mensagem”, “String”, “Alô mamãe” } };
```

Outro exemplo:

```
short triangular[][] = new short [10][];
      for ( int i = 0; i < triangular.length; ++i ) {
            triangular[i] = new short [i + 1];
            for ( int j = 0; j < i + 1; ++j )
                  triangular[i][j] = i + j;
      }
```

Fundamentos da Linguagem

- Arrays são limitados pelo fato de não poderem variar de tamanho com o tempo => Vector array de tamanho variável.

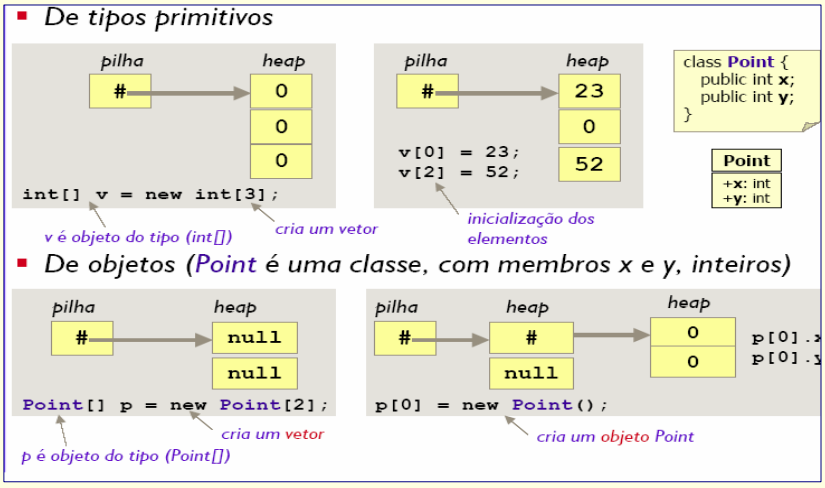
```
public Vector()                    // Construtores
public Vector(int initialCapacity)
public Vector(int initialCapacity, int capacityIncrement)
```

  - Outros métodos:

```
public void setSize(int newSize)
public boolean removeElement(Object object)
public void removeElementAt(int index)
public void removeAllElements()
public boolean isEmpty()
public boolean contains(Object object)
public int capacity()
public int size()
```


Arrays

- *De tipos primitivos*



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

Fundamentos da Linguagem

- Operadores

```
[]  .  (params)  exp++  exp--
++exp  --exp  +exp  -exp  ~exp
new  (tipo)exp
*  /  %
+  -
<<  >>  >>>
<  >  <=  >=  instanceof
==  !=
&
^
|
&&
||
?:
=  +=  -=  *=  /=  %=  <<=  >>=  >>>=  &=  ^=  |=
```

Acesso	. []
Tipagem	(tipo) instanceof
Aritméticos	++ -- * / % + -
Bits	~ << >> >>> & ^ |
Relacionais	< > <= >= == !=
Lógicos	&& ||
Atribuição	= binop= (+= *= &= ...)

Fundamentos da Linguagem

- Lista de Operadores

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

Fundamentos da Linguagem

- 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--`

Fundamentos da Linguagem

- Lista de Caracteres

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.

Fundamentos da Linguagem

- Expressões
  - Formadas por dados, operadores e parentesis
    - x * (y - (a + b) >> 5) & m[i].k * f(4)
  - Não existe aritmética de ponteiros, pois não existem ponteiros como em C/C++
    - *(mat + lin*tam_linha+col)=0 // Não em Java
    - mat[lin][col]=0 // Ok

Fundamentos da Linguagem

- Conversões em expressões
  - Java utiliza tipagem forte, assim conversões em expressões podem ser:
    - automáticas: implícita ou por promoção (conversão p/ tipo mais geral)
    - explícitas (coerção ou type-cast)
- Exemplos:

```
byte b  = 10;   // Conversão automática (implícita)
float f = 0.0F; int i;
```

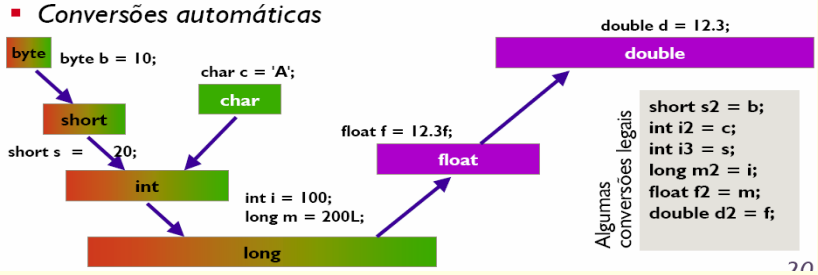
Fundamentos da Linguagem

- Exemplos (cont.)

```
b = i; // O Compilador vai indicar o erro: “Imcompatible
       // type for=. Explicit cast needed to convert int”

double d = 12 + 9L + 12.3; // promoção para double dos numeros

b = (byte)i;      // explícita (coerção) ou “type cast”
s = i.toString(); // explícita (coerção) por métodos

f = i; f = b; // Estas instruções apresentam problema ?
```


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*



Conversão de Tipos Referência

- *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();
```


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!

Fundamentos da Linguagem

- Controle de Fluxo 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, return e rótulos***
  - **Não existe comando goto**
    - ***goto, porém, é palavra-reservada.***

Fundamentos da Linguagem

- Comandos

Comando

- **expressão de atribuição**
- **formas pré-fixadas ou pós-fixadas de ++ e --**
- **chamada de métodos**
- **criação de objetos**
- **comandos de controle de fluxo**
- **bloco**

Bloco = { <lista de comandos> }

Fundamentos da Linguagem

- if-else

```
if( a>0 && b>0 )
  m = média(a, b);
else {
  errno = -1;
  m = 0;
}
```

```
if (expressão booleana)
   instrução_simples;

if (expressão booleana) {
   instruções
}
```

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

Fundamentos da Linguagem

- switch-case-default

```
switch( i=f() ) {
  case -1:
    ...
    break;
  case 0:
    ...
    break;
  default:
    ...
}
```

- *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)

Fundamentos da Linguagem

- while

```
int i = 0;
while( ++i<10 ) {
 System.out.println(i);
}
//Que números serão impressos?
//E se trocarmos por i++<10
```

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

- do while

```
int i = 0;
do {
  System.out.println(i);
} while( ++i<10);
//Que números serão impressos?
//E se trocarmos por i++<10
```

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

Fundamentos da Linguagem

- for

```
for(int i=0; i<10; ++i)
 System.out.println(i);
//Que números serão impressos?
//E se trocarmos por i++
```

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

- return

```
int média(int a, int b) {
  return (a+b)/2;
}
```

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


Fundamentos da Linguagem

- break

```
int i = 0;
while( true ) {
  if (++i==10) break;
  System.out.println(i);
}
//Que números serão impressos?
//E se trocarmos por i++
```

- continue

```
int i = 0;
while ( true ) {
  if (++i%2 == 1) continue;
  System.out.println(i);
}
//Que números serão impressos?
//E se trocarmos por i++
```

```
while (!terminado) {
  passePagina();
  if (alguemChamou == true) {
    break;       // caia fora deste loop
  }
  if (paginaDePropaganda == true) {
    continue;    // pule esta iteração
  }
  leia();
}
restoDoPrograma();
```

Fundamentos da Linguagem

- break para Label

```
início:
for (int i=0; i<10; i++)
  for (int j=0; j<10; j++)
  {
    if (v[i][j] < 0)
      break início;
    ...
  }
...
```

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

break sem rótulo quebraria aqui!

