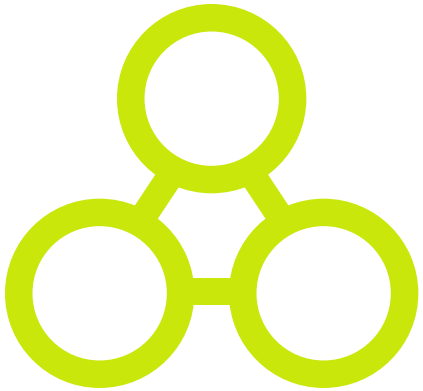


nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

4 – Classes e Objetos

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

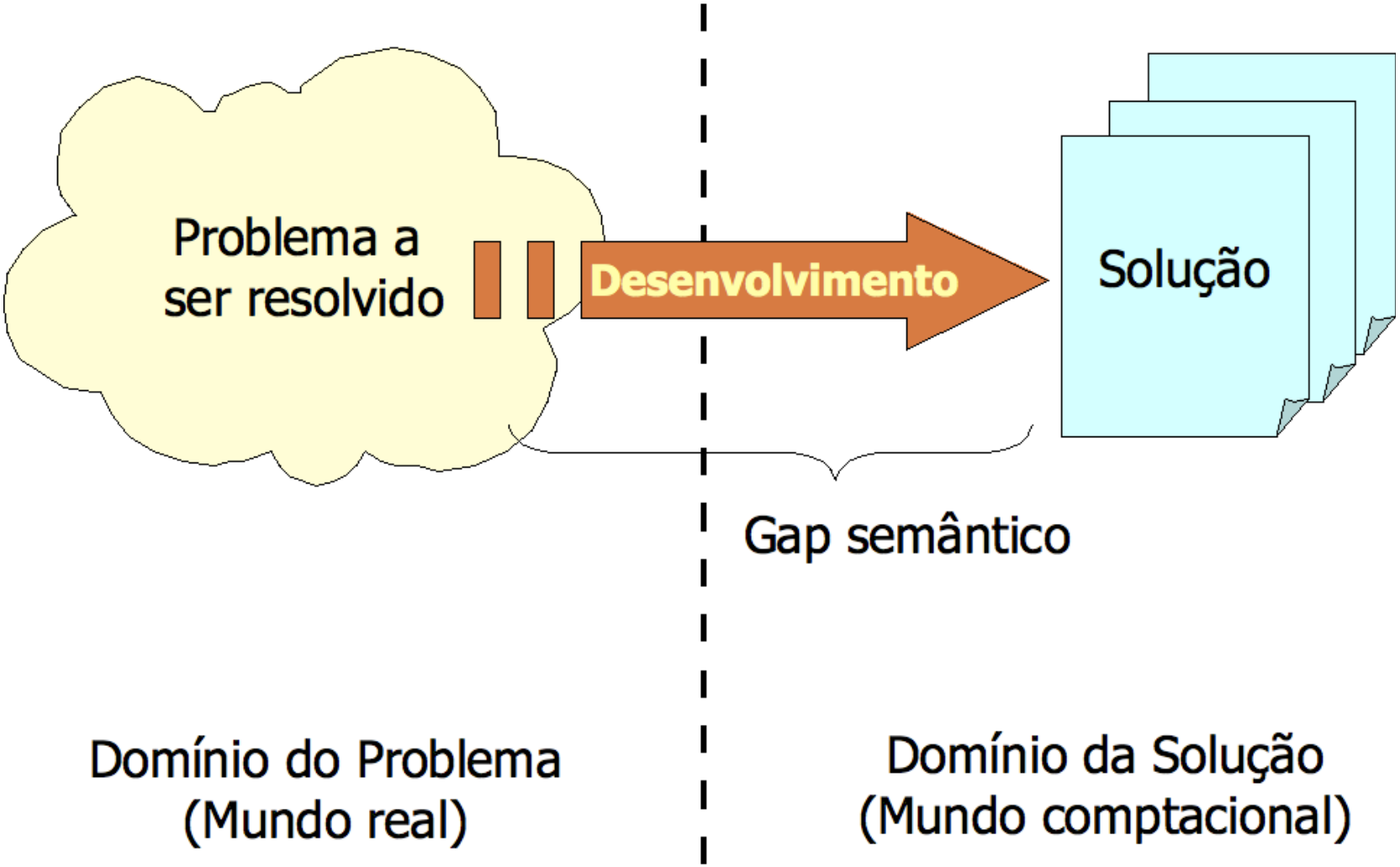
Centro Tecnológico

Universidade Federal do Espírito Santo

Objetivos da parte 4

- Apresentar a forma que Java trabalha com os conceitos da orientação a objetos:
  - Classes, objetos, atributos, métodos, escopo, pacotes, visibilidade, etc.
- Desta forma:
  - Capacitar os alunos a escrever programas realmente orientadas a objeto, utilizando todas as vantagens deste novo paradigma;
  - Estabelecer a fundação de todo o restante do curso: Java como uma linguagem de programação OO.

Gap semântico



Orientação a objetos

- Nossa mente é orientada a objetos;
- Se nossa LP também for OO, podemos diminuir o *gap* semântico:
  - Os recursos oferecidos são os mesmos, porém os mecanismos de abstração são mais poderosos;
  - Passamos a construir a solução em termos do domínio do problema (mundo real – objetos);
  - Usamos também objetos que não são parte do domínio do problema.

O que é um objeto?

- Três características principais:
  - Algo que possa atuar como um todo (identidade);
  - Algo que possua propriedades (estado);
  - Algo que possa realizar coisas e ter coisas realizadas para ele (comportamento).
- Ex.: uma conta bancária. . .
  - possui uma identificação única e pode ser considerada como um “todo”;
  - possui saldo, nome do proprietário, etc.;
  - pode sofrer depósito, efetuar saque, ser encerrada, etc.

Linguagem OO

- Em uma linguagem OO pura:
  - Tudo que existe é um objeto;
  - Um programa é um grupo de objetos dizendo uns aos outros o que fazer por meio de mensagens;
  - Cada objeto tem sua própria memória feita de outros objetos;
  - Todo objeto tem um tipo (uma classe);
  - Todos os objetos de um tipo (classe) particular podem receber as mesmas mensagens.

Conceitos OO fundamentais

- Abstração:
  - Desprezar conceitos irrelevantes.
- Encapsulamento:
  - Separar a interface da implementação.
- Modularidade:
  - Agrupar objetos em módulos coesos independentes.
- Hierarquia:
  - Organizar abstrações em hierarquias quando necessário.

Classificação de objetos

- Desde Aristóteles que o ser humano classifica os objetos do mundo;
- Juntamos objetos com mesmas características em categorias que chamamos de “classes”:
  - Todas as contas de banco tem um saldo, mas cada conta pode ter um saldo diferente;
  - Todas as contas de banco podem sofrer depósitos ou serem encerradas.
- Classes são usadas por linguagens OO para modelar tipos compostos. São modelos abstratos que definem os objetos da classe.

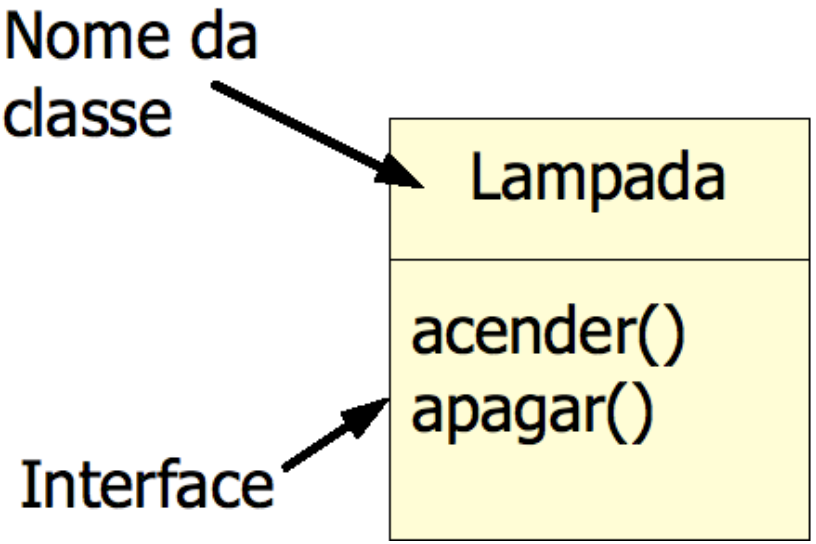
Instâncias de classes

- Objetos são instâncias de classe;
- Para criar um novo objeto, devemos especificar de qual classe;
- Podemos criar quantos objetos quisermos de uma determinada classe;
- Enviamos mensagens aos objetos (chamamos operações) para resolver nosso problema;
  - Que tipo de mensagens um determinado objeto aceita?

Um objeto tem uma interface

- A interface define o que um objeto pode fazer:

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



Um objeto tem uma implementação

- Cada mensagem que o objeto pode aceitar deve ter um código de implementação associado.

```
public class Lampada {
  public void acender() {
     /* Implementação... */
  }
}
```

Envio de mensagens

- Objetos enviam mensagens a outros objetos.

Definição de **l**, uma variável que é do tipo referência para **Lampada**.

Criação de um novo objeto da classe **Lampada** e atribuição de sua referência à variável **l**.

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

Envio de mensagem (chamada de operação) "acender" para o objeto referenciado por **l**. É executado o código referente à operação acender da classe **Lampada**.

