# Programação Orientada a Objetos em


Flávio Miguel Varejão

Departamento de Informática

UFES

# Processo de Desenvolvimento de Software

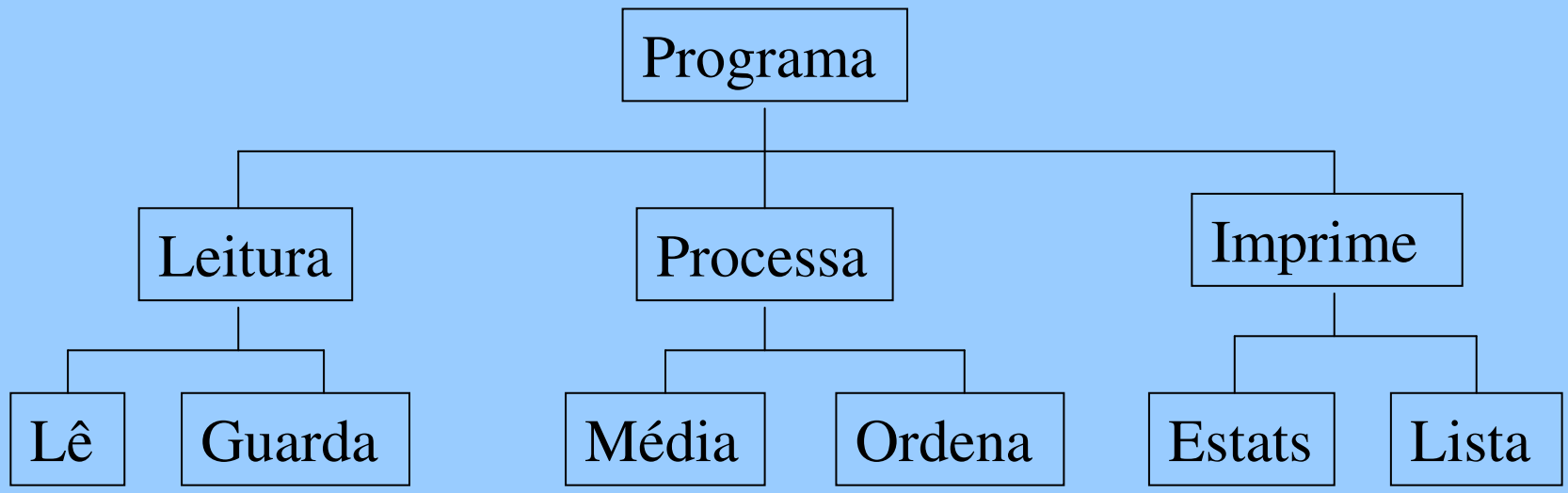
- Top-Down
  - Foco no Processo
  - Especifica Funcionalidade Desejada
  - Projeta Módulos Funcionais Hierárquicos
  - Projeta Estrutura de Dados
  - Implementação em LPs Imperativas

# Processo de Desenvolvimento de Software

- Top-Down (Recursos Humanos)
  - Funcionalidade
    - Entrada
      - Arquivo contendo nome, matrícula, idade e salário
    - Saída
      - Funcionários com menor e maior salário
      - Total de despesas com salário
      - Média salarial
      - Listagem ordenada por matrícula com dados dos funcionários marajás

# Processo de Desenvolvimento de Software

- Top-Down (Recursos Humanos)
  - Módulos Funcionais Hierárquicos



# Processo de Desenvolvimento de Software

- Top-Down (Recursos Humanos)
  - Estrutura de Dados
    - Registro para Menor Salário
    - Registro para Maior Salário
    - Variável Decimal para Total de Salário
    - Variável Decimal para Média Salarial
    - Vetor de Registros para Listagem Ordenada

# Processo de Desenvolvimento de Software

- Problemas com Abordagem Top-Down
  - Estrutura de dados global
  - Módulos funcionais necessitam conhecer a estrutura dos dados
  - Modificações na estrutura de dados implicam em modificações em todos módulos relacionados
  - Programas retratam mais a estrutura de dados do que a estrutura do problema

# Processo de Desenvolvimento de Software

- Problemas com Abordagem Top-Down (Recursos Humanos)
  - Efeitos de alterar o vetor por encadeamento dinâmico de registros é sentida em todos os módulos
  - Necessário modificar o programa inteiro

# Processo de Desenvolvimento de Software

- Bottom-Up
  - Foco nos Dados (Tipos Abtratos de Dados)
  - Especifica Entidades do Domínio
  - Estabelece seu Comportamento
  - Interface reflete funcionalidade desejada
  - Implementação especifica como atingir a funcionalidade desejada

# Processo de Desenvolvimento de Software

- Bottom-Up (Recursos Humanos)
  - Tipo Abstrato de Dados Grupo
    - Interface
      - Inserir registro
      - Excluir registro
      - Ordenar
      - Primeiro registro
      - Próximo registro
      - Final?
    - Implementação
      - Vetor ou encadeamento dinâmico ou árvore ou hash ou ???
    - Alteração da implementação da estrutura de dados não causa efeito nos módulos clientes

# Processo de Desenvolvimento de Software

- Vantagens da Abordagem Bottom-Up
  - Pensar em objetos parece ser mais natural do que em funções
  - Complexidade encapsulada nos objetos
    - Exemplo da TV
  - Distanciamento da arquitetura
  - Maior facilidade para reuso
  - Maior facilidade de manutenção

# LPs Orientadas a Objetos

- Características
  - Tudo que existe são objetos
  - Um programa é composto por um grupo de objetos dizendo um ao outro o que fazer por meio de mensagens enviadas
  - Cada objeto tem sua própria memória feita de outros objetos
  - Todo objeto tem uma classe
  - Todos objetos de uma classe particular recebem as mesmas mensagens

# Tipos e Classes

- Tipo (int)
  - conjunto de valores
  - conjunto de operações
- Classe (coord)
  - conjunto de valores
  - conjunto de operações

