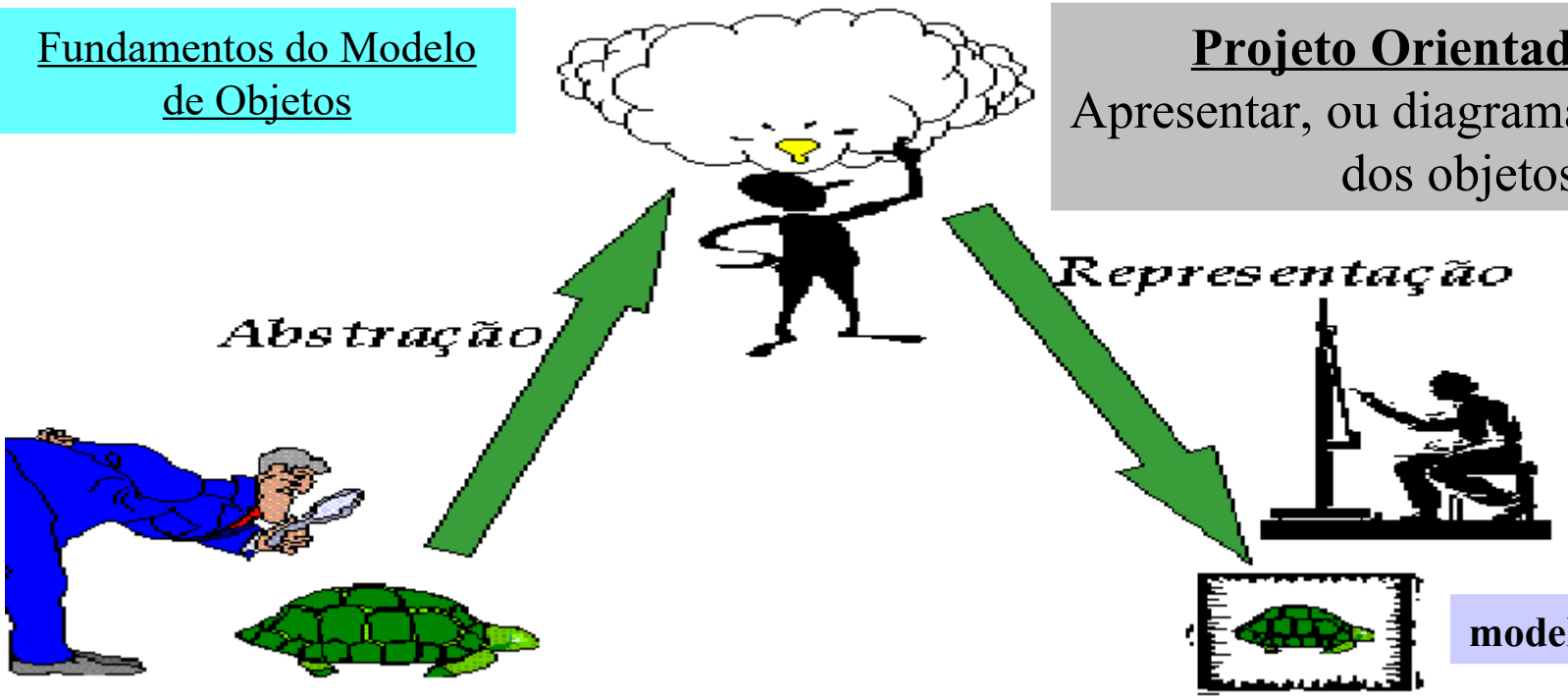
Java e a Orientação a Objetos

Fundamentos do Modelo de Objetos



Projeto Orientado a Objetos
Apresentar, ou diagramar, o modelo dos objetos observados

modelo conceitual

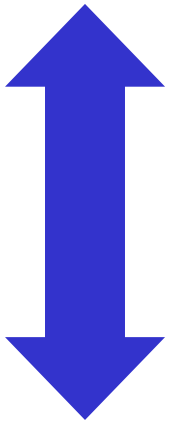
Análise Orientada a Objetos
Decompor a estrutura do problema em entidades abstratas

Programação Orientada a Objetos
"implementar" as estruturas modelas em código de computador

```
public class Tartaruga {
   // Atributos, ou propriedades
   // Métodos, ou operações
}

Tartaruga objTartura = new Tartaruga();
```

POO
Programação Orientada a Objetos



OOP
Object-Oriented Programing

Analisando o "Objeto Rádio"

Como utilizamos esse aparelho ? Por meio dos controles existentes em sua parte externa, como o botão de liga/desliga, o controle de volume, o seletor de estações e, conforme o modelo, outros controles. Isso significa que o rádio pode ser controlado pelas operações de desligar, ligar, sintonizar e aumentar ou diminuir seu volume. Mesmo sem saber como funciona internamente esse aparelho ou quais são os princípios físicos envolvidos na radiodifusão, as pessoas são capazes de usá-lo corretamente. Os detalhes de sua estrutura ficam ocultos dentro de sua "carcaça", pois temos acesso apenas aos controles que o projetista julgou necessários para seu uso doméstico. Outro aspecto é o estado do rádio, que pode estar desligado ou ligado. Estando ligado, pode ser sintonizado em diferentes freqüências e ter o volume ajustado de forma distinta. O rádio portátil, o rádio-relógio e muitos outros, também são semelhantes ao rádio que acabamos de descrever.