A implementação deve estar oculta

- Podemos classificar programadores em:
  - Fabricantes de classes;
  - Clientes de classes.
- Motivos para ocultar a implementação:
  - Somente aparece para o cliente o que interessa: a interface;
  - A implementação pode ser alterada pelo fabricante sem que o código do cliente tenha que ser mudado.
- Java usa especificadores de acesso.

Abstração

Encapsulamento

Classes devem ser reutilizáveis

- Reuso é uma das grandes vantagens da OO;
- Idealmente, nossas classes devem ser reutilizáveis;
- Formas de reutilização:
  - Criar novas classes usando as existentes como atributos (composição – relacionamento “tem um”);
  - Criar novas classes estendendo classes existentes (herança – relacionamento “é um [subtipo de]”).

Hierarquia

Modularização

Criação de objetos

- Objetos são criados com o operador new:
  - Cria o objeto na memória (monte/heap);
  - Retorna uma referência ao objeto criado.
- Construtores:
  - Métodos especiais que executam durante a criação;
  - Podem especificar valores iniciais aos atributos.
- Inicialização:
  - Atributos são "zerados" quando um objeto é construído;
  - Podem também ser declarados com valores iniciais.

Destruição de objetos

- Um objeto é alocado dinamicamente;
- Qual é o tempo de vida de um objeto?
- Antes: qual é o tempo de vida de uma variável?

```
{
  int x = 12;
  // x está disponível.
  {
     int q = 96;
     // x e q estão disponíveis.
  }
  // x está disponível, q fora de escopo.
}
// x e q fora de escopo.
```

Destruição de objetos

- Uma variável é destruída quando acaba seu escopo (funcionamento da pilha);
- E quando um objeto é destruído?

```
{
  String s;
  // referência s disponível.
  {
     String r = new String("Olá!");
     // s, r, objeto disponíveis.
     s = r;
  }
  // r não existe mais. Destruir o objeto?
}
```

Destruição de objetos

- Ao contrário das variáveis, um objeto não é destruído quando o escopo acaba;
- Um objeto é destruído automaticamente pelo Coletor de Lixo (*Garbage Collector* – *GC*) quando ele se torna inacessível;
- Um objeto é inacessível quando não há referências para ele na pilha.

```
{
  String s = new String("Olá!");
}
// Impossível acessar a string "Olá!"
```

Execução manual do coletor de lixo

- Podemos pedir para o coletor de lixo executar:

```
/* Código. */

System.gc();

/* Mais código. */
```

- Não há garantias que ele realmente vá executar;
- É útil quando estamos prestes a iniciar um trecho durante o qual não queremos o GC funcionando.

Definição de uma classe

- Uso da palavra reservada `class`;
- Significado: “segue abaixo a especificação de como objetos deste tipo devem se comportar”;

```
class NomeDaClasse {
  /* Especificação da classe vai aqui. */
}
```

- Depois de definida a classe, podemos definir variáveis (referências) e criar objetos:

```
NomeDaClasse obj = new NomeDaClasse();
```

Membros da classe

- Uma classe pode ter dois tipos de membro:
  - Variáveis (em jargão OO: “atributos”);
  - Funções (em jargão OO: “métodos”).
- Atributos são como partes de um tipo composto;
- Métodos são funções que são executadas no contexto de uma classe/objeto.

Atributos

- Definidos como variáveis no escopo da classe:

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

- Acesso via operador de seleção (“.”):

```
SoDados d = new SoDados();
d.i = 47;
d.f = 1.1;
d.b = false;
```

Atributos

- Atributos podem ser referências a objetos:

```
class OutraClasse {
  String s;
  SoDados dados;
}
```

- Operadores de seleção conectados:

```
OutraClasse oc = new OutraClasse();
oc.s = "Olá!";
oc.dados = new SoDados();
oc.dados.i = 47;
oc.dados = d;      // d criado no slide anterior!
```

Valores *default* para atributos

- Um atributo pode ser inicializado:

```
class UmDado {
  int i = 150;
}
```

- Quando não inicializamos explicitamente, um valor *default* é atribuído a ele:

Tipo	Valor
boolean	false
char	'\u0000'
byte	(byte) 0
short	(short) 0

Tipo	Valor
int	0
long	0l
float	0.0f
double	0.0

O valor null

- O valor *default* para atributos referência (objetos) é `null`;
- Um "objeto nulo" é uma referência que não aponta para nenhum objeto;
- Usar uma referência nula como se ela apontasse para um objeto causa `NullPointerException`.

```
UmDado d = null;
System.out.println(d);                    // OK!
System.out.println(d.i);                  // NPE!
```

Valores *default* para atributos

- O valor *default* pode não ser o valor correto ou mesmo um valor legal dentro da sua lógica;
- Portanto, a boa prática é sempre usar a inicialização explícita;
- Variáveis locais não são "zeradas" automaticamente e geram erros de compilação se utilizadas sem valor:

```
int i;

// variable i might not have been initialized
System.out.println(i);
```

Métodos

- Um método é uma função que opera no contexto de uma classe (mensagem que o objeto recebe):

```
class UmDado {
  int i = 150;
  void imprimir() {
     System.out.println(i);
  }
}

/* Em outro ponto do código... */
UmDado ud = new UmDado();
ud.imprimir(); // 150
ud.i = 300;
ud.imprimir(); // 300
```

Parâmetros e valor de retorno

- Um método pode aceitar parâmetros:

```
public class ObjetoX {
  void imprimir(String s) {
     System.out.println(s);
  }
}
```

- Um método pode retornar um valor:

```
public class ObjetoX {
  float quadrado(float f) {
     return f * f;
  }
}
```

Definição de um método

- Para definir um método, especificamos (no mínimo):
  - Seu tipo de retorno;
  - Seu nome;
  - Seus parâmetros;
  - Sua implementação.

```
tipoRetorno nome(/* Lista de parâmetros */) {
  /* Corpo (implementação) do método. */
}
```

Assinatura de um método

- Define o método de uma forma única;
- Em Java, o nome e os tipos de parâmetros de um método formam sua assinatura;
- Não pode haver dois métodos com a mesma assinatura na mesma classe (mesmo que o tipo de retorno seja diferente);
- Algumas linguagens incluem o tipo de retorno na assinatura. Java não o faz.

Chamada de método

- Métodos são chamados (“mensagens são enviadas”) usando o operador de seleção (“ . ”):

```
UmDado ud = new UmDado();
UmDado ud2 = new UmDado();
ud2.i = 300;

ud.imprimir();  // 150
ud2.imprimir(); // 300

// Passando argumentos e retornando valores:
ObjetoX x = new ObjetoX();
float quad = x.quadrado(ud.i);
x.imprimir("i^2 = " + quad);
```

Parâmetros

- Especificados na assinatura do método;
- Cada um tem um tipo (primitivo ou objeto):

```
class ObjetoX {
  double calcula(float f, int i, double d) {
     d = (f / i) * d;
     return d;
  }

  void imprimeDobro(UmDado dado) {
     dado.i = dado.i * 2;
     dado.imprimir();
  }
}
```

Passagem de parâmetros

- Funciona como uma atribuição:
  - Tipos devem ser compatíveis;
  - Tipos primitivos recebem cópias dos valores;
  - Tipos referência recebem cópias das referências.

```
ObjetoX x = new ObjetoX();
double d = 10, e = x.calcula(4.0f, 2, d);
System.out.println(e);  // 20.0
System.out.println(d);  // 10.0

UmDado dado = new UmDado();
dado.i = 100;
x.imprimeDobro(dado);           // 200
dado.imprimir();                // 200
```

Retorno

- Já pudemos observar que um método pode retornar um valor usando o comando return:

```
class ObjetoX {
  int calcularMemoriaArmazenada(String s) {
    return s.length() * 2;
  }
}
```

- Em métodos que não retornam valores (void), return (opcional) indica que o método deve ser interrompido.

Parâmetros variáveis

- A partir do Java 5 se tornou possível definir métodos que recebem um número variável de argumentos (*varargs*):

```
public class Teste {
  void print(boolean msg, String ... objs) {
     if (msg) System.out.println("Args:");
     for (int i = 0; i < objs.length; i++)
        System.out.println(objs[i]);
  }
  public static void main(String[] args) {
     Teste t = new Teste();
     t.print(true, "Java", "Sun", "JCP");
  }
}
```

Parâmetros variáveis

- Só pode haver uma lista de parâmetros variáveis na declaração do método;
- Deve ser a última a ser declarada;
- Funciona como um vetor do tipo declarado (no exemplo, vetor de `String`);
- Não há limite para o número de parâmetros;
- Também aceita zero parâmetros.

```
t.print(false, "A", "B", "C", "D", "E");
t.print(true, "Um", "Dois");
t.print(false);
```

Atributos e métodos de classe

- Vimos até agora que atributos e métodos pertencem aos objetos:
  - Não se faz nada sem antes criar um objeto (new)!
- No entanto, há situações que você quer usá-los sem ter que criar objetos:
  - Deseja-se um atributo associado a uma classe como um todo (todos os objetos compartilham a mesma variável, similar a uma "variável global");
  - Deseja-se chamar um método mesmo que não haja objetos daquela classe criados.