Exercícios - Questão 3

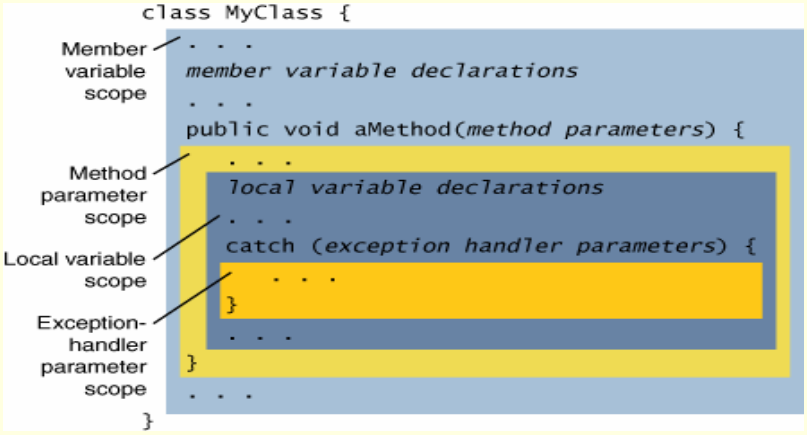
Fundamentos da Linguagem

- Classes
  - Classes definem modelos de objetos
  - Objetos = Dados + Código

```
class Empregado {              // Define Classe
     String nome;              // Atributo
     int h_total;              // Atributo
     void Trabalha (int h) {   // Método
           h_total += h;
     }
}
```

Fundamentos da Linguagem

- **Escopo de definições**



Fundamentos da Linguagem

- Inicialização de Classes
  - Bloco de inicialização de classe identificado pela palavra reservada static
  - Atributos podem ter atribuição default

```
class Nome {
  String Nome = "";  // Inicialização de atributo
  static int data[1000];
  static {      // Bloco inicialização de classe
   for (int i=0; i<1000; ++i)
         data[i] = i;
  }
}
```

Fundamentos da Linguagem

- Objetos
  - Instâncias de classes

```
class Empregado { ... }  // Define classe
Empregado Roberto;       // Declara objeto
Roberto = new Empregado; // Instancia objeto
```

  - Quando o objeto é declarado este recebe o valor nulo (null)
  - Após new o objeto é efetivamente criado em memória (heap) usando o construtor da classe

Fundamentos da Linguagem

- Acessando Atributos e Métodos
  - Operador de acesso aos membros de um objeto (.)

```
Roberto.h_total = 0;      // Atributo
Roberto.Trabalha(8);      // Método
```

- Inicialização de Objetos
  - Inicialização de objeto é realizada pelo método construtor da classe
  - O construtor é invocado automaticamente quando o comando new é executado
  - O construtor é um método que não possui identificação de valor de retorno e possui o mesmo nome que a 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");
```

Chamada de construtor

Fundamentos da Linguagem

- Inicialização de Objetos

```
class Ônibus {
    int lotação;
    Ônibus(int LotaçãoMáxima) {
          lotação = LotaçãoMáxima;
    }
}
...
Ônibus o;                 // Declara objeto
o = new Ônibus(32);       // Instancia objeto
        ou
Ônibus o2 = new Ônibus(40);
```

Fundamentos da Linguagem

Destruição de Objetos

- O destrutor de um objeto é invocado pelo coletor de lixo quando este precisa desfazer-se do objeto para liberar memória. Não se pode dizer quando o destrutor será invocado
- O destrutor é um método especial da classe que não retorna valor (void), possui o nome de finalize e não possui parâmetros

```
class Ônibus {
  void finalize() { ProFerroVelho(); }
}
```

- O método finalize() também pode ser executado explicitamente se desejado: `meuÔnibus.finalize();`
- Obs: finalize() não tem garantia de funcionar quando desejado, por isso é melhor usar o código dentro de um bloco try { … } catch(…) { …. } finally { ….. }

Fundamentos da Linguagem

Exemplo Classe Pilha

- Classe pilha de números inteiros, armazenados internamente em um array cujo tamanho máximo é dado no momento de sua criação.

```
class Stack {
  int top_index; int[] data;  // <- Variáveis
  Stack(int size) {           // <- Construtor
   data = new int[size];
   top_index = -1;
  }
  boolean isEmpty() { return (top_index < 0); }
  void push(int n) { data[++top_index] = n; }
  int pop() { return data[top_index--]; } // <- Métodos
  int top() { return data[top_index]; }
}
```

Fundamentos da Linguagem

- Uso da classe Pilha declarada

```
Stack s = new Stack(2); // Pilha para 2 números.
s.push(10);
s.push(20);
s.push(s.pop()+s.pop());
System.out.println(s.top());     // 30
System.out.println(s.isEmpty()); // false
s.pop();
System.out.println(s.isEmpty()); // true
```


Fundamentos da Linguagem

- Encapsulamento
  - Na classe Stack implementada, nós encapsulamos a definição de pilha que desejávamos, porém, por falta de controle de acesso, é possível forçar situações nas quais a pilha não se comporta como desejado.

```
Stack s = new Stack(10);
s.push(6);
s.top_index = -1;
System.out.println( s.isEmpty() );  // true!
```

Fundamentos da Linguagem

- Encapsulamento - Controle de Acesso
  - As linguagens OO disponibilizam formas de controlar o acesso aos membros de uma classe. No mínimo, devemos poder fazer diferença entre o que é público e o que é privado.
  - Membros públicos podem ser acessados indiscriminadamente, enquanto os privados só podem ser acessados pela própria classe.

Fundamentos da Linguagem

- Redefinindo a Classe Stack

```
class Stack {
  private int[] data;
  private int top_index;
  Stack(int size) {
    data = new int[size];
    top_index = -1;
  }
  boolean isEmpty() { return (top_index < 0); }
  void push(int n) { data[++top_index] = n; }
  int pop() { return data[top_index--]; }
  int top() { return data[top_index]; }
}
```

Exemplo - Calculadora Polonesa

Fundamentos da Linguagem

- Exemplo de Controle de Acesso
  - Com a nova implementação da pilha, o exemplo anterior não pode mais ser feito pois teremos um erro de compilação.

```
Stack s = new Stack(10);
s.push(6);
s.top_index = -1;    // ERRO! na compilação
System.out.println(s.isEmpty());
```


Fundamentos da Linguagem

- Sobrecarga (Overload)
  - Um recurso usual em programação OO é o uso de sobrecarga de métodos.
  - Sobrecarregar um método significa prover mais de uma versão de um mesmo método.
  - As versões devem, necessariamente, possuir algo que as diferencie: tipo e/ou número de parâmetros ou tipo do valor de retorno.

Fundamentos da Linguagem

- Sobrecarga (Overload)
  - 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)

Fundamentos da Linguagem

- Sobrecarga de Construtores
  - Como dito anteriormente, ao criarmos o construtor da classe Point para inicializar o ponto em uma dada posição, perdemos o construtor padrão que, não fazendo nada, deixava o ponto na posição (0,0).
  - Nós podemos voltar a ter esse construtor usando sobrecarga.

```
class Point {
  int x = 0;  int y = 0;
  Point() {}
  Point(int x, int y) {
    this.x = x; this.y = y;
  }
}
```

Agora no momento da criação do objeto podemos escolher qual construtor usar

```
Point p1 = new Point();   //p1 está em (0,0)
Point p2 = new Point(1,2);//p2 está em (1,2)
```

Encadeamento de Construtores

- Uma solução melhor para o exemplo dos dois construtores seria o construtor vazio chamar o construtor que espera suas coordenadas, passando zero para ambas.
- Isso é um *encadeamento* de construtores.
- Java suporta isso através da construção this(...). A única limitação é que essa chamada seja a primeira linha do construtor.

Fundamentos da Linguagem

- Encadeamento de Construtores

```
class Point {
  int x, y;
  Point() {
    this(0,0);  // <- Chama construtor com parametros
  }
  Point(int x, int y) {
    this.x = x; this.y = y;
  }
  ...
}
```

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";
      }
   }
}
```

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.*

Sobrecarga de Métodos

- Pode ser feita da mesma maneira que fizemos com os construtores.
- Quando sobrecarregamos um método, devemos manter a semântica: não é um bom projeto termos um método sobrecarregado cujas versões fazem coisas completamente diferentes.
- A classe Math possui vários métodos sobrecarregados. Note que a semântica das várias versões são compatíveis.

```
int a = Math.abs(-10);     // a = 10;
double b = Math.abs(-2.3); // b = 2.3;
```

Sobrecarga de Métodos (cont.)