Por meio desse exemplo cotidiano, descrevemos informalmente algumas das características da Programação Orientada a Objetos (POO):
Classificação, Identidade, Encapsulamento e Herança.

Caracteristicas da POO (1/2)

- Classificação

É natural para o nosso entendimento criarmos uma classificação para as coisas, ou seja, a categorização daquilo que nos rodeia. Criarmos classes de objetos para facilitar nossa compreensão do mundo, ou seja, grupos de objetos que compartilham características comuns, embora distintas, em cada objeto (os seres humamos têm características comuns: estrutura e funcionamento do corpo, embora possam ser distinguidos por sua altura, idade, seu peso, cor e aparência da pele, olhos, cabelos etc.).

- Identidade

Nosso relacionamento com os objetos do mundo não se limita à sua classificação. Usualmente, nós denominamos todos os objetos e, principalmente, as pessoas que estão ao nosso redor, de forma que cada objeto se torne único, ou seja, assuma uma identidade. As pessoas têm nomes e sobrenomes, nossos automóveis têm uma chapa e até mesmo o patrimônio de uma empresa é numerado, de modo que cada peça seja única.

Caracteristicas da POO (2/2)

- Encapsulamento

O encapsulamento indica que podemos utilizar um objeto conhecendo apenas sua “interface”, isto é, sua aparência exterior, tal como fazemos, muitas vezes, com computadores, automóveis e outras máquinas de nosso tempo. O encapsulamento permite preservar informações consideradas importantes ou sigilosas dentro do objeto, divulgando apenas aquilo que se julgue conviniente.

- Herança

A herança é o mecanismo de criação de novas classes de objetos a partir de outras já existentes. A herança toma como prontas e disponíveis certas características do objeto de origem, sendo portanto uma forma muito especial de “compartilhamento de código”. Isso permite que criemos várias classes, hierarquicamente relacionadas, partindo de uma mais geral para diversas outras mais especializadas, obtendo como resultado uma árvore de classes, tal como a classificação proposta pela biologia para a divisão dos seres vivos.

Rumbaugh define Orientação a Objetos (OO), como:

"Uma nova maneira de pensar os problemas utilizando modelos organizados a partir de conceitos do mundo real. O comportamento fundamental é o objeto que combina estrutura e comportamento em uma única entidade".

Introdução

- A idéia fundamental da programação orientada a objetos é que toda estrutura de dados utilizada em um programa, ou aplicação, deve estar intimamente associada às operações que realizam o acesso a ela, e vice-versa.
- O paradigma da OO é baseado na construção de sistemas a partir de componentes reutilizáveis, chamados de classes. Uma classe generaliza/ representa um conjunto de objetos similares.

Paradigma = Padrão

Pode ser descrito como uma estratégia ou abordagem genérica para se fazer alguma coisa. Muitas pessoas consideram o paradigma como um modelo ou um conjunto de normas específicas.

Paradigma da OO

Estratégia de desenvolvimento baseada no conceito de que o sistema deve ser construído a partir de componentes reutilizáveis, chamados de classes.

As bases da OO

- A tecnologia de objetos apresenta componentes chaves que fundamentam a *mudança de enfoque no processo de modelagem e desenvolvimento de aplicações*, trazendo benefícios intrínsecos à filosofia [FURLAN, 1998].
  - programação estrutura: estruturas básicas de controle (seqüencial, condicional e repetitiva), modularização, tipos abstratos de dados
  - programação orientada a objetos: classes, objeto, herança (reutilização de código), polimorfismo

Mudança de Enfoque (1/2)

- Enfoque tradicional de modelagem
  - programação estruturada
  - construção de sistemas de informação baseado na compreensão desse sistema como um conjunto de subprogramas que, por sua vez, executam processos sobre os dados;
  - programa = estrutura de dados + algoritmo.

Programação Estruturada

Sob Perspectiva de um Subprograma.

- Nessa visão, o principal bloco de construção do software é o procedimento ou a função.
- Quebrar a tarefa principal em subtarefas e estas em subtarefas menores (dividir-para-conquistar), até que seja possível implementá-las.
- Implementar procedimentos para resolver tarefas simples e depois agrupá-los em procedimentos mais complexos, até se chegar à resolução da tarefa principal.

Mudança de Enfoque (2/2)

- Enfoque de modelagem por objetos
  - programação orientada a objetos
  - observar o mundo como uma coletânea de objetos que interagem entre si, apresentam características próprias que são representadas pelos seus atributos (dados) e operações (processos, ou métodos);
  - classe = atributos + métodos (ou operações)

Programação Orientada a Objetos

Sob Perspectiva de uma Classe.

- Nesta visão, o principal bloco de construção de todos os sistemas de software é o objeto ou a classe.
- Identificação dos objetos envolvidos com o sistema a ser desenvolvido e sua representação em forma de classe.
- Identificação de suas características relevantes e sua representação em forma de atributos.
- Identificação de ações realizadas e serviços por esses objetos e sua representação em forma de métodos.