Atributos e métodos `static`

- Usando a palavra-chave `static` você define um atributo ou método de classe ("estático"):
  - Atributo/método pertence à classe como um todo;
  - Pode-se acessá-los mesmo sem ter criado um objeto;
  - Objetos podem acessá-los como se fosse um membro de objeto, só que compartilhado;
  - O contrário não é possível: métodos `static` não podem acessar atributos/métodos não-`static` diretamente (precisa criar um objeto).

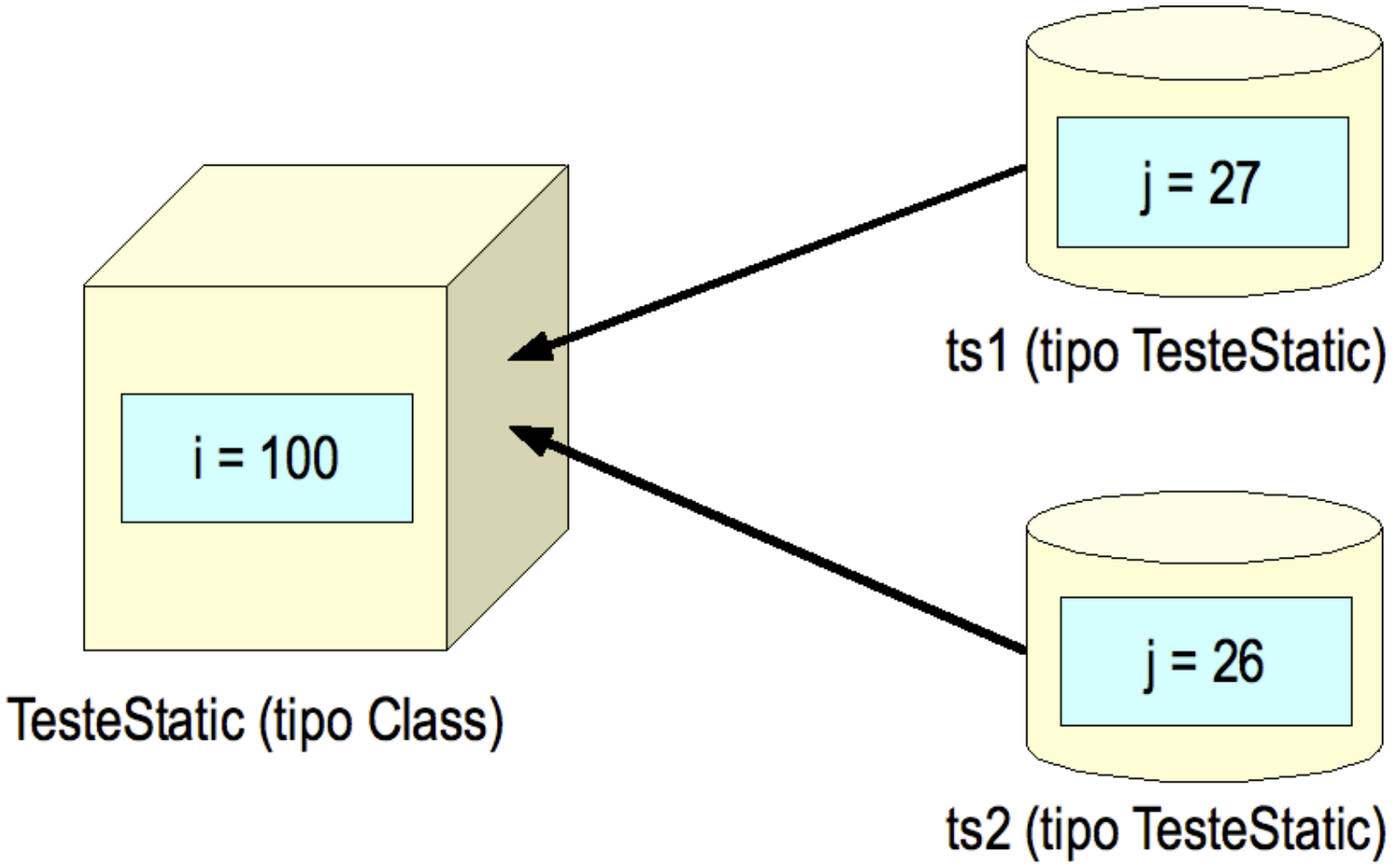
Atributos de classe (“estáticos”)

```
public class TesteStatic {
  static int i = 47;

  int j = 26;

  public static void main(String[] args) {
     TesteStatic ts1 = new TesteStatic();
     TesteStatic ts2 = new TesteStatic();

     // 47 26
     System.out.println(ts1.i + " " + ts1.j);

     // 47 26
     System.out.println(ts2.i + " " + ts2.j);

     /* Continua... */
```

Atributos de classe (“estáticos”)

```
    /* Continuação... */

    ts1.i++;
    ts1.j++;

    // 48 27
    System.out.println(ts1.i + " " + ts1.j);

    // 48 26
    System.out.println(ts2.i + " " + ts2.j);

    TesteStatic.i = 100;
    System.out.println(ts1.i);      // 100
    System.out.println(ts2.i);      // 100
  }
}
```

Armazenamento em memória



Métodos de classe (“estáticos”)

```
public class TesteStatic {
  static int i = 47;
  int j = 26;

  static void imprime(String s) {
    System.out.println(s);
  }

  static void incrementaI() { i++; }

  void incrementaJ() { j++; }

  public static void main(String[] args) {
    TesteStatic ts1 = new TesteStatic();

    /* Continua... */
```

Métodos de classe (“estáticos”)

```
    /* Continuação... */

    incrementaI();                      // OK
    TesteStatic.incrementaI();          // OK
    ts1.incrementaI();                  // OK

    // incrementaJ(); causa erro!
    // TesteStatic.incrementaJ() também!

    ts1.incrementaJ();                  // OK

    // 50 27
    imprime(ts1.i + " " + ts1.j);
  }
}
```

Armazenamento em memória

- Todos os métodos, estáticos ou não, são armazenados na área de código da classe;
- A única diferença é que métodos estáticos podem ser chamados independente de objetos criados;
- Isso é essencial para o método `main()`!

(Quase) Tudo é objeto!

- Em Java, atributos e métodos podem ser definidos somente como parte de uma classe:
  - Podem pertencer a um objeto da classe;
  - Podem pertencer à classe como um todo.
  - Em Java, com exceção dos tipos primitivos, tudo é objeto.

Vetores são objetos!

- Praticamente todas as LPs dão suporte a vetores;
- Java dá suporte a vetores, que são objetos que recebem tratamento especial:
  - Declaração diferenciada (de outros objetos);
  - Inicialização automática (valores *default*);
  - Checagem de limites.

Declarando vetores

- Utiliza-se o operador de indexação `[ ]`:

```
int[] vetor1;          // Sintaxe preferida.
String vetor2[];       // Sintaxe C...
```

- Como todo objeto, declaramos acima somente referências a objetos vetores. Para criá-los:

```
vetor1 = new int[10];     // 10 inteiros.
vetor2 = new String[50]; // 50 strings.
```

- Automaticamente, todos os elementos do vetor recebem valores *default* (0, `false` ou `null`).

Declarando vetores

- Podemos inicializar os valores do vetor:

```
int[] v3 = new int[] {1, 2, 3, 4, 5};
int[] v4 = {1, 2, 3, 4, 5};   // Simplificado
```

- *Arrays* possuem um atributo especial chamado `length`, que indica o tamanho do vetor;
- Elementos do vetor são individualizados por meio do operador de indexação;
- O primeiro elemento está no índice 0:

```
// 1, 2, 3, 4, 5,
for (int i = 0; i < v4.length; i++)
  System.out.print(v4[i] + ", ");
```

Verificação de índice

- Quando acessamos vetores, Java (ao contrário de C) verifica se o índice é válido;
- No caso de índices inválidos, ocorre um erro (exceção) em tempo de execução:

```
// 1, 2, 3, 4, 5, ERRO!
int[] v4 = {1, 2, 3, 4, 5};
for (int i = 0; i <= v4.length; i++)
  System.out.print(v4[i] + ", ");
```

Vetores multidimensionais

- Podemos criar vetores de múltiplas dimensões:

```
float[][] matriz = new float[5][6];
long[][] m2 = {
                  {1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9}
              };

for (int i = 0; i < m2.length; i++) {
  for (int j = 0; j < m2[i].length; j++) {
     System.out.println(m2[i][j]);
  }
}
```

Classes envoltório (*wrappers*)

- Em algumas situações, não podemos usar tipos primitivos:
  - Ex.: as classes utilitárias de coleção (lista, conjunto, etc.) são coleções genéricas de objetos.
- Java provê uma "classe envoltório" (*wrapper class*) para cada tipo primitivo;
- Tais classes só servem para armazenar um valor (imutável) de algum tipo primitivo.

Classes envoltório (*wrappers*)

- Todas pertencem ao pacote `java.lang`.

Primitivo	Wrapper
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`

Primitivo	Wrapper
`long`	`Long`
`float`	`Float`
`double`	`Double`
`void`	`Void`

Classes envoltório - uso

```
Integer wi = new Integer(10);
int i = wi.intValue();

boolean b = false;
Boolean wb = new Boolean(! b);
b = wb.booleanValue();

// "Encaixotamento"
Double wd = new Double(4.45e18);

// "Desencaixotamento"
double d = wd.doubleValue();
```

Autoboxing (Java 5)

- (Des)Encaixotamento automático;
- Java converte do tipo primitivo para o objeto envoltório automaticamente e vice-versa.

```
Integer[] vetor = new Integer[5];
vetor[0] = new Integer(10);

// Encaixotamento automático:
vetor[1] = 20;

// Desencaixotamento automático:
int i = vetor[0];
```

Recapitulando...