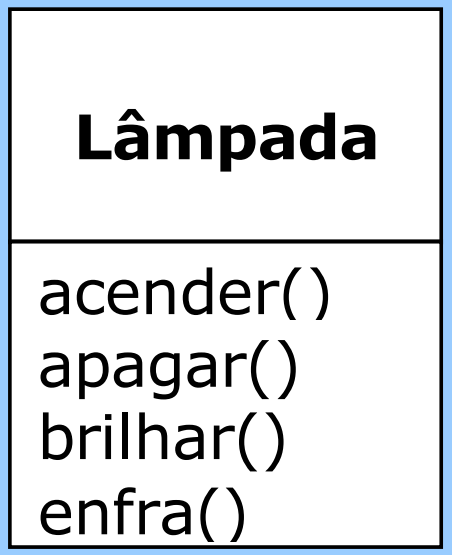
# Imperativo x OO

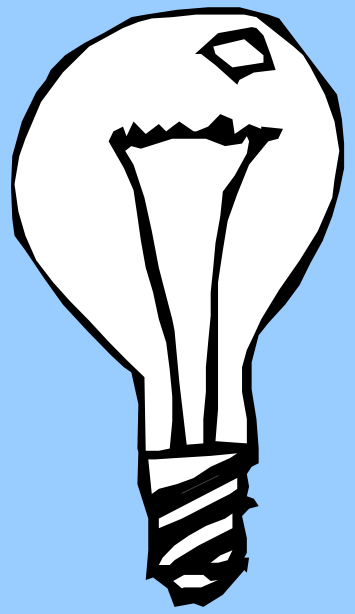
- Estrutura de Dados => Conjunto de Valores (atributos ou variáveis membro ou campos)
- Subprogramas => Operações (métodos ou funções membro)
- Chamada de Subprograma => Comunicação (mensagens)

# Objetos Tem Interface

Nome tipo

Interface



```
Lampada lt = new Lampada();
lt.acender();
```

# Ocultamento de Informação

- Permite controlar o acesso aos atributos do tipo separando interface e implementação
  - interface é o que o usuário necessita para resolver seu problema
  - implementação necessária apenas para o funcionamento interno do tipo
- garante a integridade do tipo
- permite mudanças no tipo sem afetar código do usuário

# Partes de uma Classe

- Cabeçalho

  public class Conta
- Definição de Atributos

  int numero;

  byte digitoverific;

  float saldo;
- Definição de Métodos

  void abertura( ) { … }

  void fechamento ( ) { … }

  void extrato ( ) { … }

# Partes de uma Classe

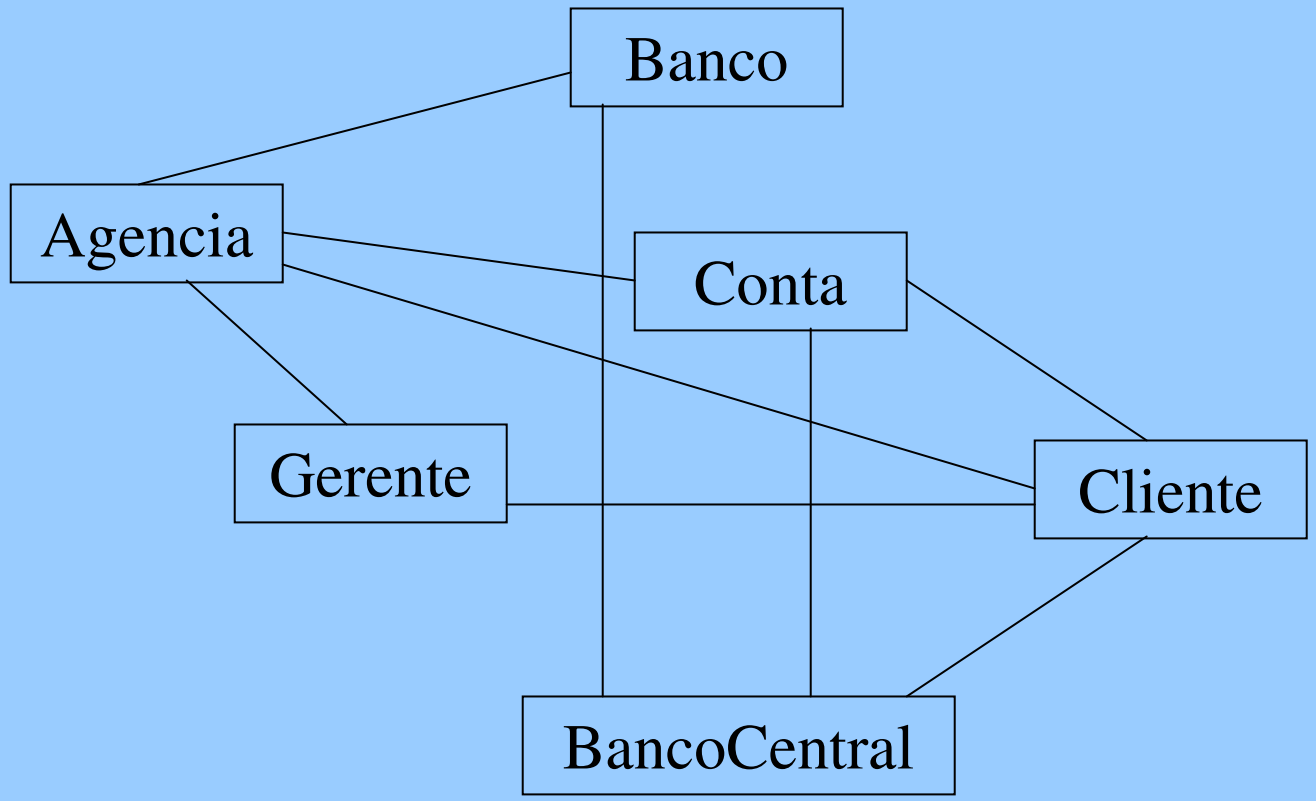
```
public class Conta {
      int numero;
      byte digitoverific;
      float saldo;
      public void abertura( ) {
        System.out.println (“Abre uma conta!”);
      }
      public void fechamento( ) {
        System.out.println (“Fecha uma conta!”);
      }
      public void fechamento( ) {
        System.out.println (“Imprime o extrato!”);
      }
}
```

# Partes de um Programa

- Várias classes
  - Cliente
  - Banco
  - Agência
  - Conta
  - Gerente
- Uma classe com o método:

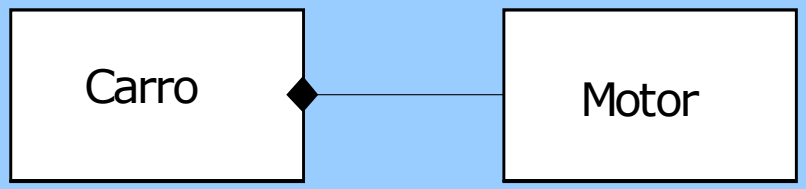
  public static void main (String [] args) { … }

# Partes de um Programa



# Reuso da Implementação

- Através da criação de um novo objeto de uma classe
- Através do uso de uma classe para definir atributo de outra classe, isto é,  composição (ou agregação)



# Atributos

```
class SoDados {
  int i;
  float f;
  boolean b;
}
```

```
SoDados d = new SoDados();
```

```
d.i = 47;
d.f = 1.1;
d.b = false;
```

```
estacao.reservatorio.capacidade = 100;
```