Programação Estruturada

```
void Operacao(TipoRegistro *Elemento);

                    x = random(100);
                    Empilha(x, &PilhaUm);
```

Programação Orientada a Objetos

```
Elemento.Operacao;

                    double x = Math.random();
                    pilhaUm.empilha(x);
```

Em Síntese:
POO, Programação Orientada a Objetos

=

Estrutura de Dados (ou Atributos)

+

Algoritmos (ou Métodos)

Histórico

- Área acadêmica
- 1967, Simula
- 1980, Smalltalk, a primeira linguagem a ter uma implementação completa do conceito de POO.
- Linguagens híbridas = linguagens POO montadas sobre as linguagens existentes
  - C C++
  - Pascal Object Pascal (Delphi).

Vocabulário da POO, Parte I

- Classe
- Atributos, ou propriedades
- Métodos, ou operações
- Objeto, ou instância
  - construtores
  - destrutores (coleta de lixo)
- package (pacote)

Classe (1/3)

A definição de um “modelo conceitual” para o domínio da aplicação, contemplando as classes relevantes e suas associações, é um dos principais resultados das etapas de análise e projeto orientado a objetos. A adoção de uma linguagem de modelagem, tal como o diagrama de classes UML, permite expressar esse resultado de maneira organizada e padronizada.

Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que “propriedades, ou atributos” o objeto terá.

Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de “métodos”. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para um objeto desse tipo.

Classe (2/3)

- Classe é um tipo de dados definido pelo usuário que tem alguns dados internos e alguns métodos, na forma de procedimentos ou funções, que atuam sobre estes dados.
- A classe é o modelo a partir do qual são criados os objetos.
- Uma classe é um agrupamento de objetos que revelam profundas semelhanças entre si, tanto no aspecto estrutural quanto funcional.
- Classe é uma abstração que descreve as propriedades relevantes de uma "aplicação" em termos de sua estrutura (dados) e de seu comportamento (operações) ⇔ Programa.

Operações de Abstração

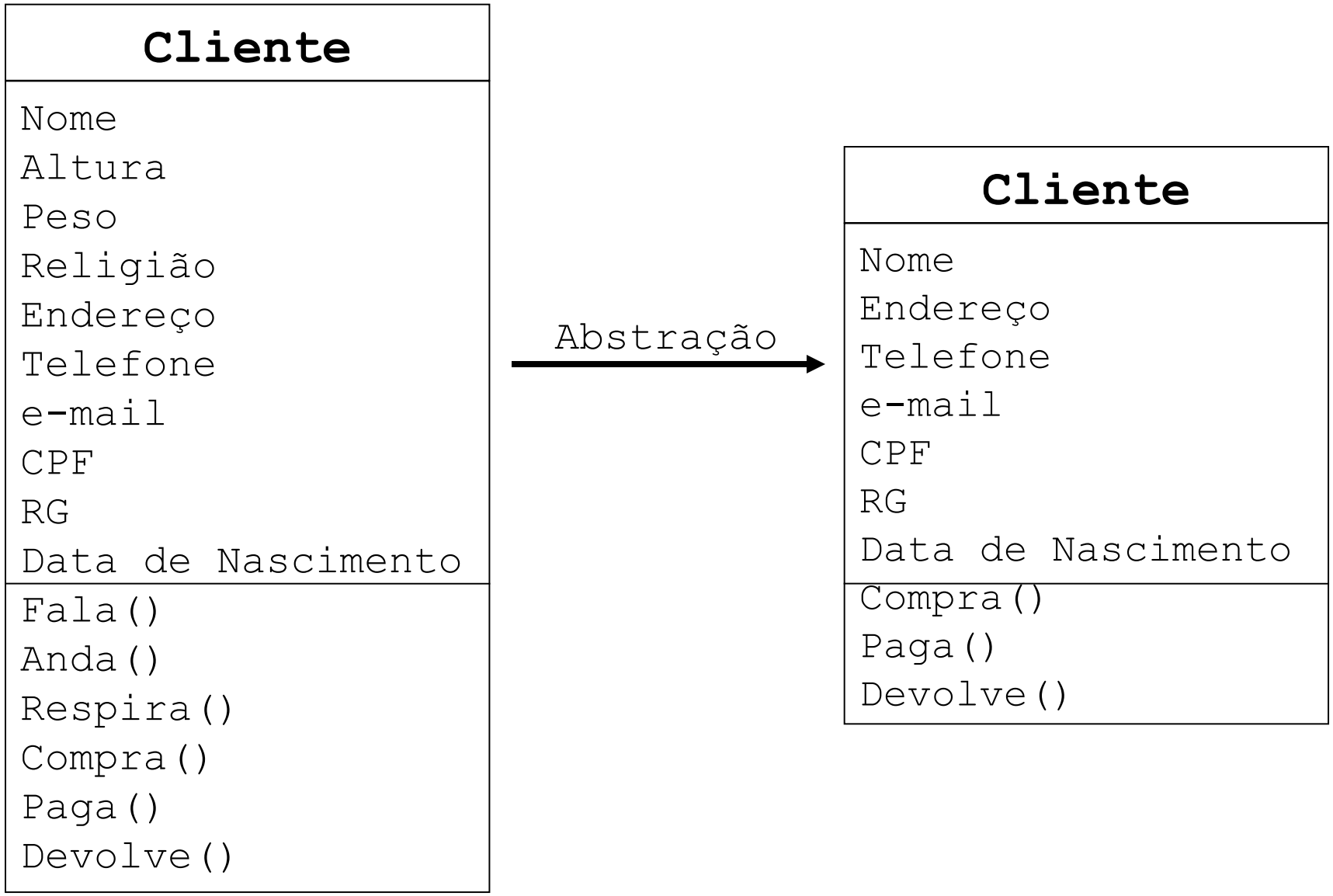
Em um sistema cuja tarefa principal seja administrar uma base de dados cadastrais de clientes, é evidente que os principais objetos envolvidos são os próprios clientes. O processo de abstração começa com a criação de uma classe para representar esse conjunto de objetos dentro do sistema.