- O que já aprendemos:
  - Definimos as classes com a palavra-chave `class`;
  - Classes possuem atributos e métodos, que podem pertencer ao objeto ou à classe como um todo;
  - Atributos são inicializados com valores *default*;
  - Enviamos mensagens a objetos chamando seus métodos, que recebem parâmetros;
  - Vetores são objetos especiais (bem como as *strings*);
  - Java possui classes *wrapper* e *autoboxing*.
- Prosseguindo: como é o processo de construção de objetos?

Inicialização

- Neologismo criado para indicar tarefas que devem ser efetuadas ao iniciarmos algo;
- Quando criamos objetos, podemos querer inicializá-lo com alguns valores;
- Poderíamos criar um método para isso:

```
class Aleatorio {
  int numero;
  void inicializar() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}
```

Construtores

- Problema do método `inicializar()`: podemos esquecer de chamá-lo!
- Por isso, Java provê o mecanismo de construtores:
  - Métodos especiais chamados automaticamente pelo Java quando um objeto novo é criado;
  - Construtores são métodos sem valor de retorno e que possuem o mesmo nome da classe.

Construtores

- Quando um novo objeto é criado:
  1. é alocada memória para o objeto;
  2. o construtor é chamado.

```
class Aleatorio {
  int numero;
  Aleatorio() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
  public static void main(String[] args) {
     Aleatorio aleat = new Aleatorio();
  }
}
```

Construtores podem ter argumentos

- Se definidos argumentos, devem ser passados na criação do objeto com new:

```
class Aleatorio {
  int numero;
  Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
  public static void main(String[] args) {
     Aleatorio aleat1 = new Aleatorio(20);
     Aleatorio aleat2 = new Aleatorio(50);
  }
}
```

Pode haver múltiplos construtores

- Nossas classes podem ter quantos construtores quisermos (com assinaturas diferentes):

```
class Aleatorio {
  int numero;
  Aleatorio() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
  Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
  public static void main(String[] args) {
     Aleatorio aleat1 = new Aleatorio();
     Aleatorio aleat2 = new Aleatorio(50);
  }
}
```

Entendendo a sobrecarga

- Quando temos vários métodos com mesmo nome, dizemos que estamos sobrecarregando aquele nome;
- É útil para evitar redundâncias:
  - “lave o carro”, “lave a camisa”, “lave o cachorro”;
  - “laveCarro o carro”, “laveCamisa a camisa”, “laveCachorro o cachorro”.
- Fizemos isso quando definimos mais de um construtor para nossa classe!
- Podemos usar este conceito para qualquer método.

Sobrecarga

```
public class Quadrado {
  public static long calcular(long x) {
     return x * x;
  }
  public static double calcular(double x) {
     return x * x;
  }

  public static void main(String[] args) {
     long l = Quadrado.calcular(45);
     double d = Quadrado.calcular(3.87492);
  }
}
```

Distinção entre métodos sobrecarregados

- Como Java distingue entre dois métodos com o mesmo nome?
  - Pelos tipos dos parâmetros!
  - Já vimos que não podemos ter dois métodos com mesma assinatura, ou seja, mesmo nome e mesmos tipos de parâmetros;
  - A ordem dos tipos de parâmetro influi:

```
/* OK! */
long multiplicar(long x, int y) { /* ... */ }
long multiplicar(int x, long y) { /* ... */ }
```

Perigos da sobrecarga

- Devemos ter cuidado ao usar sobrecarga em duas situações:
  - Tipos primitivos numéricos, que podem ser convertidos;
  - Classes que participam de uma hierarquia com polimorfismo.

Sobrecarga de valor de retorno?

- Porque o valor de retorno de um método não é incluído em sua assinatura?

```
public class SobreRetorno {
  static int retorna10() { return 10; }
  static double retorna10() { return 10.0; }

  public static void main(String[] args) {
     int x = retorna10();       // OK!
     double d = retorna10();    // OK!

     // Qual método chamar?
     System.out.println(retorna10());
  }
}
```

Construtor *default*

- Quando não especificamos construtores, Java provê um construtor *default* para nossa classe:
  - Toda classe precisa de um construtor. Se você não escreveu nenhum, Java provê um pra você;
  - Sem parâmetros e sem implementação.
- Quando especificamos construtores, o construtor *default* não é provido automaticamente:
  - Se você escreveu um construtor, Java assume que você sabe o que está fazendo e não provê um;
  - Chamar o construtor sem o parâmetro gera erro se ele não for definido explicitamente.

Chamando métodos em objetos

- Já vimos que o código compilado dos métodos fica na área de memória da classe;
- Sendo assim, como Java sabe em qual objeto estou chamando um determinado método:

```
class Num {
  int i = 5;
  void somar(int j) { i += j; }
}
public class Teste
  public static void main(String[] args) {
     Num m = new Num(), n = new Num();
     m.somar(10); n.somar(5);
  }
}
```

Chamando métodos em objetos

- Internamente é como se o método fosse:

```
// Funcionamento interno! Não escreva assim!
static void somar(Num this, int j) {
  this.i += j;
}
```

- E a chamada fosse:

```
// Funcionamento interno! Não escreva assim!
Num.somar(m, 10);
Num.somar(n, 5);
```

A palavra reservada this

- Java faz esta transformação para você, de forma que o objeto que "recebeu a mensagem" está disponível pela palavra-chave this:

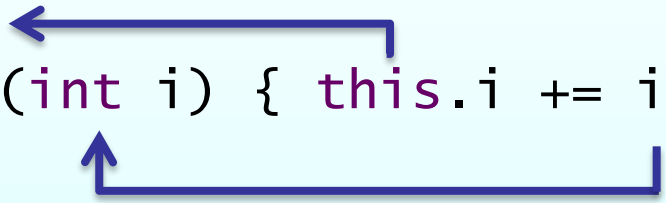
```
class Num {
  int i = 5;
  void somar(int j) { this.i += j; }
}
```

- Não é necessário usar this quando acessamos membros do objeto de dentro do mesmo (como no exemplo acima).

A palavra reservada this

- this pode ser usado para diferenciar um atributo do objeto de um parâmetro do método:

```
class Num {
  int i = 5;
  void somar(int i) { this.i += i; }
}
```



- Neste caso, o this é necessário!

A palavra reservada this

```
class Num {
  int i = 5;
  Num somar(int j) {
     i += j;
     return this;  // Aqui, this é útil!
  }
}
public class Teste
  public static void main(String[] args) {
     Num m = new Num();
     m.somar(10).somar(5).somar(1);
     System.out.println(m.i); // 21
  }
}
```

Construtores chamando construtores

- Relembrando exemplo da classe `Aleatorio`, seria interessante não haver duplicação de código:

```
class Aleatorio {
  int numero;
  Aleatorio() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
  Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}
```

Construtores chamando construtores

- Usamos novamente a palavra-chave `this`, com outro significado: chamar outro construtor:

```
class Aleatorio {
  int numero;
  Aleatorio() {
     // Chama o outro construtor com argumento 20.
     this(20);
  }
  Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}
```

Construtores chamando construtores

- A chamada `this()`:
  - Deve especificar os argumentos do construtor a ser chamado;
  - Deve ser a primeira linha do construtor que a utiliza;
  - Não pode ser usada fora de construtores.

this e static

- Métodos static não possuem this:
- Métodos estáticos pertencem à classe como um todo;
- this representa o objeto corrente;
- Em métodos de classe, não há objeto corrente!

```
// Método estático:
TesteStatic.incrementaI();

// Mesmo neste caso não existe this:
ts1.incrementaI();
```

Ordem de construção

- Quando construímos um novo objeto, os seguintes passos são efetuados:
  1. Os atributos da classe são zerados (inicializados com seus valores *default*: 0, `false` ou `null`);
  2. Se a classe participa de alguma hierarquia*, o construtor da superclasse é chamado;
  3. Os atributos da classe recebem os seus valores iniciais (se especificados), na ordem em que foram definidos na classe;
  4. O corpo do construtor da classe é chamado.

* Veremos mais sobre herança em outro capítulo.

Ordem de construção

- Atenção à ordem de construção!

```
class Medida {
  Largura x = new Largura();
  boolean b = true;
  public Medida f() { return new Medida(); }
}
class MedidaI {
  Medida i = new Medida();
  Medida j = i.f();
}
class MedidaII {
  Medida j = i.f();              // Erro!
  Medida i = new Medida();
}
```

Inicialização de atributos `static`

- Atributos estáticos são inicializados somente quando a classe é usada pela primeira vez;
  - Se a classe não for usada, não são inicializados.
- São inicializados antes dos atributos não-estáticos daquela classe;
- Seguem o mesmo processo usado para atributos não-estáticos:
  1. São zerados (inicializados com seus valores *default*: 0, `false` ou `null`);
  2. Recebem os seus valores iniciais (se especificados), na ordem em que foram definidos na classe.

Blocos de inicialização estática

- No exemplo da classe `Aleatorio`, inicializamos uma variável no construtor porque não conseguíamos fazê-lo em uma só linha;
- E se esta variável for `static`?

```
class Aleatorio {
  int numero;
  Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}
```

Blocos de inicialização estática

- Resolvemos a questão com blocos de inicialização estática;
- Os blocos estáticos de uma classe são executados quando a classe é usada pela 1ª vez.

```
class Aleatorio {
  static int numero;

  static {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}
```

Blocos de inicialização não estática

- Também podemos fazer blocos de inicialização não-estática;
- Funcionam como os construtores: chamados em cada criação de objeto.