# Valores Default de Primitivos

| Tipo Primitivo | Default |
|---|---|
| `boolean` | `False` |
| `char` | `'\u0000'` `(null)` |
| `byte` | `(byte)0` |
| `short` | `(short)0` |
| `int` | `0` |
| `long` | `0L` |
| `float` | `0.0f` |
| `double` | `0.0d` |

# Métodos

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nada(String s) { return; }
void nada2(String s, int i) {}
```

# Lista de Parâmetros Variável em J2SE 5.0

```
void testaArgsVar(int k, float f, String ... args) {
  System.out.println(k);
  System.out.println(f);
  for (int i=0;i <args.length; i++) {
    System.out.println (args[i]);
  }
}

testaArgsVar(1,2, “teste”, “args”, “var”);
testaArgsVar(3,4, “teste”, “args”);
testaArgsVar(5,6);
```

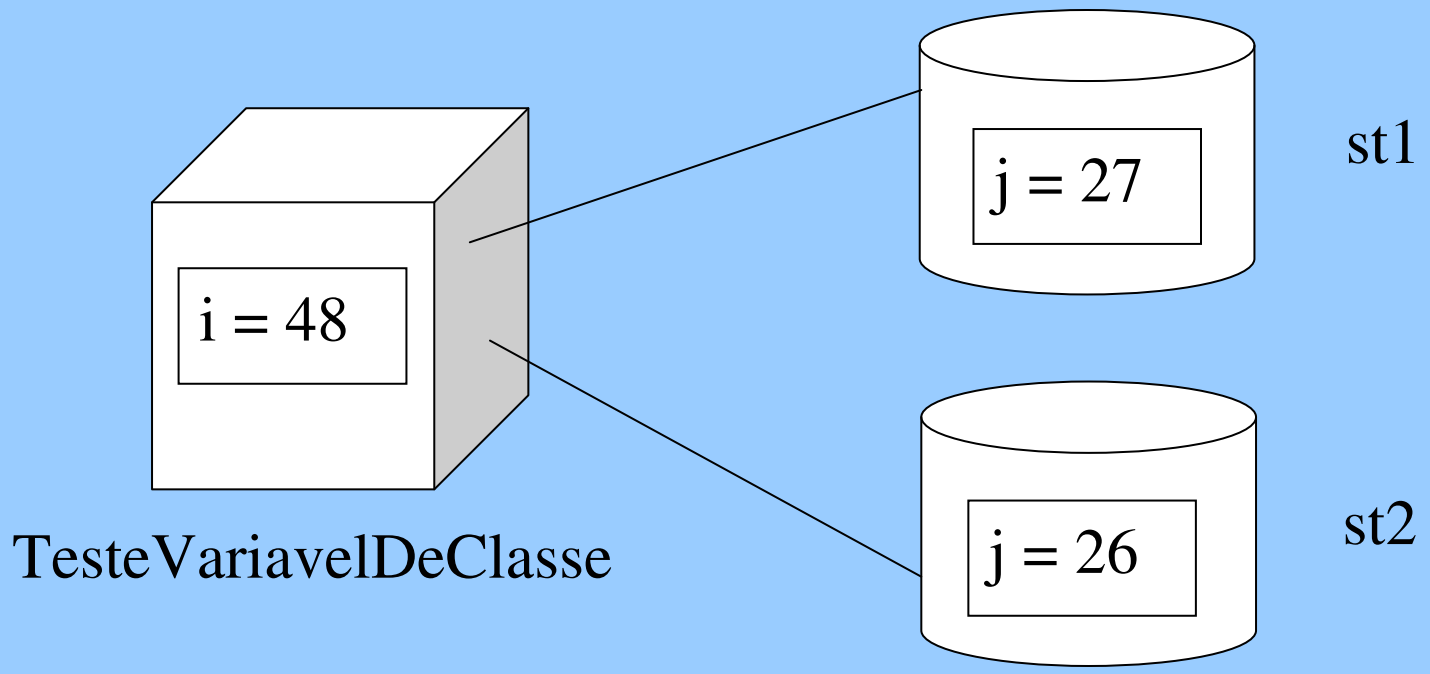
# Atributos de Classe

```
class TesteVariavelDeClasse {
    static int i = 47;
    int j = 26;
}
```

```
TesteVariavelDeClasse st1 = new TesteVariavelDeClasse ();
TesteVariavelDeClasse st2 = new TesteVariavelDeClasse ();
```

```
TesteVariavelDeClasse.i++;
```

# Atributos de Classe



# Métodos de Classe

```
class MetodoDeClasse {
  static void incr() { TesteVariavelDeClasse.i++; }
}

MetodoDeClasse sf = new MetodoDeClasse ();
sf.incr();

MetodoDeClasse.incr();
```

# Construtores

- Inicialização de Variáveis

  int j;

  int j = 0;

- Inicialização de Objetos
  - Ao criar objetos
- Problema com Método Inicializa
  - Exemplo pilha

# Inicialização com Construtores

- Métodos Especiais
  - usados para garantir inicialização
  - mesmo nome da classe
  - não tem tipo de retorno
  - default não tem parâmetros

```
Conta (float deposito) {
    saldo = deposito;
}
```

# Métodos Construtores

```
// ConstrutorSimples.java
// Demonstracao de um construtor simples.
class Rocha {
  Rocha() { // Este e o construtor
    System.out.println("Criando uma Rocha");
  }
}

public class ConstrutorSimples {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rocha();
  }
}
```

# Construtores com Argumentos

```
// ConstrutorSimples2.java
// Construtor pode ter argumentos.
class Rocha2 {
  Rocha2(int i) {
    System.out.println(
      "Criando Rocha numero " + i);
  }
}
public class ConstrutorSimples2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rocha2(i);
  }
}
```

# Sobrecarga de Métodos

- Mesmo nome para diferentes métodos de uma mesma classe
- Vantagens
  - Pode-se querer construir objetos em diferentes contextos
    - Exemplo BigInt
  - Evita que se tenha de guardar inúmeros nomes
    - print x printInt, printChar, printBool, ...

# Sobrecarga

- Distinção pela lista de parâmetros
- Tanto construtores quanto outros métodos

```
Conta ( ) {
    saldo = 0.0;
}
Conta (float deposito) {
    saldo = deposito;
}
```

# Sobrecarga de Métodos

```
// Sobrecarga.java
// Demonstracao de sobrecarga.
class Arvore {
  int altura;
  Arvore() {
    prt("Plantando uma semente ");
    altura = 0;
  }
  Arvore(int i) {
    prt("Criando nova arvore que tem "
        + i + " metros de altura ");
    altura = i;
  }
```

# Sobrecarga de Métodos