- *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*

Fundamentos da Linguagem

- Herança
  - Como vimos anteriormente, classes podem ser compostas em hierarquias, através do uso de *herança*.
  - Quando uma classe herda de outra, diz-se que ela a *estende* ou a *especializa*, ou os dois.
  - Herança implica tanto herança de interface quanto herança de código.

Fundamentos da Linguagem

- Herança de Interface x Herança de Código
  - Herança de interface significa que a classe que herda recebe todos os métodos declarados pela superclasse que não sejam *privados*.
  - Herança de código significa que as *implementações* desses métodos também são herdadas. Além disso, os campos que não sejam privados também são herdados.

Fundamentos da Linguagem

- Herança em Java
  - Quando uma classe A herda de B, diz-se que A é a sub-classe e estende B, a superclasse.
  - Uma classe Java estende apenas uma outra classe a essa restrição damos o nome de herança simples.
  - Para criar uma sub-classe, usamos a palavra reservada extends.

Fundamentos da Linguagem

- Herança
  - Java somente suporta herança simples para a especialização de classes

```
class Ônibus extends Veículo {
    ...
}
...
Veículo v[100];    // array heterogêneo
...
v[15] = new Ônibus(100);
v[16] = new Bicicleta;
v[15].Abastece();  // Ônibus.Abastece-Polimorfismo
```


Fundamentos da Linguagem

- Exemplo de Herança
  - Podemos criar uma classe que represente um pixel a partir da classe Point. Afinal, um Pixel é um ponto colorido.

```
public class Pixel extends Point {
  int color;
  public Pixel(int x, int y, int c) {
    super(x, y);
    color = c;
  }
}
```

Fundamentos da Linguagem

Herança de Código

- A classe Pixel herda a interface e o código da classe Point. Ou seja, Pixel passa a ter tanto os campos quanto os métodos (com suas implementações) de Point.

```
Pixel px = new Pixel(1,2,0); //Pixel de cor 0
px.move(1,0);                // Agora px está em (2,2)
```


Fundamentos da Linguagem

super

- Note que a primeira coisa que o construtor de Pixel faz é chamar o construtor de Point, usando, para isso, a palavra reservada super.
- Isso é necessário pois Pixel é uma extensão de Point, ou seja, ela deve inicializar sua parte Point antes de inicializar sua parte estendida.
- Se nós não chamássemos o construtor da superclasse explicitamente, a linguagem Java faria uma chamada ao construtor padrão da superclasse automaticamente.

Fundamentos da Linguagem

- super
  - Usa-se a palavra reservada super para referenciar a classe-pai em uma hierarquia de classes Java

```
class CaminhãoTanque extends Veículo {
  void Bate() {
    super.Bate(); if (CargaInflamável()) Explode();
  }
}

class A { void f() { ... } }
class B extends A { void f() { ... } }
class C extends B {
  void f() { super.f();  ou  B.f();        // Acessando B
             super.super.f();  ou  A.f();  // Acessando A
  }
}
```


Fundamentos da Linguagem

- super
  - Na implementação de um construtor, é possível chamar construtor da classe-base usando também a palavra reservada super

```
class Ônibus extends Veículo {
  Ônibus(int LotaçãoMáxima) {
    super(LotaçãoMáxima);
    ...
  }
}
```

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

super() e this()

- 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!

Árvore × Floresta

- As linguagens OO podem adotar um modelo de hierarquia em *árvore* ou em *floresta*.
- Árvore significa que uma única hierarquia compreende todas as classes existentes, isto é, existe uma superclasse comum a todas as classes.
- Floresta significa que pode haver diversas árvores de hierarquia que não se relacionam, isto é, não existe uma superclasse comum a todas as classes.

Fundamentos da Linguagem

- Modelo de Classes de Java
  - Em Java a classe Object é a raiz da hierarquia de classes à qual todas as classes existentes pertencem. Quando não declaramos que uma classe estende outra, ela implicitamente estende Object;
  - Uma das vantagens de termos uma superclasse comum, é termos uma funcionalidade comum a todos os objetos:
    - Por exemplo, a classe Object define um método chamado toString que retorna um texto descritivo do objeto;
    - Um outro exemplo é o método finalize usado na destruição de um objeto, como já dito.

Fundamentos da Linguagem

- Especialização × Extensão
  - Uma classe pode herdar de outra para especializá-la redefinindo métodos, sem ampliar sua interface.
  - Uma classe pode herdar de outra para estendê-la declarando novos métodos e, dessa forma, ampliando sua interface.
  - Ou as duas coisas podem acontecer simultaneamente.

Exercícios - Questão 4

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*

Fundamentos da Linguagem

- Polimorfismo
  - Polimorfismo é a capacidade de um objeto tomar diversas formas. A capacidade polimórfica decorre diretamente do mecanismo de herança. Ao estendermos ou especializarmos uma classe, não perdemos compatibilidade com a superclasse.
  - A sub-classe de Point, Pixel, é compatível com ela, ou seja, um Pixel, além de outras coisas, é um ponto. Isso implica que, sempre que precisarmos de um ponto, podemos usar um Pixel em seu lugar.

```
Point[] pontos = new Point[5]; // um array de pontos
 pontos[0] = new Point();
 pontos[1] = new Pixel(1,2,0); // OK! um pixel é um ponto
 pontos[2] = new String("Alo");// ERRO!, não é um ponto
```

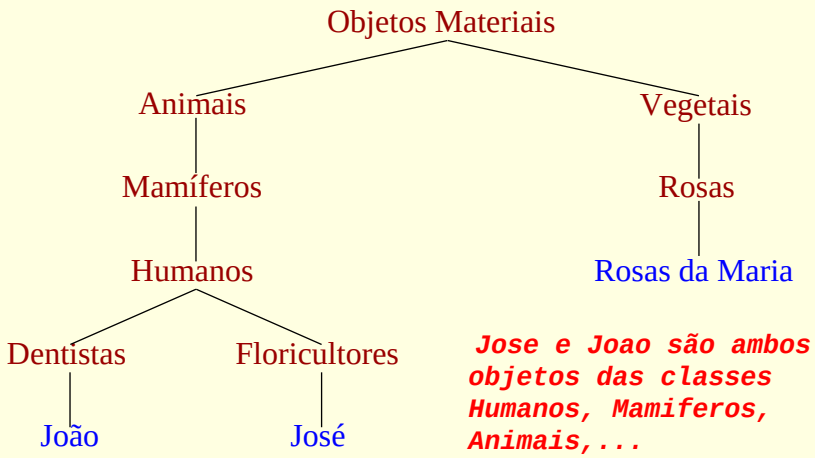

Fundamentos da Linguagem

- Note que um Pixel pode ser usado sempre que se necessita um ponto. Porém, o contrário não é verdade: não podemos usar um ponto quando precisamos de um pixel.

```
Point pt = new Pixel(0,0,1); //OK! pixel é ponto.
Pixel px = new Point(0,0);   //ERRO! ponto não é pixel.
```

- Conclusão:
  - Polimorfismo é o nome formal para o fato de que quando precisamos de um objeto de determinado tipo, podemos usar uma versão mais especializada dele. Esse fato pode ser bem entendido analisando-se a árvore de hierarquia de classes.

Uso do Polimorfismo



Jose e Joao são ambos objetos das classes Humanos, Mamiferos, Animais,...

Fundamentos da Linguagem

- Exemplo

```
public class Point {
  ...
  public void print() {
    System.out.println("Point ("+x+","+y+")");
  }
}
```

Com essa modificação, tanto a classe Point quanto a classe Pixel agora possuem um método que imprime o ponto representado. Podemos voltar ao exemplo do array de pontos e imprimir as posições preenchidas.

```
Point pt = new Point();      // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)
```

Fundamentos da Linguagem

```
pt.print(); // Imprime: “Ponto (0,0)”
px.print(); // Imprime: “Ponto (0,0)”
```

Porém, a implementação desse método não é boa para um Pixel pois a cor não é impressa. Para resolver o problema fazemos a classe Pixel redefinir o método print de forma adequada.

```
public class Pixel extends Point {
        ...
  public void print() {
    System.out.println(”Pixel(”+x+ “,” +y+ “,” + color+“)”);
  }
}
```

Com essa nova modificação, a classe Pixel agora possui um método que imprime o pixel de forma correta.