As classes não devem incluir todos os atributos dos objetos e tampouco devem incluir todas as sua ações e serviços. Somente devem incluir os atributos e as ações (ou serviços) pertinentes ao papel a ser desempenhado dentro do programa.

O processo de abstração compõe-se de um conjunto de operações realizadas com a finalidade de representar objetos do mundo real em forma de classes, de modo que eles possam interagir em um programa para realizar determinadas tarefas.

Um primeiro esforço deve ser empreendido no sentido de identificar os atributos e métodos relevantes de cada classe e para eliminar atributos e métodos desnecessários.

Processo de Abstração



Classe (3/3)

- O que deve ser ressaltado é que a “Classe” propõe o que é denominado de “modelo conceitual” para os objetos analisados.
- É a representação da idéia (processo de abstração):
  - de como os objetos são (característica = atributos = dados)
  - e o que os objetos fazem (comportamento = operações = métodos = funções = algoritmos).

Processo de Abstração

Criar classes que representam os aspectos relevantes de objetos é apenas um passo no contexto do processo de abstração. Há, na verdade, seis operações de abstração distintas a serem analizadas:

1. Classificação
2. Instanciação
3. Generalização
4. Especialização
5. Agregação
6. Decomposição

Classificação

A classificação é a operação de abstração através da qual são analisados os objetos do mundo real para identificar suas características comuns e definir a que classe cada um deles pertence. Na prática, a operação de classificação refere-se à criação de uma classe para representar vários objetos que possuem os mesmos atributos e realizam as mesmas ações.

A operação de classificação inicia-se com a percepção de que alguns objetos possuem as mesmas características e manifestam o mesmo comportamento. Pode-se, por exemplo, perceber que existem algumas características comuns entre os veículos: todos possuem uma marca, um modelo, um ano, uma cor, uma placa, um chassi, etc. Além disso, todos eles realizam algumas ações comuns: ligar-se, desligar-se, mover-se, acender e apagar faróis, ligar e desligar setas de direção, trocar marchas, etc. Sendo assim, todos eles podem ser classificados dentro de uma mesma categoria. O passo seguinte da operação de classificação seria a criação de uma classe para representar as características e o comportamento comuns identificados nesses objetos (os veículos).

Instanciação

A operação de instanciação segue o caminho inverso: ela é caracterizada pela criação de objetos a partir de uma classe já existente.

Depois que uma classe é criada para representar um grupo de objetos, você poderá utilizá-la para criar objetos dentro de um programa. É possível criar a quantidade de objetos que for necessária a partir de uma mesma classe. Instanciar uma classe significa criar um objeto com os seus atributos e métodos.

Objetos são instâncias de classes e são entidades concretas. Quando um programa é executado, os objetos ocupam memória para armazenar dados em seus atributos e para representar seus métodos. A relação existente entre classes e objetos é a mesma que existe entre os tipos de dados e as variáveis.

Instância é do tipo definido pela Classe
João da Silva *é um* Cliente
Mari dos Santos *é uma* Cliente