```
void info() {
    prt("Arvore tem " + altura
        + " metros de altura ");
  }
  void info(String s) {
    prt(s + ": Arvore tem "
        + altura + " metros de altura ");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

public class Sobrecarga {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Arvore t = new Arvore(i);
      t.info();
      t.info("metodo sobrecarregado ");
    }
    // Construtor sobrecarregado:
    new Arvore();
  }
}
```

# Sobrecarga de Métodos

```
// Sobrecarga baseada na ordem dos argumentos.
public class OrdemSobrecarregada {
  static void print(String s, int i) {
    System.out.println("String: " + s +
      ", int: " + i);
  }
  static void print(int i, String s) {
    System.out.println("int: " + i +
      ", String: " + s);
  }
  public static void main(String[] args) {
    print("primeira String ", 11);
    print(99, "primeiro Int ");
  }
}
```

# Sobrecarga e Promoção

```
// Promocao de tipos primitivos e sobrecarga
public class PrimitivoSobrecarregado {
  static void prt(String s) {
    System.out.println(s);
  }
  void f1(char x) { prt("f1(char): "+x); }
  void f1(byte x) { prt("f1(byte): "+x); }
  void f1(short x) { prt("f1(short): "+x); }
  void f1(int x) { prt("f1(int): "+x); }
  void f1(long x) { prt("f1(long): "+x); }
  void f1(float x) { prt("f1(float): "+x); }
  void f1(double x) { prt("f1(double): "+x); }
  void f2(short x) { prt("f2(short): "+x); }
  void f2(long x) { prt("f2(long): "+x); }
  void f2(double x) { prt("f2(double): "+x); }
  void f3(double x) { prt("f3(double): "+x); }
```

# Sobrecarga e Promoção

```
void testeConstVal() {
    prt("Testando com 5");
      f1(5);f2(5);f3(5);
   }
  void testeChar() {
    char x = 'x';
    prt("argumento char:");
    f1(x);f2(x);f3(x);
  }
  void testeByte() {
    byte x = 0;
    prt("argumento byte:");
    f1(x);f2(x);f3(x);
 }
```

# Sobrecarga e Promoção

```
void testeFloat() {
    float x = 0;
    prt("argumento float:");
    f1(x);f2(x);f3(x);
  }
public static void main(String[] args) {
    PrimitivoSobrecarregado p =
      new PrimitivoSobrecarregado ();
    p.testeConstVal();
    p.testeChar();
    p.testeByte();
    p.testeFloat();
  }
}
```

# Sobrecarga e Rebaixamento

```
// Rebaixamento de primitivos e sobrecarga.
public class Rebaixamento {
  static void prt(String s) {
    System.out.println(s);
  }
  void f1(int x) { prt("f1(int)"); }
  void f1(double x) { prt("f1(double)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(float x) { prt("f2(float)"); }
  void f3(short x) { prt("f3(short)"); }
  void f3(long x) { prt("f3(long)"); }
  void f4(byte x) { prt("f4(byte)"); }
  void f4(int x) { prt("f4(int)"); }
  void f5(char x) { prt("f5(char)"); }
  void f5(short x) { prt("f5(short)"); }
  void f6(char x) { prt("f6(char)"); }
  void f6(byte x) { prt("f6(byte)"); }
  void f7(char x) { prt("f7(char)"); }
```

# Sobrecarga e Rebaixamento

```
  void testeDouble() {
      double x = 0;
      prt("argumento double:");
      f1(x);f2((float)x);f3((long)x);f4((int)x);
      f5((short)x);f6((byte)x);f7((char)x);
    }
    public static void main(String[] args) {
      Rebaixamento p = new Rebaixamento ();
      p.testDouble();
    }
}
```

# Sobrecarga de Valor de Retorno

```
void f() {}
int f() {}
```

Problemas:

- Dependente de Contexto
- Ambiguidade quando descarta retorno;

```
f( );
```

# Construtor Default

```
//ConstrutorDefault.java
class Passaro {
  int i;
}
public class ConstrutorDefault {
  public static void main(String[] args) {
    Passaro nc = new Passaro(); // default!
  }
}
```

# Problema com Construtor Default

```
class Arbusto {
  Arbusto(int i) {}
  Arbusto(double d) {}
}

new Arbusto();
```

# Objeto como Parâmetro

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

equivale semanticamente a

```
Banana.f(a,1);  // sintaxe nao e valida
Banana.f(b,2);  // sintaxe nao e valida
```

# this

```
//Folha.java
// Uso simples da palavra chave this.
public class Folha {
  int i = 0;
  Folha incremento() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Folha x = new Folha();
    x.incremento().incremento().incremento().print();
  }
}
```

# this como construtor

```
// Fabrica.java
// Chamando um construtor com "this."
public class Fabrica {
  int contFunc = 0;
  String s = new String("nula");
  Fabrica(int funcionarios) {
    contFunc = funcionarios;
    System.out.println("Fabrica(int): " + contFunc);
  }
  Fabrica(String ss) {
    System.out.println("Fabrica(String): " + ss);
    s = ss;
  }
```

# this como construtor

```
Fabrica(String s, int funcionarios) {
    this(funcionarios);
//! this(s);        // não pode chamar dois!
    this.s = s;     // Outro uso de "this"
    System.out.println("String & int args");
  }
  Fabrica() {
    this("ola", 47);
    System.out.println("construtor default");
  }
  void print() {
//! this(11); // Nao pode dentro de nao construtores!
    System.out.println(contFunc + " - " + s);
  }
  public static void main(String[] args) {
    Fabrica x = new Fabrica();
    x.print();
  }
}
```

# Remoção

- Coletor
  - Alocação tão rápida quanto pilha
  - Contagem de Referências
    - Referência Mútua
  - Caminhamento
  - Chamada explícita do Construtor
    - System.gc();

# Inicialização de Atributos

```
class Medida {
  Largura x = new Largura();
  boolean b = true;
  public Medida f ( ) { return new Medida( ) }
  public Medida g (Medida m) { return m; }
}
class MedidaI {
  Medida i = new Medida( );
  Medida j = i.f();
}
class MedidaII {
  Medida i = new Medida( );
  Medida j = i.f();
  Medida k = i.g(j);
}
class MedidaIII {
  Medida i = new Medida( );
  Medida k = i.g(j);
  Medida j = i.f();
}
```

# Ordem de Inicialização