Fundamentos da Linguagem

```
Point pt = new Point();      // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)
pt.print() ; // Imprime: “Ponto (0,0)”
px.print();  // Imprime: “Pixel (0,0,0)”
```

Voltemos ao exemplo do Array de pontos.

```
Point[] pontos = new Point[5];
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);

pontos[0].print(); // Imprime: “Ponto (0,0)”
pontos[1].print(); // Imprime: “Pixel (1,2,0)”
```

Fundamentos da Linguagem

- Polimorfismo – outro exemplo

```
class Veiculo {
   public Veiculo() { System.out.print("Veiculo "); }
   public void checkList(){ System.out.println("Veiculo.checkList"); }
   public void adjust() { System.out.println("Veiculo.adjust"); }
   public void cleanup() { System.out.println("Veiculo.cleanup"); }
}
class Automovel extends Veiculo {
   public Automovel() { System.out.println("Automovel"); }
   public void checkList() { System.out.println("Automovel.checkList");
   }
   public void adjust() { System.out.println("Automovel.adjust"); }
   public void cleanup() { System.out.println("Automovel.cleanup"); }
}
class Bicicleta extends Veiculo {
   public Bicicleta() { System.out.println("Bicicleta"); }
   public void checkList(){System.out.println("Bicicleta.checkList"); }
   public void adjust() { System.out.println("Bicicleta.adjust"); }
   public void cleanup() { System.out.println("Bicicleta.cleanup"); }
}
```

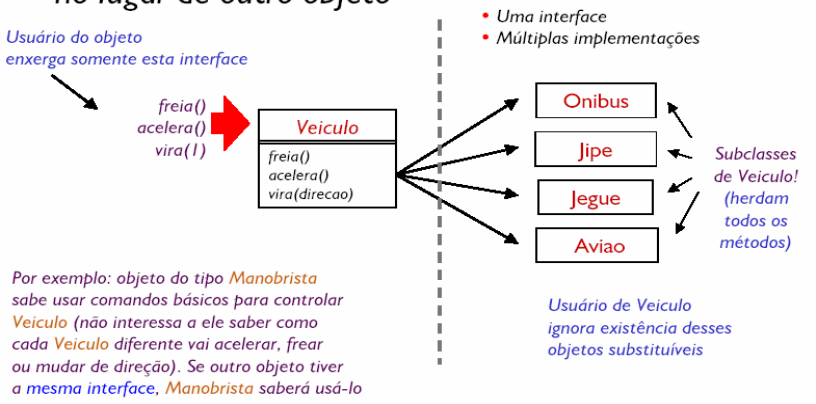

Fundamentos da Linguagem

```
public class Oficina {
   Random r = new Random();
   public Veiculo proximo() {
     Veiculo v; int code = r.nextInt();
     if (code%2 == 0) v = new Automovel();
     else v = new Bicicleta();
     return v;
   }
   public void manter(Veiculo v) {
       v.checkList(); v.adjust(); v.cleanup();
   }
   public static void main(String[] args) {
     Oficina o = new Oficina(); Veiculo v;
     for (int i=0; i<4; ++i) {
       v = o.proximo(); o.manter(v); // polimorfismo !
     }
   }
}
```

Diga o que será impresso !

Outro exemplo - Polimorfismo

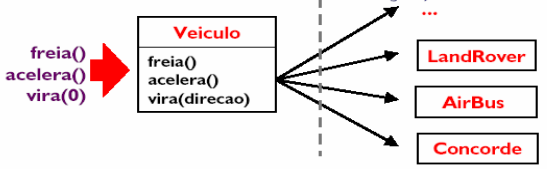
- *Polimorfismo significa que um objeto pode ser usado no lugar de outro objeto*

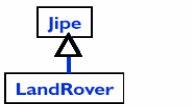


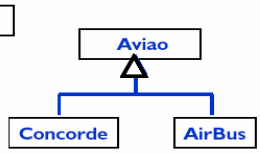
Por exemplo: objeto do tipo Manobrista sabe usar comandos básicos para controlar Veiculo (não interessa a ele saber como cada Veiculo diferente vai acelerar, frear ou mudar de direção). Se outro objeto tiver a mesma interface, Manobrista saberá usá-lo

Polimorfismo => Código extensível

- *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)*





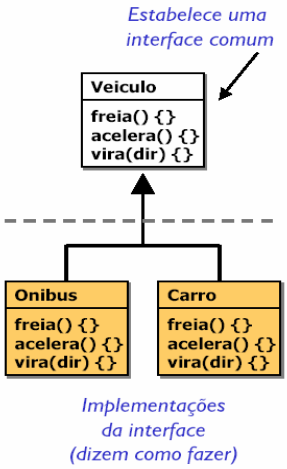


Mesmo nome. Implementações diferentes.

```
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 x 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



Fundamentos da Linguagem

Late Binding (Ligação tardia)

As linguagens OO possuem um recurso chamado late binding, que permite o adiamento da resolução de um método até o momento no qual ele deve ser efetivamente chamado. Ou seja, a resolução do método acontecerá em tempo de execução, ao invés de em tempo de compilação. No momento da chamada, o método utilizado será o definido pela classe real do objeto.

Voltando ao exemplo do array de pontos, agora que cada classe possui sua própria codificação para o método print, o ideal é que, ao correrm os o array imprimindo os pontos, as versões corretas dos métodos fossem usadas. Isso realmente acontece, pois as linguagens OO usam um recurso chamado late binding.

Late Binding na prática

- Graças a esse recurso, agora temos:

```
Point[] pontos = new Point[5];
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);
pontos[0].print(); // Imprime: “Point (0,0)”
pontos[1].print(); // Imprime: “Pixel (1,2,0)”
```

- Suporte ao 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;

Late Binding na prática

- Late Binding (cont.)
  - O objeto chamado pode ser da mesma classe ou de uma subclasse da referência (TODA a interface está implementada no objeto !);
  - Uma única referência, pode ser ligada, durante a execução, a vários objetos diferentes (a referência é dita polimórfica)

Fundamentos da Linguagem

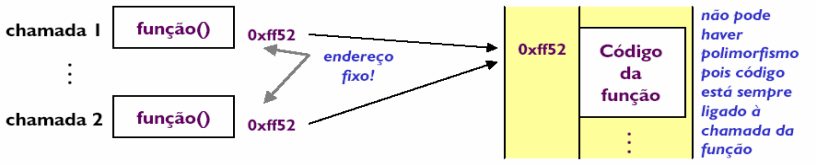
- Late Binding x Eficiência
  - O uso de late binding implica em perda no desempenho dos programas visto que a cada chamada de método um processamento adicional deve ser feito, devido ao uso da indereção causada pelo mecanismo de implementação do late-binding (tabelas virtuais). Esse fato levou várias linguagens OO a permitir a construção de métodos constantes (chamados métodos finais em Java), ou seja, métodos cujas implementações não podem ser redefinidas nas sub-classes.

Late Binding => suporte Polimorfismo

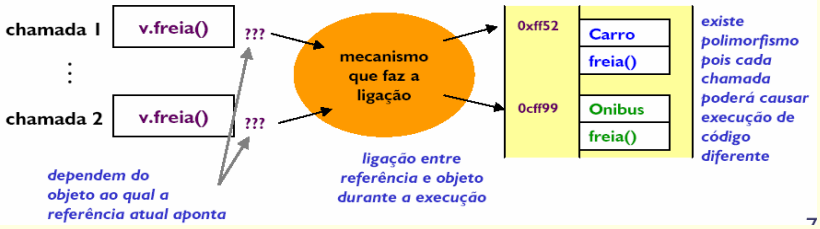
- *Suporte a polimorfismo depende do suporte à ligação tardia (late binding) de chamadas de função*
  - *A referência (interface) é conhecida em **tempo de compilação** mas o objeto a que ela aponta (implementação) não é*
  - *O objeto pode ser da mesma classe ou de uma subclasse da referência (garante 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)*

Late x Early Binding

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

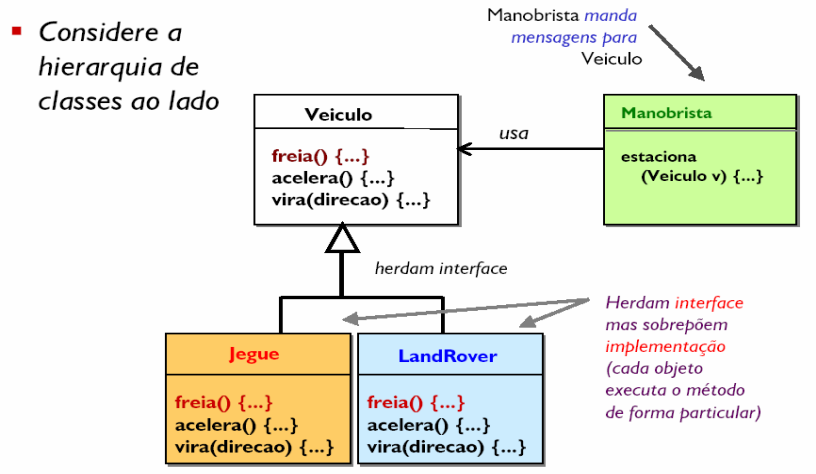


- *Em tempo de execução (late binding) - Java!*

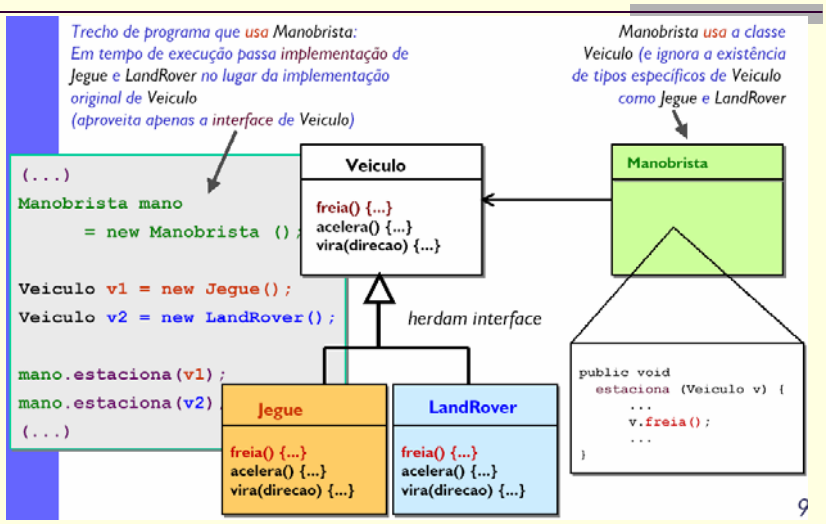


Outro exemplo - Polimorfismo

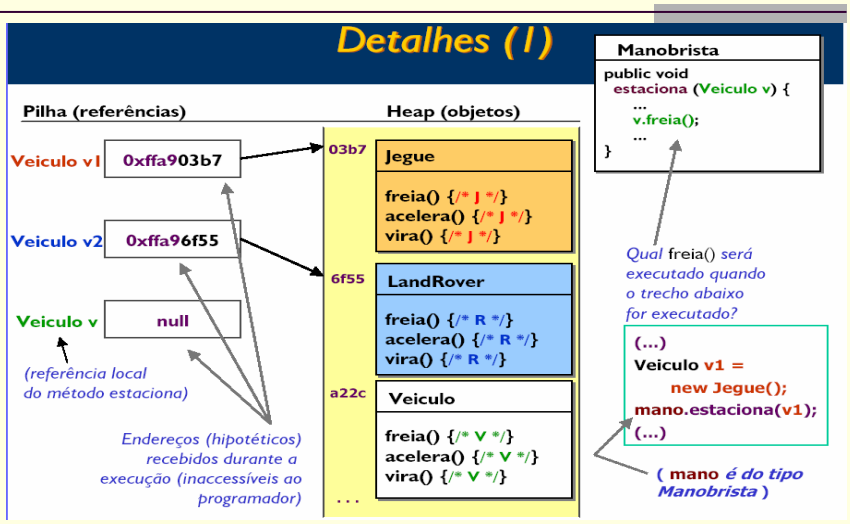
- *Considere a hierarquia de classes ao lado*



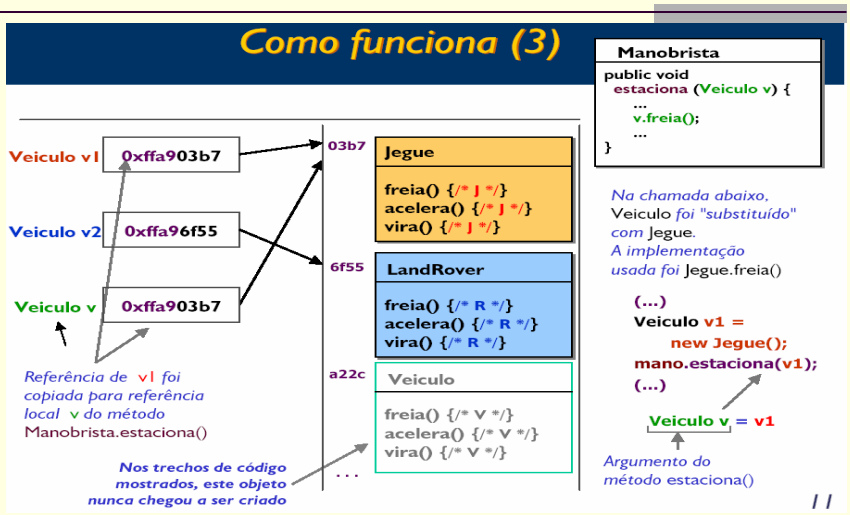
Outro exemplo - Polimorfismo



Outro exemplo - Polimorfismo



Outro exemplo - Polimorfismo



Alguns comentários

- 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

Fundamentos da Linguagem

- Atributos Constantes
  - Java permite declarar um campo ou uma variável local que, uma vez inicializada, tenha seu valor fixo. Para isso utilizamos o modificador final.

```
class A {
   final int ERR_COD1 = -1;  // constantes !
   final int ERR_COD2 = -2;
   ...
}
```

- Métodos e Classes Constantes em Java

```
public class A { public final int f() { ...  }}
```

  - Uma classe inteira pode ser definida final. Isto serve para evitar que também a classe seja estendida.

```
public final class B {  ... }
```


Fundamentos da Linguagem

- Conversão de Tipo – Type Casting
  - Podemos usar uma versão mais especializada quando precisamos de um objeto de certo tipo mas o contrário não é verdade. Por isso, para fazer a conversão de volta ao tipo mais especializado, teremos que fazê-lo explicitamente.
  - A conversão explícita de um objeto de um tipo para outro é chamada type casting. Tipos genéricos (acima, na hierarquia) sempre podem receber objetos de suas subclasses: upcasting, enquanto tipos específicos (abaixo, na hierarquia) não podem receber explícitamente seus objetos que foram declarados como referências de suas superclasses: downcasting.

Fundamentos da Linguagem

- Conversão de Tipo – Type Casting (cont)

```
Point pt1 = new Pixel(0,0,1); // OK – upcasting, pixel é ponto !
Pixel px = (Pixel)pt1;        // OK – downcasting valido!

Point pt2 = new Point(0, 0);
Pixel px= pt2;       // ERRO não compila, downcasting invalido!
Pixel px =(Pixel)pt2;// Compila, erro execução ClassCastExcetion!