Classificação e Instanciação

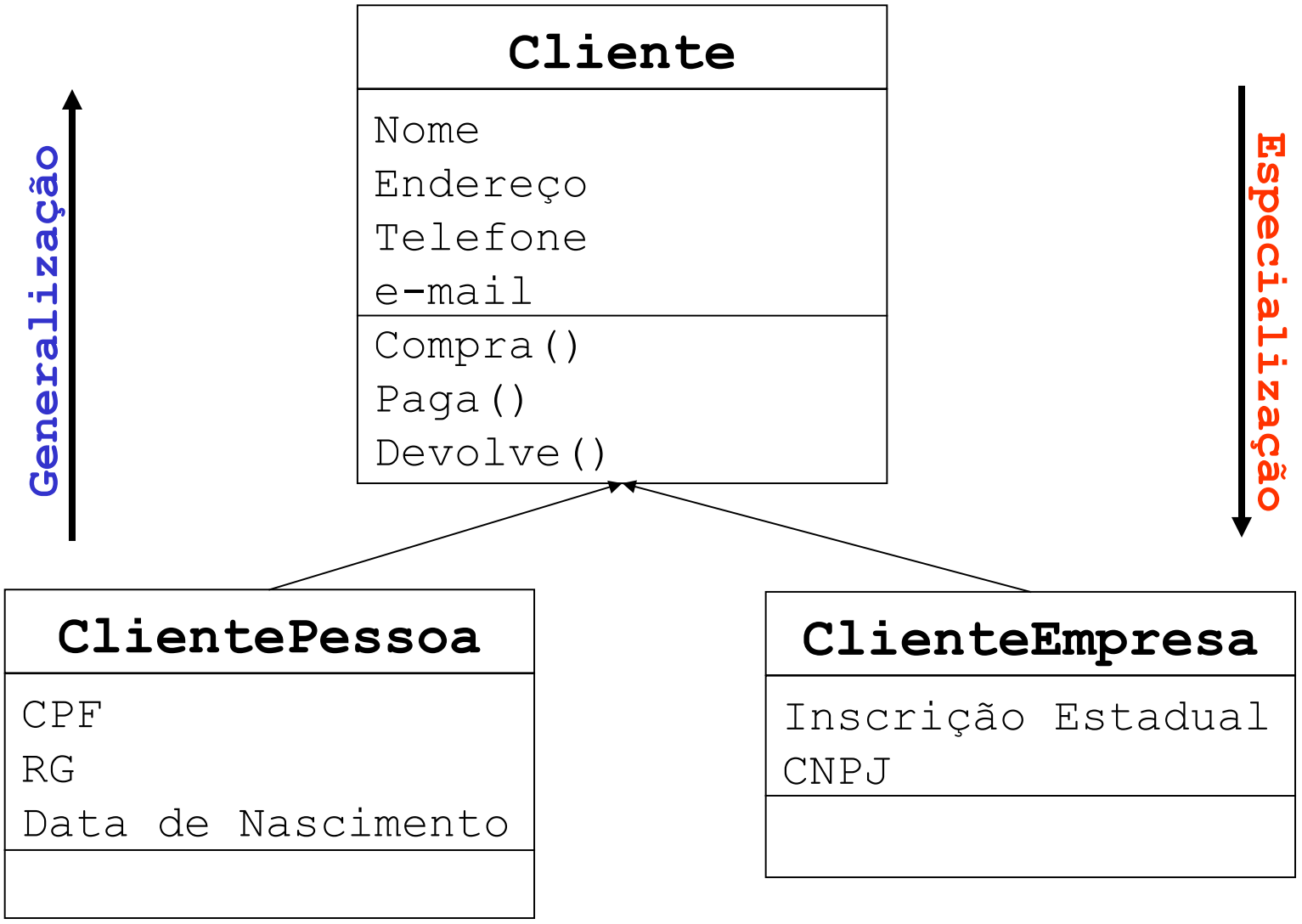
Generalização *versus* Especialização

A generalização é a operação e abstração através da qual se procede a análise de um conjunto de classes e a identificação de características comuns entre elas. O objetivo dessa operação é a criação de uma classe mais "genérica" para representar os atributos e métodos existentes em duas ou mais classes específicas.

A especialização segue no caminho inverso da generalização. Ela se caracteriza pela criação de duas ou mais classes "específicas" a partir de um classe genérica para representar atributos e métodos que são "distintos" entre elas.

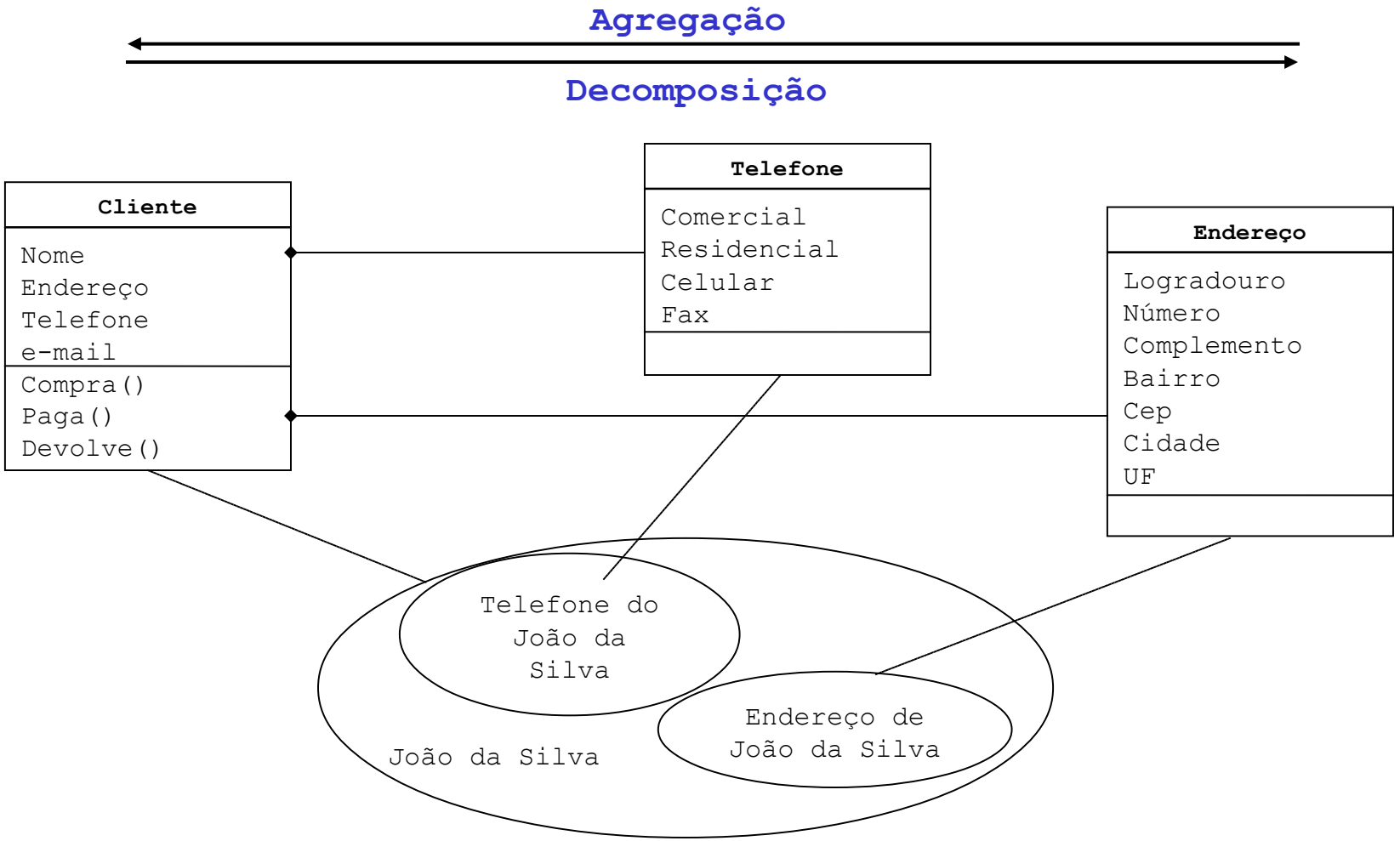
Especialização representa um tipo da Classe Genérica
ClientePessoa representa *um tipo* de Cliente
ClienteEmpresa representa *um tipo* de Cliente