```
//: OrdemDeInicializacao.java
// Demonstra a ordem de inicializacao.
class Rotulo {
  Rotulo(int marca) {
    System.out.println("Rotulo(" + marca + ")");
  }
}
class Cartao {
  Rotulo t1 = new rotulo(1);  // Antes do construtor
  Cartao() {
    System.out.println("Cartao()");
    t3 = new Rotulo(33); // Reinicializa t3
  }
  Rotulo t2 = new Rotulo(2);  // Depois do construtor
  void f() {
    System.out.println("f()");
  }
  Rotulo t3 = new Rotulo(3);  // no fim
}
public class OrdemDeInicializacao {
  public static void main(String[] args) {
    Cartao t = new Cartao();
    t.f(); // Mostra que construcao e feita  }
}
```

# Inicialização Estática

```
// InicializacaoEstatica.java
// Especificando valores iniciais em uma definicao
// de classe.
class Tigela {
  Tigela (int marca) {
    System.out.println("Tigela(" + marca + ")");
  }
  void f(int marca) {
    System.out.println("f(" + marca + ")");
  }
}
class Mesa {
  static Tigela b1 = new Tigela(1);
  Mesa() {
    System.out.println("Mesa()");
    b2.f(1);
  }
  void f2(int marca) {
    System.out.println("f2(" + marca + ")");
  }
  static Tigela b2 = new Tigela(2);
}
```

# Inicialização Estática

```
class Armario {
  Tigela b3 = new Tigela(3);
  static Tigela b4 = new Tigela(4);
  Armario() {
    System.out.println("Armario()");
    b4.f(2);
  }
  void f3(int marca) {
    System.out.println("f3(" + marca + ")");
  }
  static Tigela b5 = new Tigela(5);
}

public class InicializacaoEstatica {
  public static void main(String[] args) {
    System.out.println("Criando novo Armario()");
    new Armario();
    System.out.println("Criando novo Armario()");
    new Armario();
    t2.f2(1);
    t3.f3(1);
  }
  static Mesa t2 = new Mesa();
  static Armario t3 = new Armario();
}
```

# Inicialização Estática

```
Tigela(1)
Tigela(2)
Mesa()
f(1)
Tigela(4)
Tigela(5)
Tigela(3)
Armario()
f(2)
Criando novo Armario ()
Tigela(3)
Armario ()
f(2)
Criando novo Armario ()
Tigela(3)
Armario ()
f(2)
f2(1)
f3(1)
```

# Bloco de Inicialização Estático

```
//EstaticoExplicito.java
// Inicializacao estatico explicito
class Copo {
  Copo(int marca) {
    System.out.println("Copo(" + marca + ")");
  }
  void f(int marca) {
    System.out.println("f(" + marca + ")");
  }
}
class Copos {
  static Copo c1;
  static Copo c2;
  static {
    c1 = new Copo(1);
    c2 = new Copo(2);
  }
```

# Bloco de Inicialização Estático

```
  Copos() {
    System.out.println("Copos()");
  }
}
public class EstaticoExplicito {
  public static void main(String[] args) {
    System.out.println("Dentro do main()");
    Copos.c1.f(99);  // (1)
  }
  // static Copos x = new Copos();  // (2)
  // static Copos y = new Copos();  // (2)
}
```

# Bloco de Inicialização

```
//Carros.java
// Java "Inicializacao Instancia."
class Carro {
Carro(int marca) {
    System.out.println("Carro(" + marca + ")");
  }
  void f(int marca) {
    System.out.println("f(" + marca + ")");
  }
}
public class Carros {
  Carro c1;
  Carro c2;
  {
    c1 = new Carro(1);
    c2 = new Carro(2);
    System.out.println("c1 & c2 inicializados");
  }
  Carros() {
    System.out.println("Carros()");
  }
```

# Bloco de Inicialização

```
public static void main(String[] args) {
    System.out.println("Dentro de main()");
    Carros x = new Carros();
    Carros y = new Carros();
  }
}
```

# Declaração de Vetores

```
int[] a1;

int a1[];

int[] a1 = { 1, 2, 3, 4, 5 };

int[] a2;

a2 = a1;
```

# Inicialização de Vetores Primitivos

```
// Vetores.java
// Vetores de primitivos.
public class Vetores {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)  a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println("a1[" + i + "] = " + a1[i]);
  }
}
```

# Inicialização de Vetor com Tamanho Dinâmico

```
import java.util.*;
public class VetorNovo {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
    System.out.println("tamanho de a = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println("a[" + i + "] = " + a[i]);
  }
}
```

# Vetores de Objetos

```
// VetorClasseObj.java
// Criando um vetor de objetos nao primitivos.
import java.util.*;
public class VetorClasseObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pRand(20)];
    System.out.println("tamanho de a = " + a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(pRand(500));
      System.out.println("a[" + i + "] = " + a[i]);
    }
  }
}
```

# Inicialização de Vetores

```
public class InicVetor {
  public static void main(String[] args) {
    Integer[] a = {           // Java 1.0
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
    Integer[] b = new Integer[] {  // Java 1.1
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
}
```

# Autoboxing em J2SE 5.0

```
// AutoBox.java
public class AutoBox {
  public static void main(String[] args) {
    Integer[] a = new Integer[3];
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    int i = a[1];
    System.out.println("" + a[0] + a[1] + a[2]);
  }
}
```

# Vetores Multidimensionais

```
import java.util.*;
public class VetorMultiDim {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  static void prt(String s) {
    System.out.println(s);
  }
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
```

# Vetores Multidimensionais

```
for(int i = 0; i < a1.length; i++)
  for(int j = 0; j < a1[i].length; j++)
    prt("a1[" + i + "][" + j +
        "] = " + a1[i][j]);
// Vetor 3-D com tamanho fixo:
int[][][] a2 = new int[2][2][4];
for(int i = 0; i < a2.length; i++)
  for(int j = 0; j < a2[i].length; j++)
    for(int k = 0; k < a2[i][j].length; k++)
      prt("a2[" + i + "][" + j + "][" + k +
          "] = " + a2[i][j][k]);
```

# Vetores Multidimensionais

```
    // Vetor 3-D com vetores de tamanho variável:
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length; k++)
          prt("a3[" + i + "][" + j + "][" + k +
              "] = " + a3[i][j][k]);
  }
}
```

# Exemplo

- Programa para ler uma data no formato dd/mm/aaaa ou d/mm/aaaa ou dd/m/aaaa ou d/m/aaaa e dizer se a data é válida.

# Exemplo

```
public class DataValida {

  public static void main (String [] args) {
     byte dia, mes;
     short ano;
     System.out.print (
          "Entre com a data no formato dd/mm/aaaa: ");
     String d = Console.readString();
     byte i = 0, j = 0;
     byte tam = (byte) d.length();
     if (tam < 8 || tam > 10) {
       System.out.println (" A data " + d +
          " e' invalida! ");
       return;
```

# Solução