Point pt=new Point();
Pixel px=(Pixel)pt; // Erro execução: ClassCastException, pq ?

pt = new Pixel(0,0,0);
px = pt;        // ERRO compilação: falta type-cast explicíto.
```


Fundamentos da Linguagem

Para evitar a exceção deveríamos fazer o seguinte:

```
if (pt instanceof Pixel) {
   Pixel px = (Pixel)pt; .....
}
```

Exercícios - Questão 5

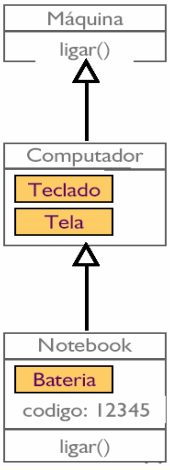
- Note que, assim como o late binding, o type casting e instanceof só podem ser resolvidos em tempo de execução: só quando o programa estiver rodando é possível saber o valor que uma dada variável terá e, assim, decidir se a conversão é válida ou não.

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*

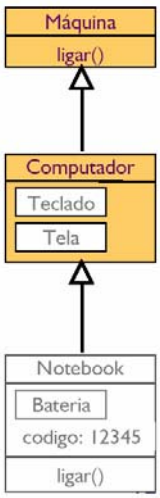
Exemplo (1) - Herança

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

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

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



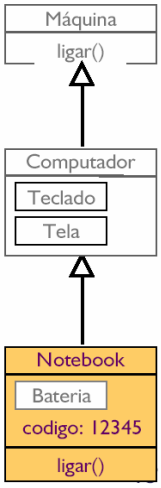
Exemplo 2 - Herança

```
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 - Herança

```
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()

```
Maquina()
Notebook.ligar(); codigo = 0
Tela()
Teclado()
Computador()
Bateria()
Notebook(); codigo = 12345
```

1. Construtor de Maquina chamado

2. Método ligar() de Notebook (e não de Maquina) chamado!

3. PROBLEMA!!!!! Variável codigo, de Notebook ainda não foi inicializada quando ligar() foi chamado!

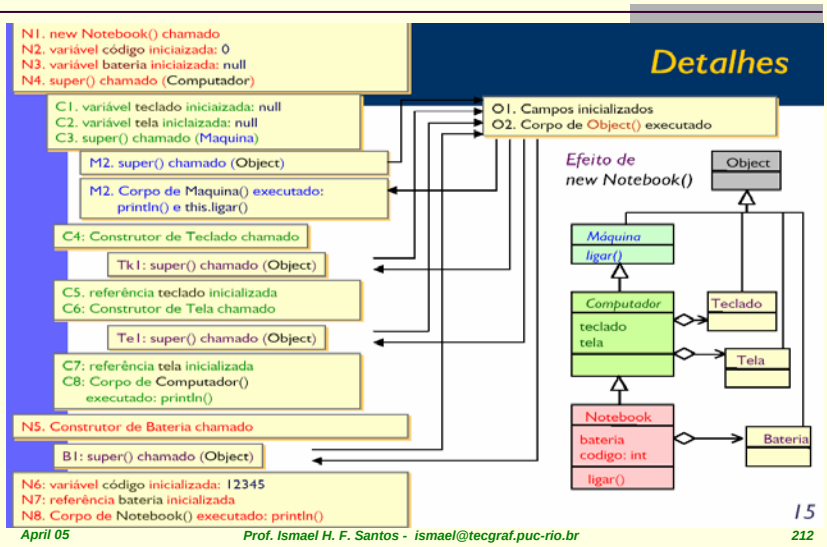
4. Variáveis tela e teclado, membros de Computador são inicializadas

5. Construtor de Computador chamado

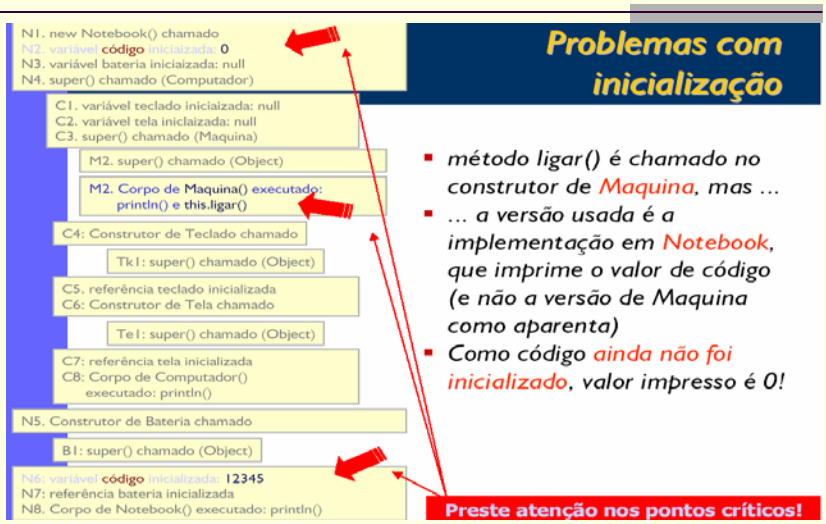
6. Variável bateria, membro de Notebook é inicializada

7. Construtor de Notebook chamado. Variável codigo finalmente inicializada

Analise da execução



Analise da execução



Como evitar o problema ?

- **Evite chamar métodos locais dentro de construtores**
  - **Construtor (qq um da hierarquia) sempre usa a versão sobreposta do método (late-binding)**
- **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 provocando NullPointerException**
  - **Dependa de outros métodos sobrecarregados**
- **Use apenas métodos finais em construtores**
  - **Estes métodos não podem ser sobrecarregados nas subclasses**

Inicialização estática

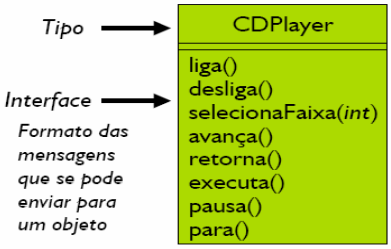
- *Para inicializar valores estáticos, é preciso atuar logo após a carga da classe*
  - *O bloco 'static' tem essa finalidade*
  - *Pode estar em qualquer lugar da classe, mas será chamado antes de qualquer outro método ou variável*

```
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!*

Sumário - Objetos

- *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

Sumário - Objetos

- 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

Sumário - Objetos

- 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

POO-Java

Interfaces e Classes Abstratas

Interfaces e Classes Abstratas

- Classes Abstratas
  - Ao criarmos uma classe para ser estendida, às vezes codificamos vários métodos usando um método para o qual não sabemos dar uma implementação, ou seja, um método que só sub-classes saberão implementar.
  - Uma classe desse tipo não deve poder ser instanciada pois sua funcionalidade está incompleta. Tal classe é dita abstrata.
  - Java utiliza o modificador abstract para declarar uma classe abstrata. Métodos também podem ser declarados abstratos para que suas implementações fiquem adiadas para as sub-classes.

Interfaces e Classes Abstratas

- Classes Abstratas em Java

```
abstract class Veículo {
  abstract void Abastece();
}
class Ônibus extends Veículo {
   void Abastece() {
      EncheTanqueComDiesel();
   }
 }
 ...
 objAbstrato = new Veículo();    // Erro
 objConcreto = new Ônibus();     // Ok
```

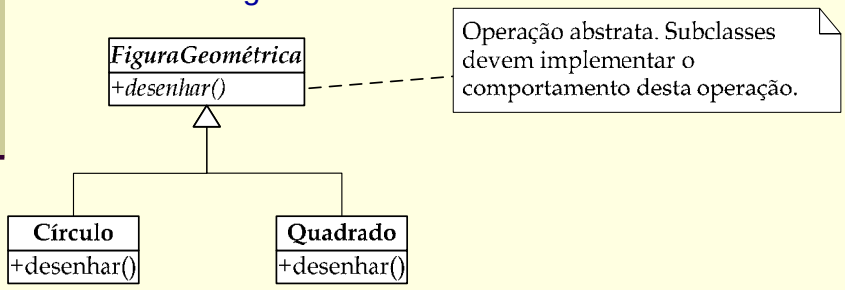

Interfaces e Classes Abstratas

- Classes Abstratas em Java

```
public abstract class Drawing {
  public abstract void draw();
  public abstract BBox getBBox();
  public boolean contains(Point p) {
    BBox b = getBBox();
    return (p.x>=b.x && p.x<b.x+b.width &&
            p.y>=b.y && p.y<b.y+b.height);
  }  ...
}
```

Interfaces e Classes Abstratas

- Seja uma classe abstrata, FiguraGeometrica.
  - Em FiguraGeometrica: `abstract void desenhar();`
- As classes Círculo e Quadrado, para serem concretas, devem fornecer implementação para o método abstrato herdado de FiguraGeometrica.