Generalização e Especialização



Agregação e Decomposição

Uma operação de agregação é realizada quando um objeto é utilizado como parte da composição de outro objeto. Uma operação de decomposição, por outro lado, diz respeito à atividade voltada para a separação dos objetos que compõem outro objeto, isolando-os.



1. Definição de Classes em Java

Como foi demonstrado, uma classe tem dois grupos de elementos:
- seus Atributos, também denominados de variáveis-membro ou campos; e
- seus Métodos, operações ou funções-membro.

A sintaxe Java para a definição de uma "classe elementar" é bastante simples, constituindo-se de uma declaração seguida de um corpo delimitado por chaves, no qual residem as declarações de todos os atributos e métodos da classe, como mostra a seguinte estrutura de código:

```
[Modificador] class NomeDaClasse {
// Corpo da Classe
// ---------------------------------------
// Atributos, ou variáveis da Classe
:
// Métodos, ou funções da Classe
:
}
```

PrimeiraClasse.java

O Modificador especifica a acessibilidade da classe (opcional), se presente, pode ser uma combinação de public e abstract ou final.

No momento, é necessário entender que uma classe declarada com um especificador de acesso public, indica que todo o conteúdo (atributos e métodos) público da classe pode ser utilizado livremente (sem restrições) pelas classes das aplicações, por outras classes do mesmo pacote (package) ou de outro pacote. Veja, a seguir, a construção de uma nova classe pública, denominada **PrimeiraClasse**:

```
public class PrimeiraClasse {
// Corpo da Classe
// ---------------------------------------
// Declaração de Atributos
:
// Declaração de Métodos
:
}
```

Salvando o Código da Classe

Ao salvar o código correspondente a uma classe em um arquivo, deve-se ter os seguintes cuidados:

1. Em um arquivo Java, pode existir várias definições diferentes de classes, mas apenas uma delas pode ser pública. Se desejarmos várias classes públicas, então, cada uma delas deverá ser separadamente salva em arquivos distintos.
2. O "nome do arquivo" deve ser sempre o "nome da classe" pública que ele contém, observando-se cuidadosamente o mesmo uso de letras maiúsculas e minúsculas, tanto para o nome da classe como para o nome do arquivo (Java é uma linguagem sensível ao uso de letras maiúsculas e minúsculas em tudo).

Portanto, o nome de um arquivo, para conter a classe pública PrimeiraClasse, deve ser PrimeiraClasse.java. O que inicialmente pode parecer uma restrição é, na verdade, um mecanismo inteligente e simples para manter nomes realmente representativos para os arquivos de um projeto.

Regras para Nomes de Classes

Lembrando: O nome de uma variável ou classe em Java pode ser formado por uma seqüência de um ou mais caracteres alfabéticos e numéricos, iniciados por uma letra ou ainda pelos caracteres de sublinhar '_' ou cifrão '**$**'.

Em Java, recomenda-se que a declaração de classes utilize nomes iniciados com letras maiúsculas, "diferenciando-se" dos nomes de variáveis ou instâncias de objetos que devem iniciar com uma letra minúscula. Caso o nome seja composto de mais de uma palavra, todas as iniciais de cada palavra também deverão ser iniciadas com letras maiúsculas, tal como nos exemplos:

Bola	Socket	Filter
BolaDeFutebol	ServerSocket	DataChangeObserver

A utilização de caracteres numéricos no nome de classes também é livre, ao passo que o uso do traço de sublinhar (*underscore* '_') não é indicado.

Tal como no caso de variáveis, e recomendável uma escolha criteriosa para os nomes das classes, de forma que estes sejam verdadeiramente significativos, expressando de maneira simples a idéia central da classe ou dos objetos que a mesma representa.