```
} else {
    if (tam == 9) {
       if (d.charAt(1) == '/') {
          i = 1;
       } else {
        j = 1;
       }
    } else {
      if (tam == 8) {
        i = 1;
        j = 1;
      }
    }
}
dia = (byte) Integer.parseInt(d.substring(0,2-i));
mes = (byte) Integer.parseInt(d.substring(3-i,5-i-j));
ano = (short) Integer.parseInt(
                    d.substring(6-i-j,10-i-j));
```

# Solução

```
        if ( (dia < 1 || dia > 31 || mes < 1 ||
              mes > 12 || ano < 1 || ano > 9999) ||
             (dia == 29 && mes == 2 &&
              !((ano - 2000) % 4 == 0))) ||
             (dia == 30 && mes == 2) ||
             (dia == 31 && (mes == 2 || mes == 4 ||
              mes == 6 || mes == 9 || mes == 11))
           ) {
             System.out.println (" A data " + d +
                  " e' invalida! ");
        } else {
             System.out.println (" A data " + d +
                  " e' valida! ");
        }
    }
}
```

# Exercício

Faça um programa para ler várias datas no formato dd/mm/aaaa ou d/mm/aaaa ou dd/m/aaaa ou d/m/aaaa e imprimi-las no formato ingles ?m/?d/aaaa. O programa deve se encerrar quando for digitada uma data inválida.

# Questão

Quanto foi possível reusar do primeiro programa no segundo?

# Usando OO

```
public class Data {
  byte dia, mes;
  short ano;

  public Data (String d) {
     byte i = 0, j = 0;
     byte tam = (byte) d.length();
     if (tam < 8 || tam > 10) {
       dia = 0;
       mes = 0;
       ano = 0;
       return;
     } else {
```

# Usando OO

```
        if (tam == 9) {
           if (d.charAt(1) == '/') {
              i = 1;
           } else {
            j = 1;
           }
        } else {
          if (tam == 8) {
            i = 1;
            j = 1;
          }
        }
      }
      dia = (byte) Integer.parseInt(d.substring(0,2-i));
      mes = (byte) Integer.parseInt(d.substring(3-i,5-i-j));
      ano = (short) Integer.parseInt(d.substring(6-i-j,
                                      10-i-j));
  }
```

# Usando OO

```
public boolean bissexto() {
   return (ano - 2000) % 4 == 0;
}
 public static Data leConsole() {
   System.out.print ("Entre com a data no formato " +
                     "dd/mm/aaaa: ");
   Data d = new Data(Console.readString());
   return d;
}
public String toString() {
   return dia + "/" + mes + "/" + ano;
}
```

# Usando OO

```
  public void imprimeFormatoIngles() {
     System.out.println (mes + "/" + dia + "/" + ano);
  }
  public boolean valida() {
     return !(
               (dia < 1 || dia > 31 || mes < 1 ||
                mes > 12 || ano < 1 || ano > 9999) ||
               (dia == 29 && mes == 2 && !bissexto()) ||
               (dia == 30 && mes == 2) ||
               (dia == 31 && (mes == 2 || mes == 4 ||
                mes == 6 || mes == 9 || mes == 11))
             );
  }
}
```

# Usando OO

```
public class DataValidaI {

  public static void main (String [] args) {
     Data d = Data.leConsole();
     if (d.valida()) {
          System.out.println (" A data " + d +
                              " e' valida! ");
     } else {
          System.out.println (" A data " + d +
                              " e' invalida! ");
     }
  }
}
```

# Usando OO

```
public class FormatoIngles {

  public static void main (String [] args) {
     Data d;
     for (;;) {
       d = Data.leConsole();
       if (!d.valida()) {
         break;
       }
       d.imprimeFormatoIngles();
     }
  }
}
```

# Questão

Caso a representação interna da classe Data fosse alterada para armazená-la apenas com o número de dias a que corresponde, o que deveria ser alterado nos programas usuários?

# Tipos Abstratos de Dados

- Métodos Inicializadores (Construtores)
- Métodos de Acesso
- Métodos Modificadores
- Métodos Analisadores
- Métodos Finalizadores (Destrutores)

# A Classe Data Completa

- Inicializadores (construtores)
  - string
  - default: inicializa com data corrente
  - dia, mês e ano
  - outra data
- Métodos Modificadores
  - modifica recebendo data
  - modifica recebendo dia, mes e ano
  - incrementa dia

# A Classe Data Completa

- Métodos de Acesso
  - retorna dia
  - retorna mês
  - retorna ano
  - dias do ano
  - converte dias
  - diferença dias
  - diferença em anos
  - imprime

# A Classe Data Completa

- Métodos Analisadores
  - igualdade de data
  - menor que data
  - igualdade de dia e mês
  - menor dia e mes que outra data
- Métodos finalizadores (destrutores)

# Pacotes

- Identificação das Classes Relacionadas
- Recuperação Fácil das Classes Relacionadas
- Evita Conflito de Nomes entre Classes de Diferentes Pacotes
- Estabelece uma Política de Acesso aos Atributos e Métodos das Classes

# Pacotes

```
package meupacote;
public class MinhaClasse {
. . .
}
// usando import
import meupacote.*;
MinhaClasse m = new MinhaClasse();

// mesma coisa sem import
meupacote.MinhaClasse m = new meupacote.MinhaClasse();