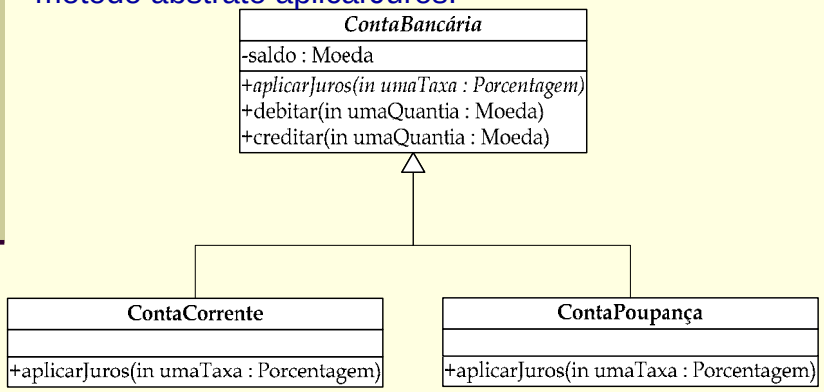


Interfaces e Classes Abstratas

```
public abstract class FiguraGeometrica {
  ...
  abstract void desenhar();
}

public class Circulo extends FiguraGeometrica {
  ...
  void desenhar() { ... }
}

public class Quadrado extends FiguraGeometrica {
  ...
  void desenhar() { ... }
}
```

Interfaces e Classes Abstratas

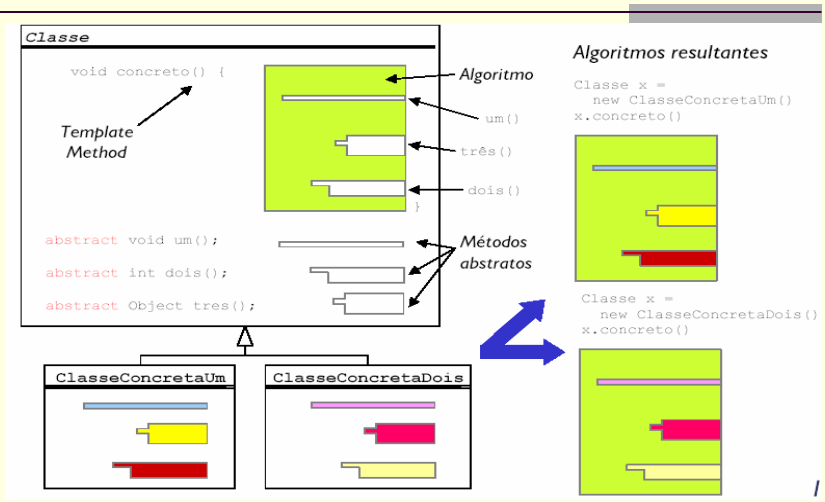
- Classes ContaCorrente e ContaPoupança herdam os métodos concretos creditar e debitar, e implementam o método abstrato aplicarJuros.



Classes Abstratas

- *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



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>";
   }
}
```

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!
```

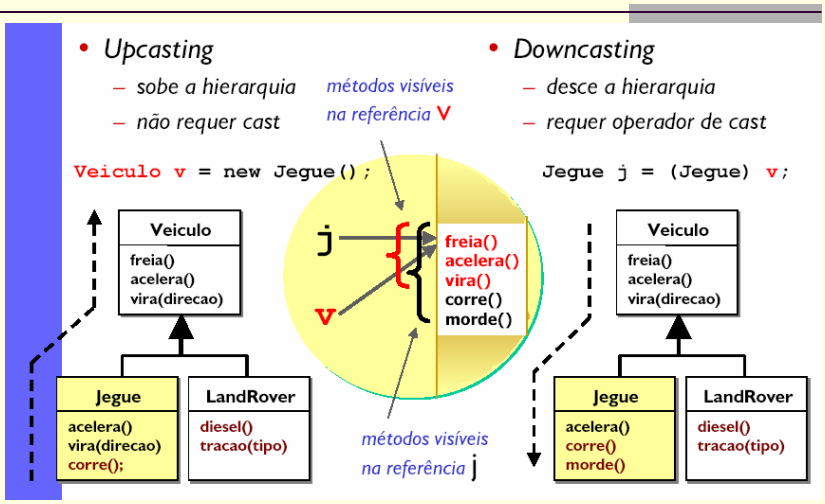
  - *O código acima não compila, apesar de v apontar para um Carro! É preciso converter a referência:*

```
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*



ClassCastException

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

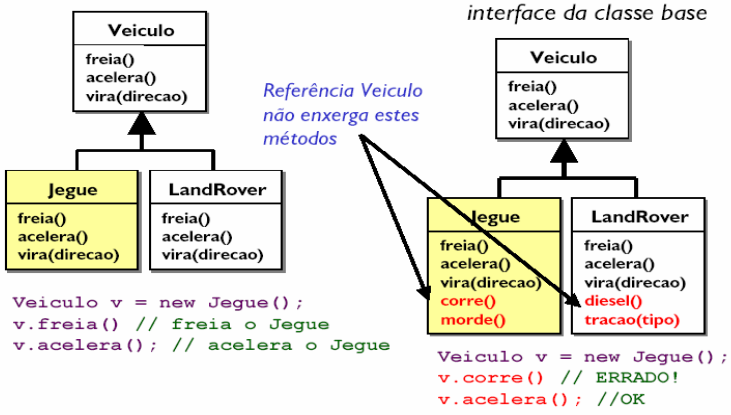
```
Veiculo v = new Onibus();
Carro c = (Carro) v; // passa na compilação
```

  - *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;
```

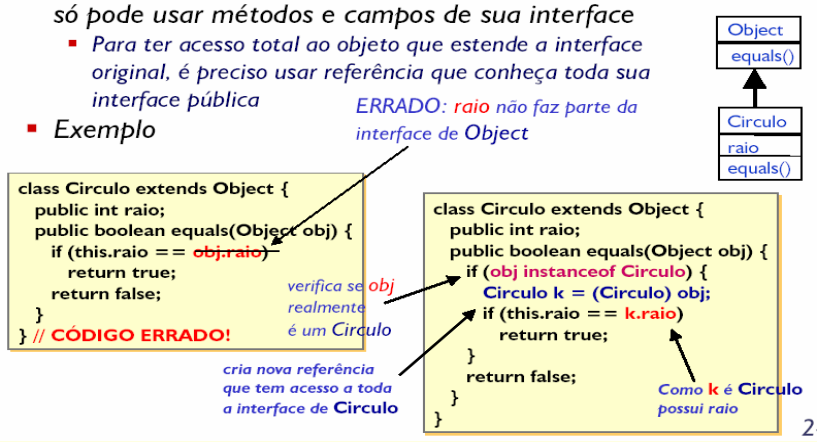
Herança Pura x Extensão

- *Herança pura: referência têm acesso a todo o objeto*
- *Extensão: referência apenas tem acesso à parte definida na interface da classe base*



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*

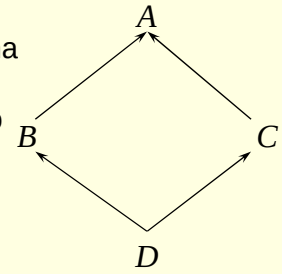


Herança Múltipla

- Herança: Simples × Múltipla
  - O tipo de herança que usamos até agora é chamado de herança simples pois cada classe herda de apenas uma outra. Existe também a chamada herança múltipla onde uma classe pode herdar de várias classes.
  - Herança múltipla não é suportada por todas as linguagens OO, pois apresenta um problema quando construímos hierarquias de classes onde uma classe herda duas ou mais vezes de uma mesma superclasse. O que, na prática, torna-se um caso comum.

Herança Múltipla (cont)

- Problemas de Herança Múltipla
  - O problema de herdar duas vezes de uma mesma classe vem do fato de existir uma herança de código.
  - Inúmeras vezes, quando projetamos uma hierarquia de classes usando herança múltipla, estamos, na verdade, querendo declarar que a classe é *compatível* com as classes herdadas. Em muitos casos, a herança de código não é utilizada.