Atributos, ou Propriedades (1/3)

As classes podem ter zero, um ou mais atributos, que são variáveis destinadas a armazenar informações intrinsecamente associadas aos objetos representados. Por exemplo, ao falarmos de um "Objeto Bola" qualquer, seu tamanho é uma variável associada, pois toda bola, sendo um objeto concreto, deve ter um tamanho. O tamanho pode ser definido mais rigorasamente como uma medida de raio pois essa medida está associada naturalmente à sua forma geométrica. Assim sendo, é bastante natural definirmos uma classe **Bola** que possua como um atributo uma variável destinada a armazenar seu tamanho ou raio, como abaixo:

```
// Bola.java
public class Bola {
// Corpo da Classe
// ---------------
// Atributos
float raio;
}
```

Como mostra o trecho de código dado, a adição de um atributo a uma classe correspondente à simples declaração de uma variável de um certo tipo, cujo nome deveria indicar seu propósito. Tal declaração deve ser colocada dentro do corpo da classe, ou seja, dentro das chaves que delimitam a declaração da classe. Muitas vezes, nos referimos aos atributos de uma classe como seus campos (*fields*) ou também como suas variáveis-membro (*members*).

Atributos, ou Propriedades (2/3)

- A porção dos dados manipulados por uma aplicação, ou classe Java.
- *Atributo* é uma propriedade nomeada de um tipo.
- A definição de *dados* ou *atributos* é idêntica à descrição dos campos de um registro, isto é, uma lista de identificadores com tipos associados.
- Define o "estado" de um objeto.
- Os atributos descrevem as características, propriedades, dos objetos analisados ⇔ Estrutura de Dados.

Atributos, ou Propriedades (3/3)

A sintaxe utilizada para definir um atributo de um objeto é:

```
[modificador] tipo nome [ = default];
```

onde:

- **modificador** (opcional), uma combinação de especifacor de acesso (public, protected ou private); final e static.
- **tipo** deve ser um dos tipos de dados da linguagem Java ou o nome de uma classe.
- **nome** deve ser um identificador válido.
- **default** (opcional) é a especificação de um "valor inicial" para a variável.

exemplos de declarações:

```
public int campoInt;                // atributo com acesso público do tipo inteiro
protected boolean campoBoolean;     // atributo com acesso protegido do tipo lógico
private char campoChar;             // atributo com acesso privado do tipo char
static byte campoByte;              // atributo com acesso estático do tipo byte
String campoString;                 // campo com acesso no pacote do tipo String
```

Métodos, ou operações (1/3)

Enquanto os atributos permitem armazenar dados associados aos objetos, ou seja, valores que descrevem a aparência ou o estado de um certo objeto, os métodos (*methods*) ou funções-membro realizam operações sobre os atributos de uma classe ou são capazes de especificar ações ou transformações possíveis para um objeto. Isso significa que os métodos conferem um caráter dinâmico aos objetos, pois permitem que os objetos exibam um comportamento que, em muitos casos, pode mimetizar (imitar) o comportamento de um objeto real ou concreto.

Outra maneira de entender o que são os métodos é imaginar que os objetos são capazes de enviar e receber mensagens, de tal forma que possamos construir programas (ou aplicações) nos quais os objetos trocam mensagens, proporcionando o comportamento desejado. A idéia central contida na mensagem que pode ser enviada ao objeto é a mesma quando ligamos um rádio: acionar o botão que liga esse aparelho corresponde a enviar uma mensagem ao rádio: “ligue”. Quando fazemos isso, não é necessário compreender em detalhes como funciona o rádio, mas apenas entender superficialmente como operá-lo ⇔ Conceito de “Caixa Petra”.

Métodos, ou operações (2/3)

- *Método* é um serviço que é requisitado a um objeto como parte de seu comportamento em resposta a estímulos (procedimento algorítmico).
- Um método é formado por uma interface e sua implementação. A interface descreve as características externas do método, sua parte visível como: *nome, parâmetros* e *valor retornado*. A implementação contém o código efetivo para a operação, isto é, uma seqüência de instruções da linguagem.
- Os métodos descrevem o comportamento, como agem e reagem os objetos analisados ⇔ Funções ou Procedimentos (Algoritmos).

Métodos, ou operações (3/3)

A forma genérica para a definição de um método em uma classe é a seguinte:

```
[modificador] tipo nome([parâmetros]) {
    // Corpo do Método
}
```

onde:

- **modificador** (opcional), uma combinação de especifacor de acesso (public, protected ou private); abstract ou final e static.
- **tipo** é um indicador do valor de retorno (void quando o método não possuir um valor de retorno = procedimento).
- **nome** do método deve ser um identificador válido.
- **parâmetros** (opcional) são representados por uma lista de parâmetros separados por vírgulas, onde cada parâmetro obedece à forma: tipo nome.

Métodos são essencialmente subrotinas que podem manipular atributos de objetos para os quais o método foi definido. Além dos atributos de objetos, métodos podem definir e manipular variáveis locais; também podem receber parâmetros por valor através da lista de argumentos. Uma boa prática de programação é manter a funcionalidade de um método simples, desempenhando uma única tarefa. O nome do método deve refletir de modo adequado a tarefa realizada.

```
// Classe elementar "Matematica.java" implementada para demonstrar
// a definição de atributos e métodos.

public class Matematica
{

   public static int a, b;

   public static int adicao() {
      return (a + b);
   }

   public static int subtracao() {
      return (a - b);
   }

   public static int multiplicacao() {
      return (a * b);
   }

   public static int divisao() {
      return (a / b);
   }
}
```

Declaração da Classe **Matematica**

Atributos, ou variáveis da Classe

"Protocolo" de **Métodos** da Classe

Corpo da Classe

Variáveis de Classe (static)

Cada objeto definido a partir de uma classe terá sua cópia separada dos atributos definidos para a classe. No entanto, há situações em que é interessante que todos os objetos compartilhem a mesma variável, similarmente ao que ocorre com variáveis globais em linguagens de programação tradicional. O mecanismo para realizar esse compartilhamento é a definição de "variáveis da classe".