import meupacote.MinhaClasse;
```

# Pacotes e Diretórios

- Existe relação direta entre os nomes dos pacotes e a estrutura dos diretórios
  - cada pacote corresponde a um diretório
    - arquivos e classes de um mesmo pacote devem estar em um mesmo diretório
  - uso de . para indicar existência de subpacote
    - classes e arquivos do subpacote devem estar em subdiretório correspondente no diretório do pacote
    - uso de . como notação multiplataforma para definição de trilha de diretórios

# Pacote Padrão

- Classes criadas em arquivos que não especificam o pacote são colocadas no pacote padrão do diretório
- Uma vez criado um pacote no diretório, as classes deste pacote não terão acesso às classes pertencentes ao pacote padrão
- Classes do pacote padrão poderão se autoreferenciar
- Não se usa import para acessar classes do pacote padrão

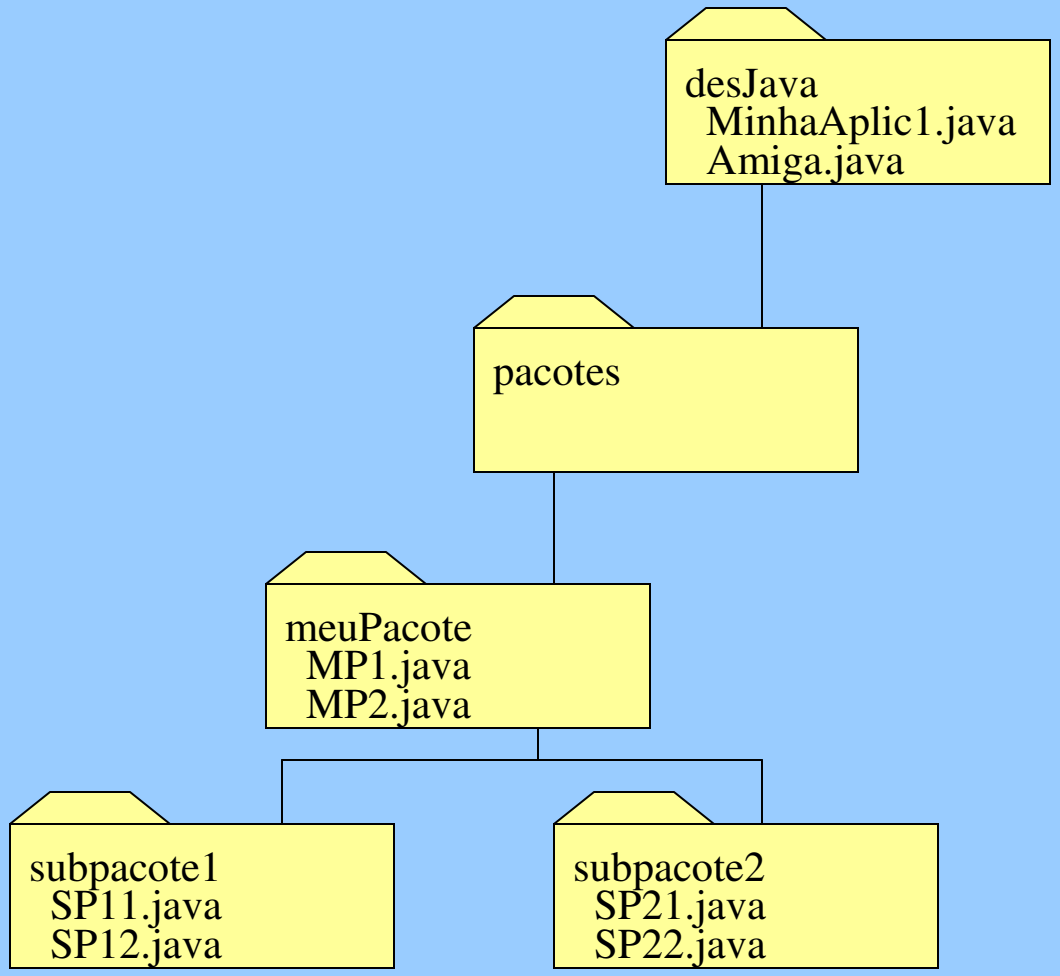
# Pacote Padrão

```
// Bolo.java
class Bolo {
 public static void main(String[] args) {
 Torta x = new Torta();
 x.f();
 }
}
// Torta.java
class Torta {
 void f() { System.out.println("Torta.f()"); }
}
```

# Exemplo de Pacotes e Diretórios

- Criação de pacote *meuPacote* com classes *MP1* e *MP2* e subpacotes *subpacote1* com classes *SP11* e *SP12* e *subpacote2* com classes *SP21* e *SP22*
- Colocá-lo dentro de diretório de pacotes em um diretório de desenvolvimento
- Criação de aplicação *MinhaAplic1* dentro do diretório de desenvolvimento

# Exemplo de Pacotes e Diretórios



# Exemplo de Pacotes e Diretórios

```
// SP11.java
package pacotes.meuPacote.subPacote1;

public class SP11 {
  int i;
}

// SP12.java
package pacotes.meuPacote.subPacote1;

public class SP12 {
  int j;
}
```

# Exemplo de Pacotes e Diretórios

```
// SP21.java
package pacotes.meuPacote.subPacote2;

public class SP21 {
  int k;
}

// SP22.java
package pacotes.meuPacote.subPacote2;

public class SP22 {
  int l;
}
```

# Exemplo de Pacotes e Diretórios

```
// MP1.java
package pacotes.meuPacote;

public class MP1 {
  int m;
}

// MP2.java
package pacotes.meuPacote;

public class MP2 {
  int n;
}
```

# Exemplo de Pacotes e Diretórios

```
class Amiga {
  int a;
}

// MinhaAplic1.java
import pacotes.meuPacote.*;
import pacotes.meuPacote.subPacote1.*;
import pacotes.meuPacote.subPacote2.*;
import java.util.*;
```

# Exemplo de Pacotes e Diretórios

```
public class MinhaAplic1 {
  public static void main(String [] args) {
    ArrayList al = new ArrayList();
    Amiga a = new Amiga();
    SP11 i = new SP11();
    SP12 j = new SP12();
    SP21 k = new SP21();
    SP22 l = new SP22();
    MP1 m = new MP1();
    MP2 n = new MP2();
  }
}
```

# Exemplo de Pacotes e Diretórios

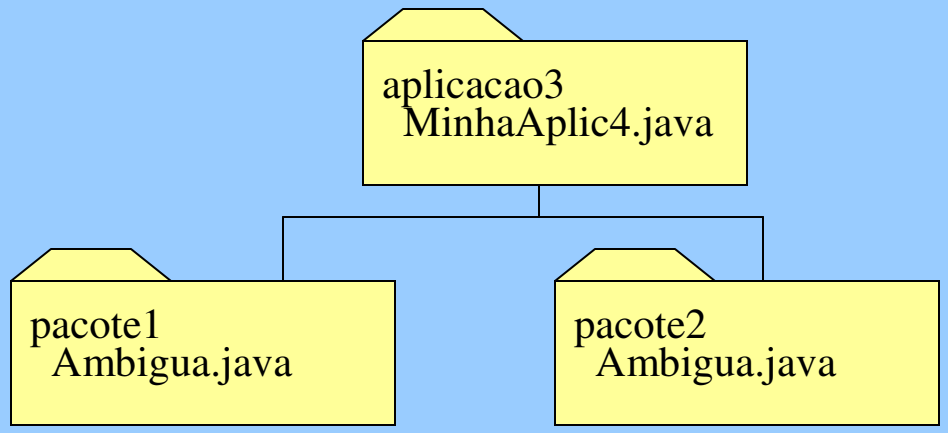
- Compilador e interpretador enxergam o que está no diretório corrente
  - Dentro do diretório *desJava*

    *javac MinhaAplic1.java*
  - Classes dos pacotes importados serão compiladas
  - Classe Amiga será compilada
  - Ainda dentro do diretório *desJava*

    *java MinhaAplic1*

# Erro de Ambigüidade

- Compilador identifica classes de mesmo nome importadas por pacotes diferentes e indica onde ocorreu a ambigüidade



# Erro de Ambigüidade