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*

Interfaces em Java

- Interfaces
  - *Algumas linguagens OO incorporam o conceito de duas classes serem compatíveis através do uso de compatibilidade estrutural ou da implementação explícita do conceito de interface.*
  - Java não permite herança múltipla com herança de código, mas implementa o conceito de interface. É possível herdar múltiplas interfaces.
  - Em Java, uma classe estende uma outra classe e implementa zero ou mais interfaces. Para implementar uma interface em uma classe, usamos a palavra implements.

Interfaces em Java (cont)

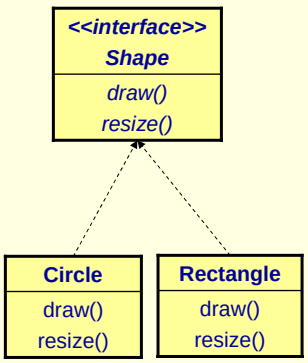
- *Interface é uma estrutura que representa uma classe abstrata "pura" em Java*
  - *Não têm atributos de dados (só pode ter constantes estáticas)*
  - *Não tem construtor*
  - *Todos os métodos são abstratos*
  - *Não é declarada como class, mas como interface*
- *Interfaces Java servem para fornecer 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*

Interfaces e Classes Abstratas

- Exemplo

```
interface Pessoa { void Come(int Kg);}

interface Ciclista extends Pessoa {
  void Pedala(int Km);
}
interface Corredor extends Pessoa{
  void Corre(int Km);
}
interface Nadador extends Pessoa {
  void Nada(int Km);
}

...

class Triatleta extends Atleta
                    implements Ciclista, Corredor, Nadador
...
```

Interfaces e Classes Abstratas




```
public interface Shape {
     double PI = 3.1425926; //static final !
     void draw();
     void resize();
}

public class Circle implements Shape {
     public void draw() {  /* draw a circle */ }
     public void resize() { /* draw a circle */ }
}

public class Rectangle implements Shape {
     public void draw() { /* draw a rectangle */  }
     public void resize() { /* draw a rectangle */ }
}
```


Interfaces e Classes Abstratas

- Comparable é uma interface pré-definida em Java.
  - Muitas classes da biblioteca Java implementam essa interface.
  - compareTo deve retornar
    - inteiro < 0 se o objeto remetente é “menor” que o parâmetro,
    - 0 se eles são iguais,
    - inteiro > 0 se o objeto remetente é maior que o parâmetro.

Definida no package java.lang:

```
public interface Comparable {
  public int compareTo( Object other );
}
```

<<interface>>
Comparable
int compareTo(Object o)

Implementando uma interface

```
public class Card implements Comparable {
      ...
  public int compareTo(Object otherObject) {
    Card other = (Card)otherObject;
    return myValue - other.myValue;
  }
}
```


Polimorfismo novamente

```
public static SelSort(Comparable[] list) {
  Comparable temp;
  int small;
  for(int i = 0; i < list.length - 1; i++) {
    small = i;
    for(int j = i + 1; j < list.length; j++) {
      if( list[j].compareTo(list[small]) < 0)
        small = j;
    }
    temp = list[i];
    list[i] = list[small];
    list[small] = list[i];
  }
}
```

Interfaces e Classes Abstratas

Exemplo de Interface

- Ao implementarmos o TAD Pilha, poderíamos ter criado uma interface que definisse o TAD e uma ou mais classes que a implementassem.

```
interface StackInterf {

  boolean isEmpty();
  void push(int n);
  int pop();
  int top();
}
```

```
class StackImpl implements
                StackInterf {

     int data[], top_index;
     ……….
}
```


Interfaces e Classes Abstratas

Membros de Interfaces

- Uma vez que uma interface não possui implementação, temos que:
  - **seus campos devem ser** públicos**,** estáticos **e** constantes**;**
  - **seus métodos devem ser** públicos **e** abstratos**.**
- Como esses qualificadores são fixos, não precisamos declará-los (note o exemplo anterior).

```
interface StackInterf {
  public abstract boolean isEmpty();
  public abstract void push(int n);
  public abstract int pop();
  public abstract int top();
}
```

Interfaces e Classes Abstratas

- Pilha revisitada

```
class StackImpl implements StackInterf {
  private int[] data;
  private int top_index;
  Stack(int size) {
    data = new int[size];
    top_index = -1;
  }
  boolean isEmpty() { return (top_index < 0); }
  void push(int n) { data[++top_index] = n; }
  int pop() { return data[top_index--]; }
  int top() { return data[top_index]; }
}
```


Interfaces e Classes Abstratas

- Limites da Herança
  - O mecanismo de herança que analisamos não resolve alguns problemas. Considere o TAD Pilha que implementamos: Ele define uma pilha de números inteiros mas isso não devia ser (e não é) necessário. Por exemplo, poderia ser útil ter uma pilha de inteiros e uma outra de objetos Point. Podemos criar pilhas específicas mas não podemos criar todas as possíveis...

Interfaces e Classes Abstratas

- Resumindo Interfaces
  - Não são classes
  - Oferece compatibilidade de tipos de objetos

```
Comparable  x; // Comparable é uma interface
x = new Pessoa(); // Pessoa implementa Comparable
```

  - Permite o uso de *instanceof*

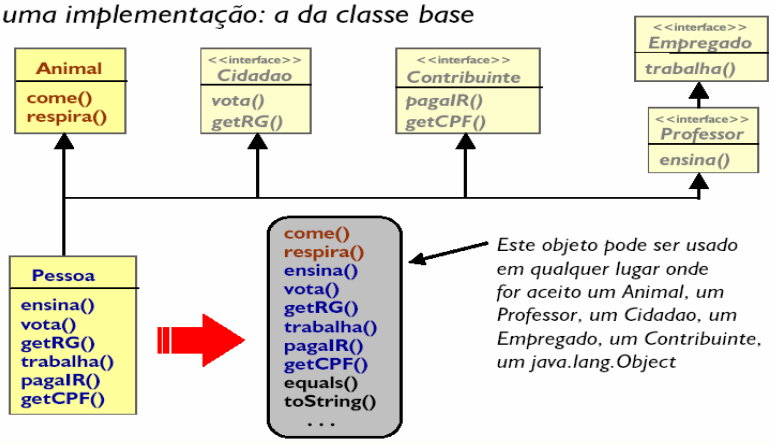
```
if (x instanceof Comparable) {…}
```

  - Uma interface pode estender outra

```
public interface Compativel extends Comparable {
  …
}
```


Herança Múltipla de Interfaces

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



Exemplo – Herança Interfaces

```
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 – Herança Interfaces (cont.)

```
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*

Exemplo – Herança Interfaces (cont.)

```
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.*

POO-Java

Classes Parametrizadas

Classes Parametrizadas

- Herança × Parametrização
  - Uma alternativa a criar novas classes para cada diferente tipo de pilha que iremos usar é parametrizar a própria classe que implementa a pilha. Várias linguagens OO suportam parametrização de tipos.
  - Parametrizar um tipo significa passar o tipo a ser usada em alguma operação como um parâmetro. No caso da pilha, poderíamos passar o tipo dos elementos que pilha deveria conter como um parâmetro do construtor, por exemplo.

Classes Parametrizadas

- Parametrização em Java

Java não provê suporte direto à construção de classes parametrizadas, ate J2SE 5.0. Como Java adota o modelo de hierarquia em árvore, com uma superclasse comum a todas as classes, e, além disso, mantém as informações de tipo em tempo de execução, podemos simular um TAD paramétrico usando type-casting.

Para simularmos parametrização em Java podemos consultar as informações de tipo em tempo de execução através do comando instance of

Classes Parametrizadas

```
Point pt = new Point();      // pt contém um ponto
boolean b = pt instanceof Pixel; // b = false
pt = new Pixel(1,2,3);       // pt contém um pixel
b = pt instanceof Pixel;     // b = true
```

Podemos então mudar a definição do nosso TAD para especificar pilhas de objetos genéricos.

```
interface StackInterf {
  boolean isEmpty();
  void push(Object obj);
  Object pop();

  Object top();
}
```

Exercícios – Questão 6