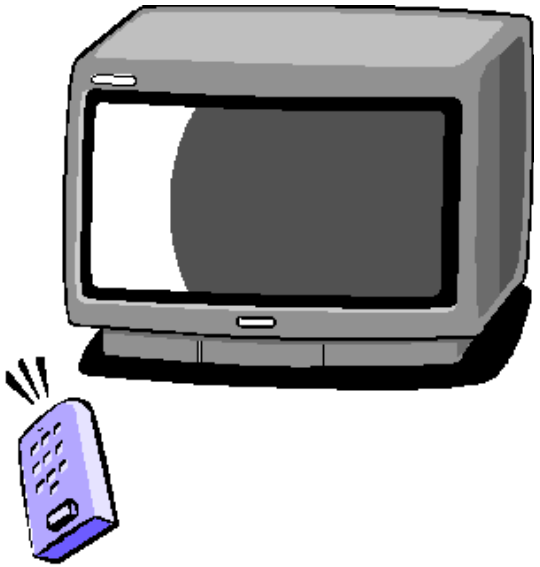
```
class Aleatorio {
  int numero;

  {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}
```

Recapitulando II...

- O que já aprendemos:
  - Construtores são chamados ao criarmos objetos;
  - Podemos sobrecarregar construtores e métodos;
  - Quando chamamos métodos não-estáticos, existe uma referência implícita ao "objeto atual": `this`;
  - A ordem de construção de objetos é importante;
  - Podemos ter blocos de inicialização estática e não-estática em nossas classes.
- Prosseguindo...
  - Como encapsular e ocultar a implementação para que os clientes só conheçam a interface?

Encapsulamento

- Usamos objetos sem saber seu funcionamento interno;
- Assim também deve ser em nossos sistemas OO:
  - Maior manutenibilidade;
  - Maior reusabilidade.



Pacotes (*packages*)

- À medida que aumenta o número de classes, aumenta a chance de coincidência de nomes;
- Precisamos separar as classes em espaços de nomes;
- Java possui o conceito de pacotes:
  - Espaço de nome para evitar conflitos;
  - Agrupamento de classes semelhantes;
  - Maneira de construir bibliotecas de classes;
  - Estabelecimento de políticas de acesso às classes.

Pacotes da API Java

- As APIs Java (ex.: Java SE) são divididas em pacotes:
  - `java.lang`: classes do núcleo da plataforma;
  - `java.util`: classes utilitárias;
  - `java.io`: classes para I/O (entrada/saída);
  - Dentre muitos outros...
- Pacotes são organizados em níveis hierárquicos:
  - `java`
    - `lang`
    - `util`
    - `...`
  - `javax`
    - `swing`
    - `xml`
    - `...`

Pacotes & bibliotecas

- Coleção de arquivos `.class`;
- Compilados de códigos-fonte `.java`;
  - Geralmente uma classe pública por arquivo fonte.
- Declaração do mesmo pacote:
  - Primeira linha não comentada da classe.

```
package meupacote;

public class MinhaClasse {

}
```

Importação de pacotes

- Para usar classes de outros pacotes, é preciso importá-las;
- Uma IDE ajuda nesta tarefa.

```
package outropacote;

// Importa todas as classes do meupacote.
import meupacote.*;

public class OutraClasse {
  MinhaClasse mc;
}
```

Outras opções de importação

```java
package outropacote;

// Importa uma classe específica.
import meupacote.MinhaClasse;

public class OutraClasse {
  MinhaClasse mc;
}
```

```java
package outropacote;

public class OutraClasse {
  // Uso do nome completo da classe.
  meupacote.MinhaClasse mc;
}
```

O Pacote `java.lang`

- As classes do pacote `java.lang` são importadas automaticamente;
- Não é necessário:
  - `import java.lang.String;`
  - `import java.lang.Math;`
  - `import java.lang.*;`

Importação estática

- A partir do Java 5 é possível importar os membros estáticos de uma classe:
- Antes:

```
/* ... */
r = Math.exp(x) + Math.log(y) -
  Math.sqrt(Math.pow(Math.PI, y));
```

- Depois:

Também pode importar somente um específico.

```
import static java.lang.Math.*;

/* ... */
r = exp(x) + log(y) - sqrt(pow(PI, y));
```

Convenção de nomes

- Para não haver conflito com ninguém, sugere-se usar seu domínio na Internet ao contrário:

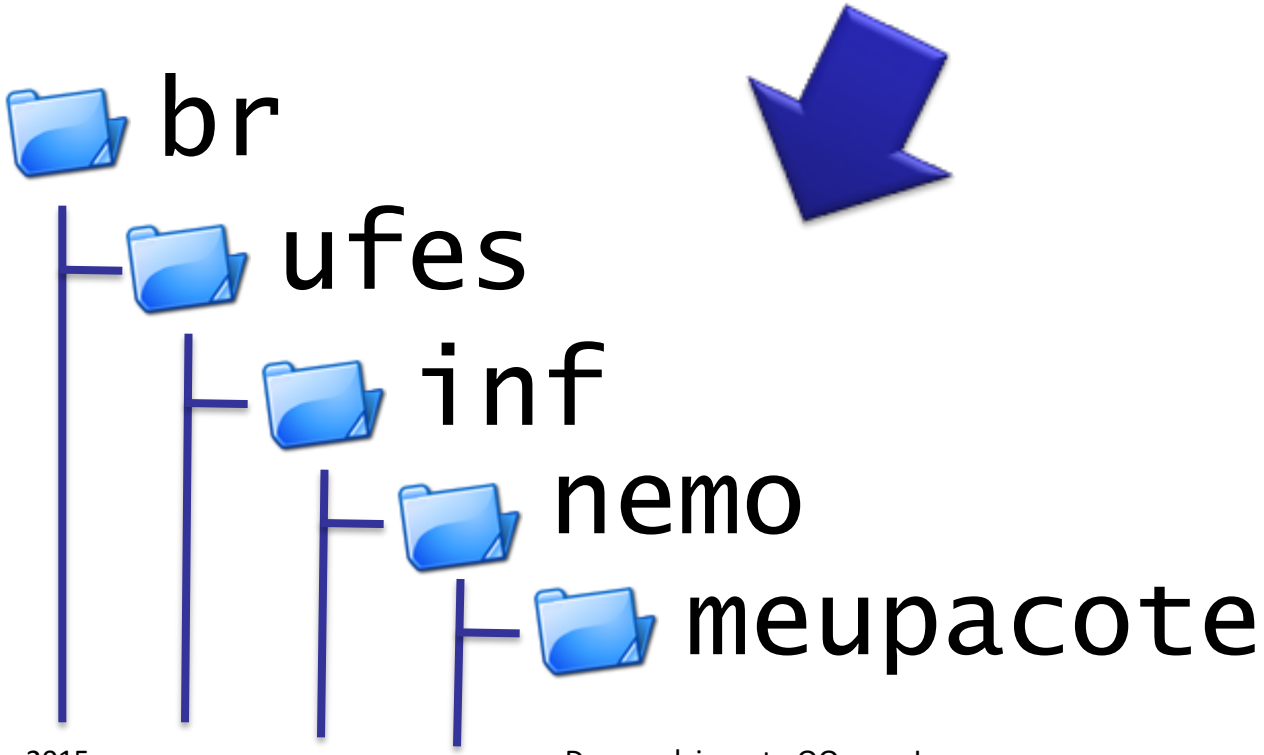
http://nemo.inf.ufes.br

br.ufes.inf.nemo

Localização de pacotes

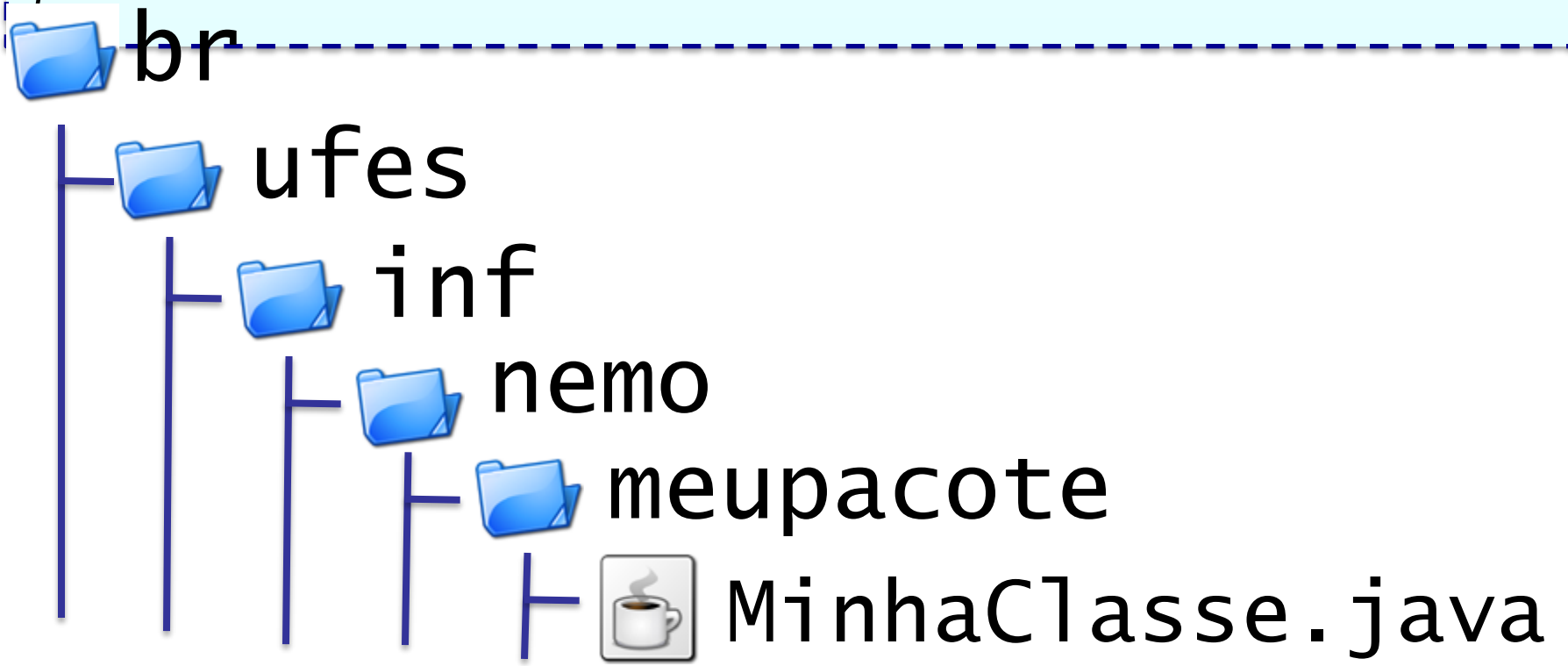
- Como dispor arquivos `.class` em pacotes?
- Maioria das JVMs utiliza pastas no sistema de arquivos do SO:

`br.ufes.inf.nemo.meupacote`



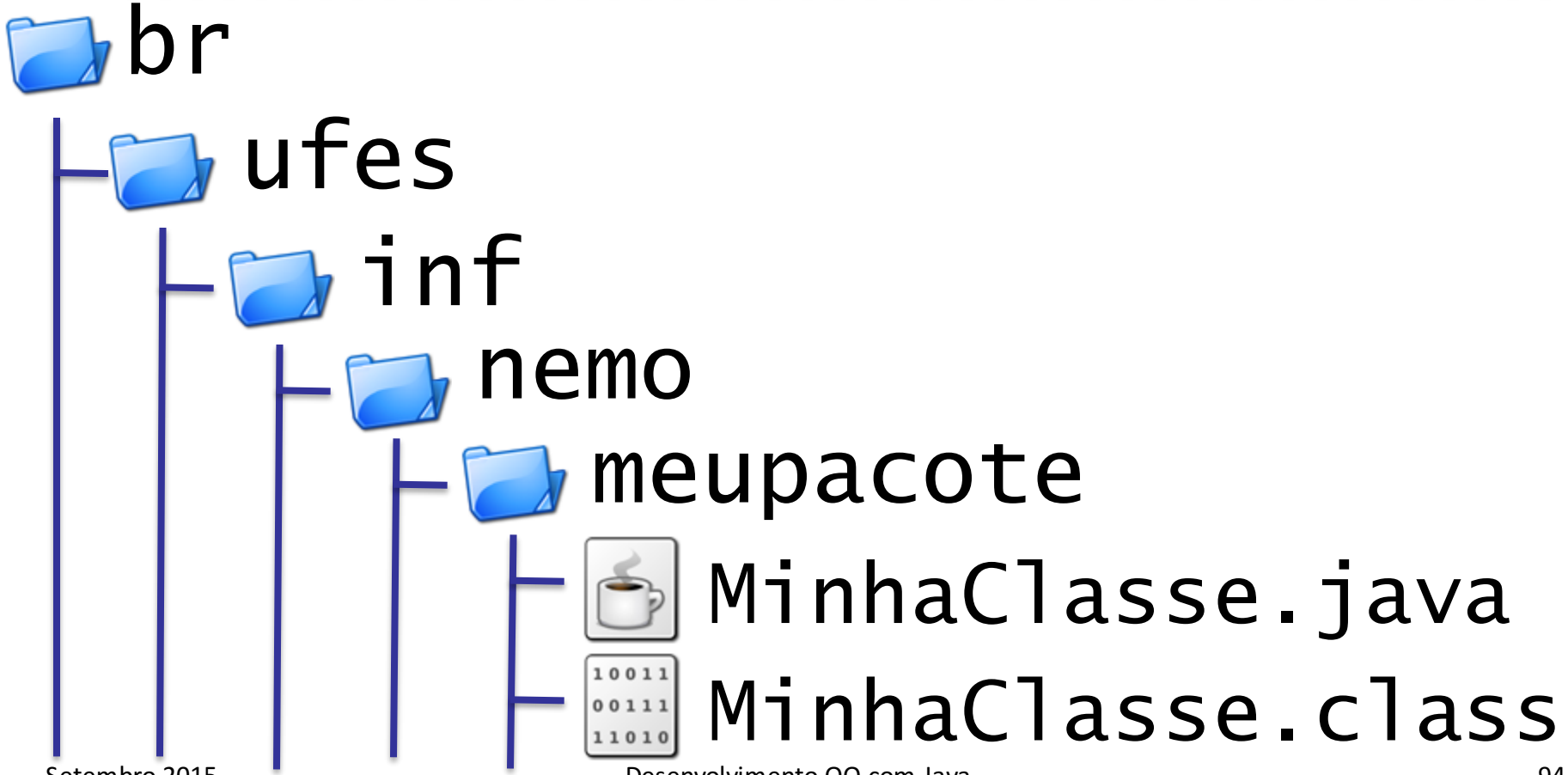
Localização de pacotes

```
package br.ufes.inf.nemo.meupacote;
import java.util.Date;
public class MinhaClasse {
  public static void main(String[] args) {
     System.out.println(new Date());
  }
}
```



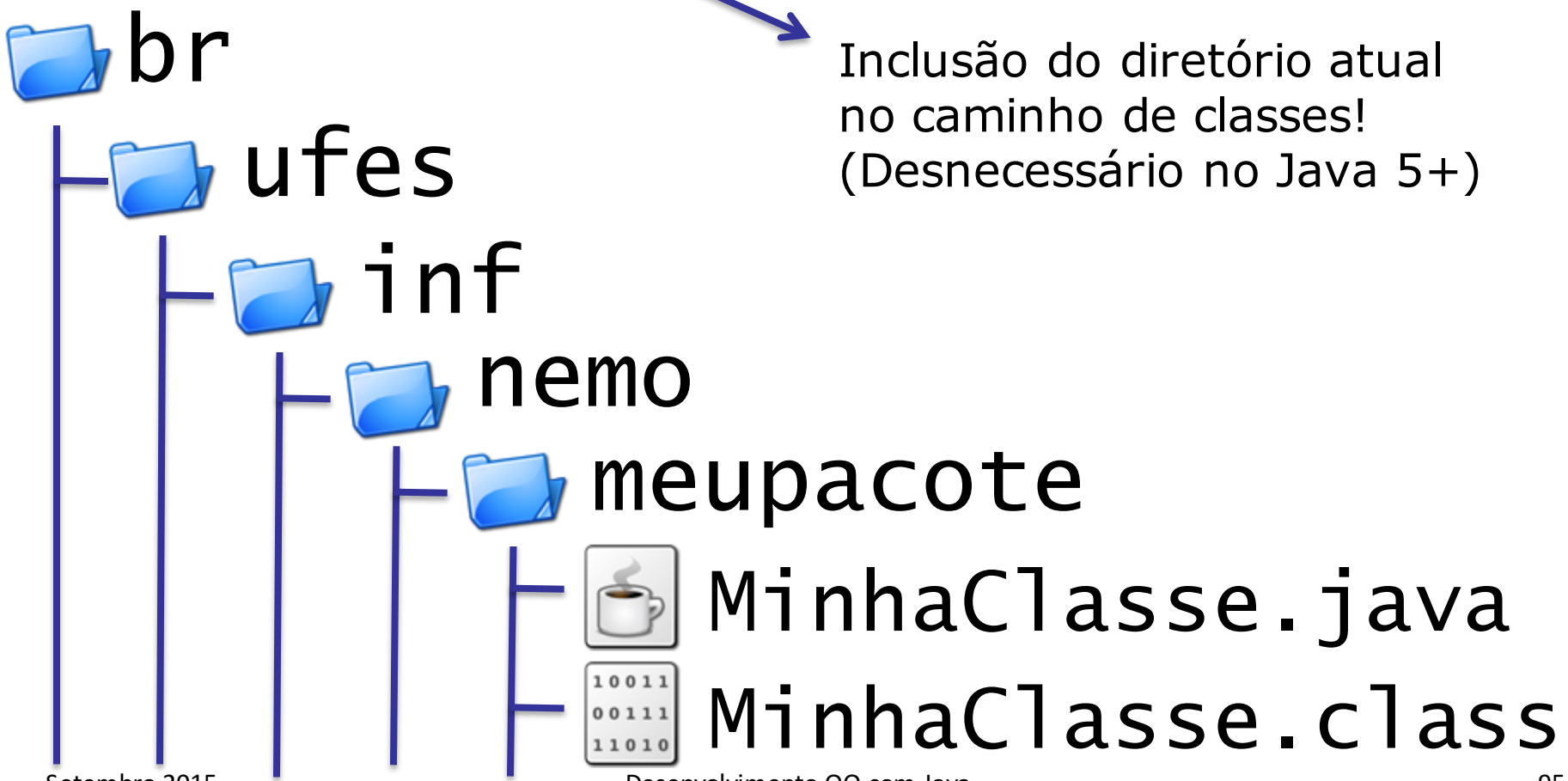
Localização de pacotes

```
$ ls
br

$ javac  br/ufes/inf/nemo/meupacote/MinhaClasse.java
```