```
// Ambigua.java de pacote1
package pacote1;
public class Ambigua {
  int i;
  public Ambigua () {
   System.out.println("pacote1");
  }
}
```

```
// Ambigua.java de pacote2
package pacote2;
public class Ambigua {
  int i;
  public Ambigua () {
   System.out.println("pacote2");
  }
}
```

# Erro de Ambigüidade

```
// MinhaAplic4.java em aplicacao3
import pacote1.*;
import pacote2.*;
class MinhAplic4 {
  public static void main(String [] args) {
   Ambigua a = new Ambigua ();
  }
}
```

# Convenção para Nomes de Pacotes

- Uso de Pacotes obtidos na Internet pode gerar Conflito de Nomes
- Convenção proposta pelos projetistas de Java
  - Uso do domínio do criador do pacote como prefixo do pacote na ordem inversa
  - InterNIC garante que nomes de domínios são únicos
  - Dentro de um mesmo domínio devem ser estabelecidas convenções específicas

    *package br.ufes.inf.fvarejao.pacotes.meuPacote;*

# CLASSPATH

- Variável de Ambiente usada pelas Ferramentas do JDK para encontrar classes
- Default inclui diretório corrente e diretórios com bibliotecas padrão de classes de JAVA
- Trilha de diretórios definida no pacote importado é concatenada a todos as trilhas existentes na CLASSPATH

  Se ~ está no CLASSPATH ao compilar ou executar *MinhaAplic1*, as classes do pacote MeuPacote serão procuradas no diretório ~/*pacotes/meuPacote* e ./pacotes/MeuPacote

# Cuidados com CLASSPATH

- A busca pela classe pertencente a PACOTE PADRÃO será interrompida quando for encontrada a primeira classe com o nome da classe procurada - pode provocar erro difícil de encontrar
- Erro tende a ocorrer mais frequentemente quando a variável de Ambiente CLASSPATH é modificada permanentemente
- Recomenda-se definir CLASSPATH provisoriamente usando a opção -classpath das ferramentas do JDK
  - Diretório corrente deve ser incluído na opção -classpath

# Pacotes

- Necessário importar subpacotes mesmo que pacote tenha sido importado por completo
- Compilação Automática
  - Só .java
    - compila
  - Só .class
    - não compila
  - Se .java e .class
    - Verifica data e recompila se .java é mais recente que .class

# Membros Públicos

```
package sobremesa;
public class Biscoito {
 public Biscoito() {
  System.out.println("construtor do biscoito");
 }
 public void calorias() {
  System.out.println("MUITAS!");
 }
 void mordida() {
  System.out.println("mordida!");
 }
}
```

# Membros Públicos

```
// Jantar.java
// uso da biblioteca sobremesa
import sobremesa.*;
public class Jantar {
 public Jantar() {
  System.out.println("Construtor do Jantar");
 }
 public static void main(String[] args) {
  Biscoito x = new Biscoito();
  x.calorias();
  // x.mordida(); // não pode ser acessado!
 }
}
```

# Membros Privados

```
// Sundae.java
class Sundae {
 private Sundae() { }
 public static Sundae fazerUmSundae() {
  return new Sundae();
 }
}
// Sorvete.java
public class Sorvete {
 public static void main(String[] args) {
  //! Sundae x = new Sundae();
  Sundae x = Sundae.fazerUmSundae();
 }
}
```

# Membros Amigos

```
// Chocolate.java
package sobremesa;
public class Chocolate {
 public Chocolate() { }
 void esquentarChocolate() {
  System.out.println("Esquentando chocolate...");
 }
}
// Brigadeiro.java
package sobremesa;
public class Brigadeiro {
 public Brigadeiro() { }
 void fazerBrigadeiro() {
  Chocolate choc = new Chocolate();
  choc.esquentarChocolate(); // amigo
  System.out.println("Fazendo Brigadeiro...");
 }
}
```

# Membros Protegidos

```
// Biscoito.java
package sobremesa;
public class Biscoito {
 public Biscoito() {
  System.out.println("construtor do biscoito");
 }
 protected void mordida() {
  System.out.println("mordida!");
 }
}
```

# Membros Protegidos

```
// BiscoitoMaizena.java
package doce;
import sobremesa.*;
public class BiscoitoMaizena extends Biscoito {
 public BiscoitoMaizena () {
  System.out.println("construtor de maizena");
 }
 public static void main(String[] args) {
  BiscoitoMaizena x = new BiscoitoMaizena();
  x.mordida(); // pode ser acessado!
 }
}
```

# Especificadores de Acesso

| Acesso | Público<br>*public* | Protegido<br>*protected* | Amigo<br>- | Privado<br>*private* |
|---|---|---|---|---|
| Mesma Classe | Sim | Sim | Sim | Sim |
| Classes no Mesmo Pacote | Sim | Sim | Sim | Não |
| Subclasses em Pacotes Diferentes | Sim | Sim | Não | Não |
| Não Subclasses em Pacotes Diferentes | Sim | Não | Não | Não |

# Ocultamento de Informação

```
import java.util.*;
public class Pilha {
 private Vector estrutura;
 public Pilha() {
  estrutura = new Vector(10,10);
 }
 public void empilha(Object obj) {
  estrutura.add(obj);
 }
 public Object desempilha() {
  Object obj = estrutura.get(estrutura.size()-1);
  estrutura.remove(estrutura.size()-1);
  return obj;
 }
}
```

# Ocultamento de Informação

- Recomendação Básica
  - Implementação
    - Atributos *private*
    - Métodos *private*
  - Interface
    - Métodos públicos
- Atributos amigos podem ser usados quando implementação de pacote é bem controlada

# Alteração da Implementação

```
import java.util.*;
public class Pilha {
 private LinkedList estrutura;
 public Pilha() {
   estrutura = new LinkedList();
 }
 public void empilha(Object obj) {
   estrutura.addFirst(obj);
 }
 public Object desempilha() {
   return estrutura.removeFirst();
 }
}
```

# Acesso às Classes

```
package sobremesa;
public class Biscoito {
```

```
import sobremesa.Biscoito;
```

ou

```
import sobremesa.*;
```

# Acesso às Classes

- Acesso
  - Públicas
    - Qualquer pacote tem acesso
  - Amigas (não públicas)
    - Somente classes do mesmo pacote
- Máximo de uma classe pública por arquivo fonte (é possível não ter nenhuma)
- Nome do Arquivo = Nome da Classe Pública
- Se arquivo não tem classe pública pode ter nome qualquer
  - Compilador só encontra classe se já tiver sido compilada