Localização de pacotes

```
$ java -cp . br.ufes.inf.nemo.meupacote.MinhaClasse

Wed Jun 05 21:01:29 BRT 2013
```



Inclusão do diretório atual no caminho de classes! (Desnecessário no Java 5+)

Classpath

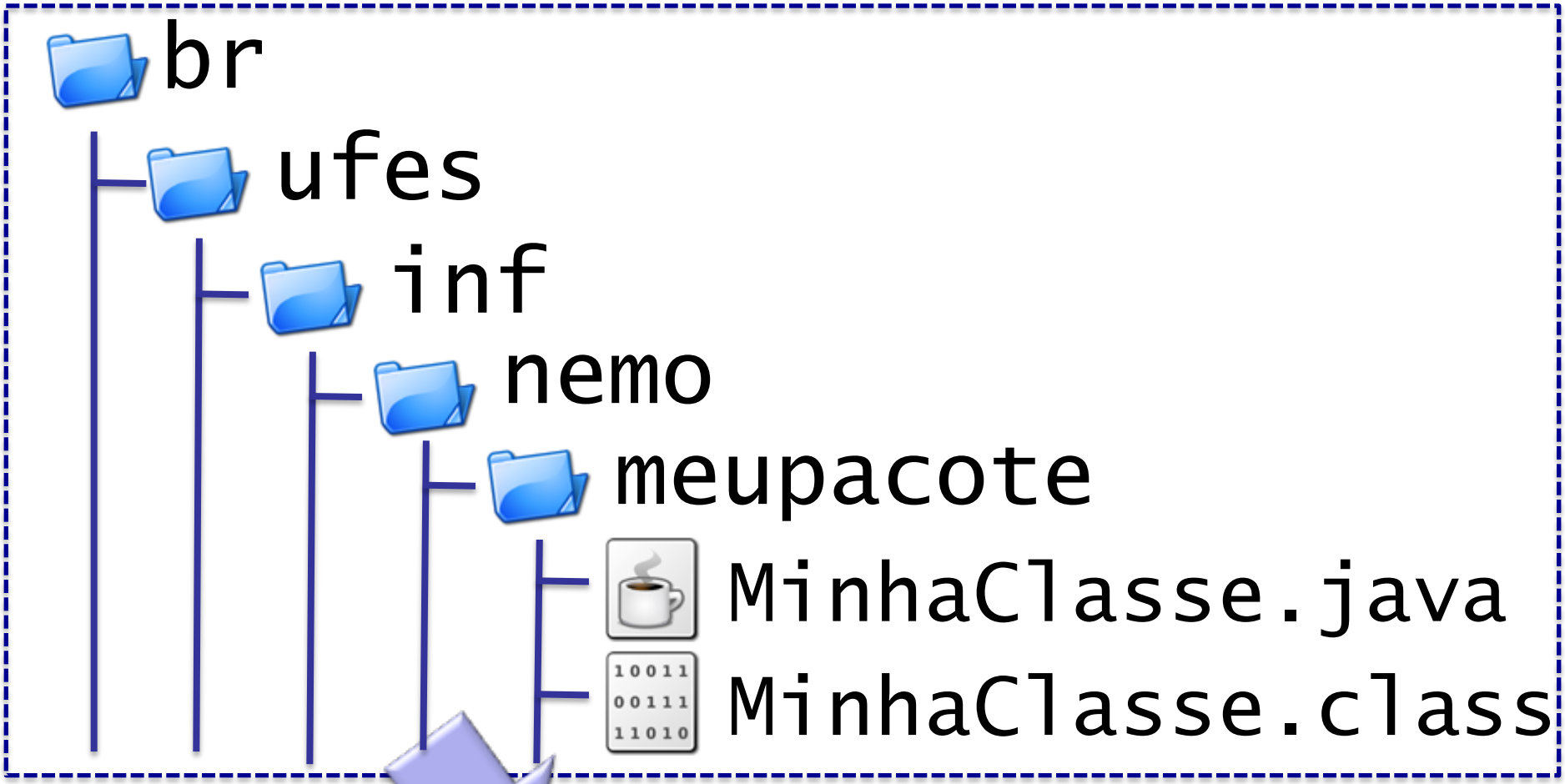
- O "caminho de classes" ou "trilha de classes" é onde as ferramentas do JDK e a JVM procuram classes;
  - A partir dos diretórios do *classpath* procura-se as classes segundo seus pacotes (usa a 1ª encontrada).
- Estão por padrão no *classpath*:
  - A biblioteca de classes da API Java SE;
  - O diretório atual.
- O *classpath* pode ser alterado:
  - Variável de ambiente (não recomendado);
  - Opção `-classpath` ou `-cp`.

Compilação automática

- Ao compilar uma classe, se ela faz referência a outra que não foi compilada, esta última é compilada se o código está disponível;
- Se já foi compilada, mas o arquivo fonte está com data mais recente, ela é recompilada.

- Uso de IDEs:
  - Utilizar uma IDE abstrai todas estas preocupações;
  - A IDE cuida de todo o processo de compilação.

Pacotes JAR




```
jar -c -f meujar.jar br/ufes/inf/nemo/meupacote/*.class
```

meujar.jar

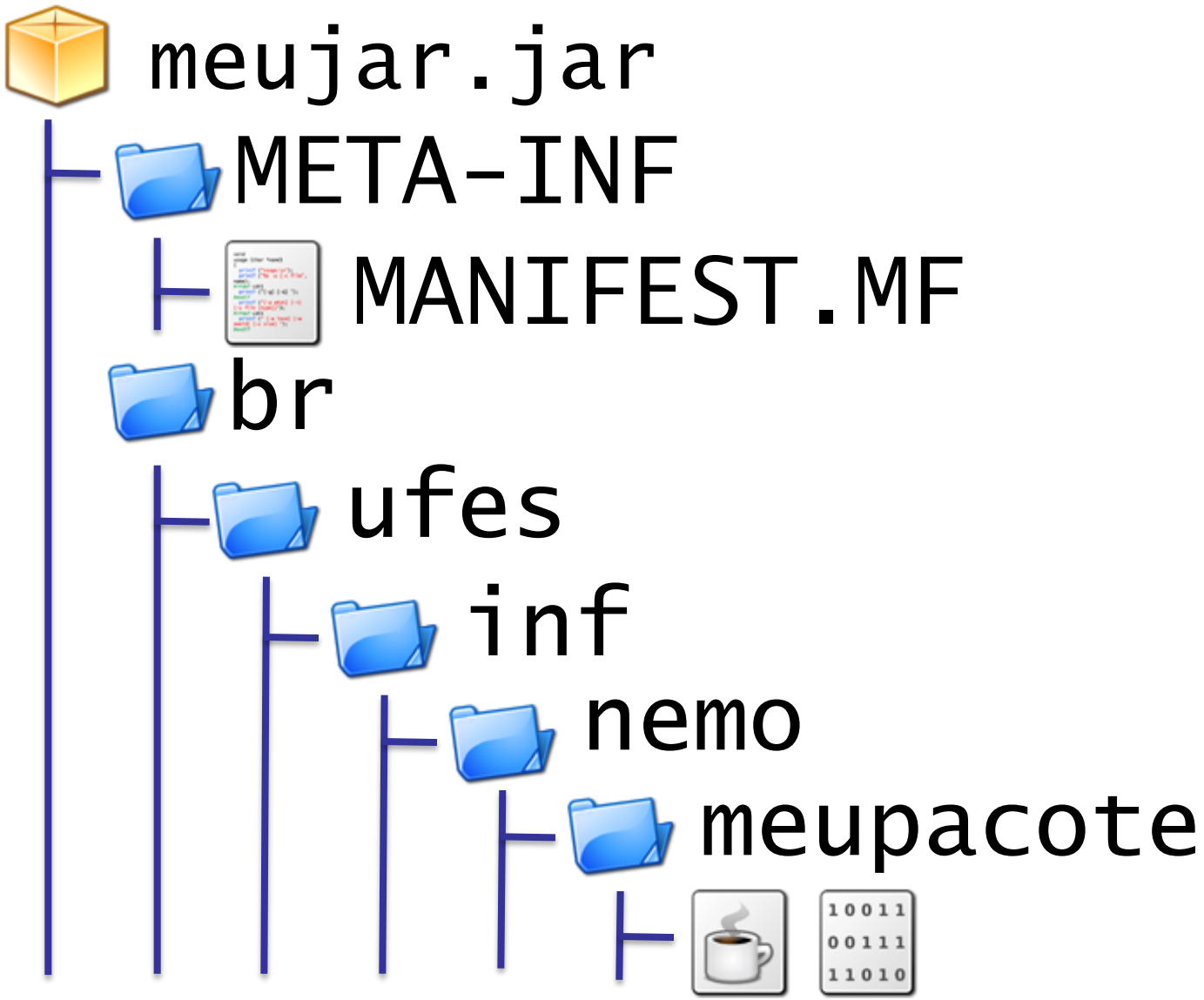
Pacotes JAR

```
$ java -cp meujar.jar
br.ufes.inf.nemo.meupacote.MinhaClasse

Wed Jun 05 21:15:06 BRT 2013
```

meujar.jar

Arquivos JAR são compactados no formato ZIP e podem ser abertos por qualquer programa compatível.

Pacotes JAR


meujar.jar
META-INF
MANIFEST.MF
br
ufes
inf
nemo
meupacote
10011
00111
11010

O arquivo MANIFEST

- Contém meta-dados sobre o pacote:
- Crie um arquivo MANIFEST.MF:

```
Main-Class: br.ufes.inf.nemo.meupacote.MinhaClasse

```

- Digite os seguintes comandos:

```
$ jar -c -f meujar.jar -m MANIFEST.MF
br/ufes/inf/nemo/meupacote/*.class

$ java -jar meujar.jar
Wed Jun 05 21:23:03 BRT 2013
```

O pacote padrão

- Toda classe que não especifica o pacote pertence ao pacote padrão;
- Seu `.class` deve estar numa pasta raiz do *classpath*.

```
public class Bolo {
  public static void main(String[] args) {
     // Não há import, estão no mesmo pacote.
     Torta t = new Torta();
     t.f();
  }
}

class Torta {
  void f() { System.out.println("Torta.f()"); }
}
```

Especificadores de acesso

- Determinam a visibilidade de um determinado membro da classe com relação a outras classes;
- Há quatro níveis de acesso:
  - Público (*public*);
  - Privado (*private*);
  - Protegido (*protected*);
  - Amigo ou privado ao pacote (*friendly* ou *package-private)*.

Palavras-chave

- Três palavras-chave especificam o acesso:
  - `public`
  - `private`
  - `protected`
- O nível de acesso *package-private* é determinado pela ausência de especificador;
- Devem ser usadas antes do nome do membro que querem especificar;
- Não podem ser usadas em conjunto.

Membros públicos

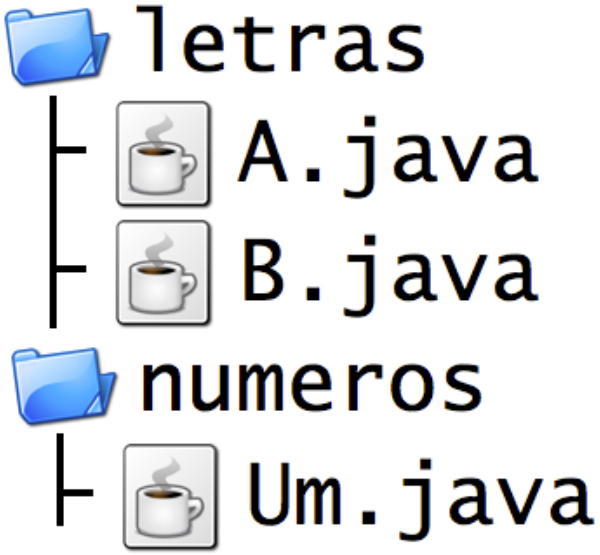
Membro	Resultado
Classes	Classes públicas* podem ser importadas por qualquer classe.
Atributos	Atributos públicos podem ser lidos e alterados por qualquer classe.
Métodos	Métodos públicos podem ser chamados por qualquer classe.

* Só pode haver uma classe pública por arquivo-fonte e os nomes (da classe e do arquivo) devem ser iguais.

Membros públicos

```
public class A {
  public int x = 10;
  public void print() {
     System.out.println(x);
  }
}
```

```
import letras.B;
public class Um {
  B b = new B();
  public void g() {
     b.f();
  }
}
```




```
public class B {
  public A a = new A();
  public void f() {
     a.x = 15;
     a.print();
  }
}
```

Finalmente, PSVM!

- O método `main()` é:
  - `public`, pois deve ser chamado pela JVM;
  - `static`, pois pertence à classe como um todo (a JVM não instancia um objeto para chamá-lo);
  - `void`, pois não retorna nada.
- A classe que possui o método `main()` deve ser:
  - `public`, pois deve ser acessível pela JVM.

Membros privados

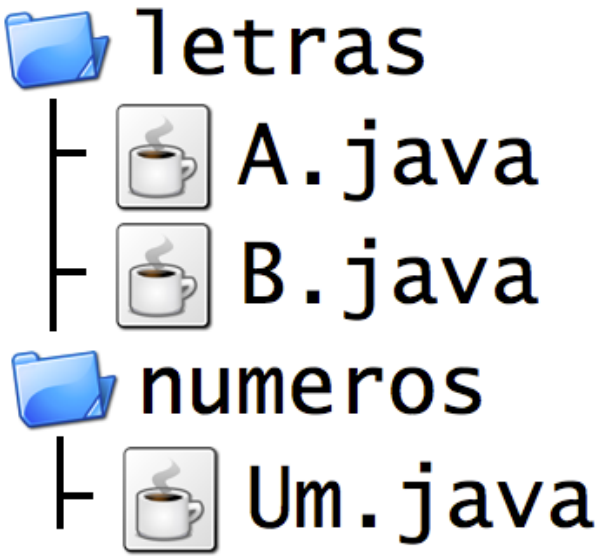
Membro	Resultado
Classes	Somente classes internas* podem ser declaradas privadas.
Atributos	Atributos privados só podem ser lidos e alterados pela própria classe.
Métodos	Métodos privados só podem ser chamados pela própria classe.

* Tópico avançado, veremos posteriormente.

Membros privados

```
public class A {
  private int x = 10;
  private void print() {
     System.out.println(x);
  }
  void incr() { x++; }
}
```

```
import letras.B;
public class Um {
  B b = new B();
  public void g() {
     b.f();
  }
}
```




```
public class B {
  public A a = new A();
  public void f() {
     // Erro: a.x = 15;
     // Erro: a.print();
  }
}
```

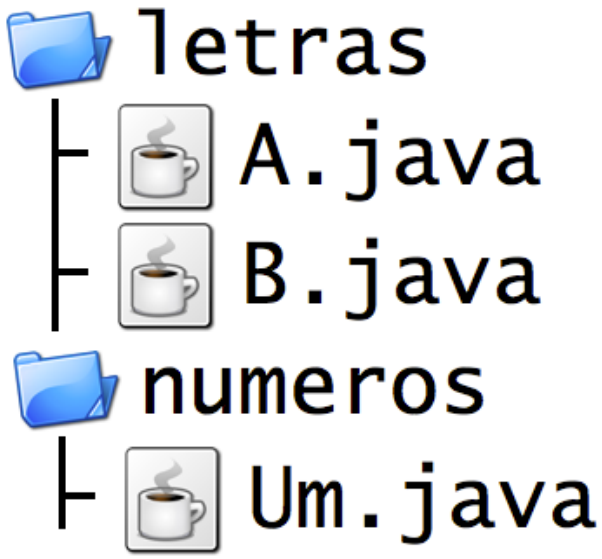
Membros amigos (*package-private*)

Membro	Resultado
Classes	Classes amigas só podem ser utilizadas por classes do mesmo pacote.
Atributos	Atributos amigos só podem ser lidos e alterados por classes do mesmo pacote.
Métodos	Métodos amigos só podem ser chamados por classes do mesmo pacote.

Membros amigos

```
class A {
  int x = 10;
  void print() {
     System.out.println(x);
  }
  void incr() { x++; }
}
```

```
import letras.*;
public class Um {
  // Erro: A a;
  B b = new B();
  public void g() {
     // b.a.incr();
     b.f();
  }
}
```




```
public class B {
  A a = new A();
  public void f() {
     a.x = 15; a.print();
  }
}
```

Membros protegidos

Membro	Resultado
Classes	Somente classes internas* podem ser declaradas protegidas.
Atributos	Atributos protegidos só podem ser lidos e alterados por classes do mesmo pacote ou subclasses*.
Métodos	Métodos protegidos só podem ser chamados por classes do mesmo pacote ou subclasses*.

* Tópico avançado, veremos posteriormente.

Membros protegidos

Acesso	Público	Protegido	Amigo	Privado
Mesma classe	Sim	Sim	Sim	Sim
Classe no mesmo pacote	Sim	Sim	Sim	Não
Subclasse em pacote diferente	Sim	Sim	Não	Não
Não-subclasse em pacote diferente	Sim	Não	Não	Não

Separando interface e implementação

- Em OO é fundamental o ocultamento de informação:
  - Estrutura interna fica inacessível;
  - Interface do objeto é pública.

- O que é uma pilha?
  - Uma lista?
  - Um vetor?
  - Uma estrutura que me permite empilhar e desempilhar itens?

Ocultamento de informações

```
import java.util.*;

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

Mudando a implementação

```
import java.util.*;

public class Pilha {
  private LinkedList elems;
  public Pilha() {
     elems = new LinkedList();
  }
  public void empilha(Object obj) {
     elems.addFirst(obj);
  }
  public Object desempilha() {
     return elems.removeFirst();
  }
}
```

Regra geral dos POJOs

- Atributos devem ser privados;
- Se precisarem ser lidos ou alterados, prover métodos get/set:

```
public class Pessoa {
  private String nome;
  public String getNome() {
     return nome;
  }
  public void setNome(String nome) {
     this.nome = nome;
  }
}
```

http://nemo.inf.ufes.br/