Uma variável de classe tem sua declaração precedida pela palavra-chave **static** que indica, também, que não é necessário uma instância da classe para utilização dos recursos declarados como static.

Várias constantes são definidas em Java como public static final. Por exemplo, a classe Math de Java define a constante e PI (3.14159). O acesso a esses valores, basta precedê-los com o nome da classe e um ponto: **double** pi2 = Math.PI / 2;

Outro exemplo de variável public static final é a variável **out** da classe System. Essa variável, System.out, está associada a um objeto que representa a saída padrão (o monitor, tipicamente), sendo utilizada sempre que se deseja enviar

Referenciando Atributos e Métodos

Para denotar ou referenciar os atributos ou métodos de uma classe ou objeto deve-se utilizar um operador, denominado seletor, simbolizado por um caracter ponto '.' como segue:

Atributos

NomeDoObjeto, ou NomeDaClasse.nomeDoAtributo

por exemplo: Matematica.a = 10;

System.out.println(Matematica.a);

Métodos

NomeDoObjeto, ou NomeDaClasse.nomeDoMétodo([argumentos])

por exemplo: System.out.println(Matematica.adicao());

Os parênteses após o nome do método têm duplo propósito: um é diferenciar a construção ou o uso dos métodos da declaração de atributos, e o outro é "permitir" que sejam "especificados valores auxiliares que podem ser enviados" em anexo à mensagem (denominados de argumentos), para informar mais precisamente a forma como a ação deve ser realizada.

```
// Usando a classe "Matematica", que na realidade representa uma
// biblioteca de códigos, já que seus atributos e métodos são static,
// não necessitando uma instância, ou objeto da classe para
utilizar // estes recursos.

public class UsandoMatematica {

  public static void main(String args[]) {
// Fazendo referência aos atributos ou variáveis membro
// NomeDaClasse.nomeDoAtributo
    System.out.println("Informe o valor da variável (a):");
    Matematica.a = Input.readInteger();

// NomeDaClasse.nomeDoAtributo
    System.out.println("Informe o valor da variável (b):");
    Matematica.b = Input.readInteger();

// Ativando os métodos ou funções membro:
// NomeDaClasse.nomeDoMétodo()
    System.out.println(... + Matematica.adicao());
    System.out.println(... + Matematica.subtracao());
    System.out.println(... + Matematica.multiplicacao());
    System.out.println(... + Matematica.divisao());
  }

}
```

Parâmetros dos Métodos (1/2)

Na declaração dos métodos, pode ser especificada uma lista de tipos e nomes de parâmetros, tal como declarações de variáveis, que receberão os valores fornecidos aos métodos. Tal lista é denominada de lista de parâmetros formais, lista de parâmetros ou simplesmente parâmetros. Se o método receber parâmetros, os mesmos deverão ser colocados no interior dos parênteses, separados por vírgulas; caso contrário, os parênteses permanecerão vazios, esses valores são denominados de argumentos ou parâmetros reais ou efetivos.

Consideramos novamente a sintaxe geral dos métodos:

```
[modificador] tipo nome([parâmetros]) {

    // Corpo do Método
}
```

Sendo uma relação de tipos e nomes dos parâmetros que receberão o valor dos argumentos desejados, uma lista de parâmetros formais tem a forma:

Tipo1 par1, Tipo2 par2, …, TipoN parN

Parâmetros dos Métodos (2/2)

Assim, um método pode receber nenhum ou tantos argumentos quantos forem desejados. Os tipos dos parâmetros são os mesmos que podem ser empregados nas variáveis simples ou nos atributos das classes. Os nomes dos parâmetros também são arbitrários, e ressaltamos que constituem variáveis locais, pertencentes ao escopo definido pelo corpo do método, podendo ter nomes idênticos ao de variáveis locais de outros métodos.

Em Síntese:

- são variáveis opcionalmente passadas a um método
- um método pode ter zero ou mais parâmetros
- são definidos no cabeçalho do subprograma
- através da passagem de parâmetros é feita a transferência de informações entre os métodos sejam: constantes, variáveis, ou expressões, ao invés de somente o valor de variáveis ou atributos da classe.
- esta utilização formaliza a “comunicação” entre os métodos.

```java
// Demonstrando a declaração de "parâmetros" na definição de métodos.
public class Parametro {
  public static void main(String args[]) {
    System.out.println("Informe o valor da variavel (a):");
    int a = Input.readInteger();
    System.out.println("Informe o valor da variavel (b):");
    int b = Input.readInteger();
// Ativando o método e enviando o argumentos respectivos.
    System.out.println("..." + Matematica.operacao(a, '+', b));
    System.out.println("..." + Matematica.operacao(a, '-', b));
    System.out.println("..." + Matematica.operacao(a, '*', b));
    System.out.println("..." + Matematica.operacao(a, '/', b));
  }
}
```

```java
class Matematica {
// Método declarado com 3 (três) parâmetros formais.
  public static int operacao(int a, char op, int b) {
    switch (op) {
      case '+': return(a + b);
      case '-': return(a - b);
      case '*': return(a * b);
      case '/': return(a / b);
       default: return(0);
    }
  }
}
```

Valor de Retorno dos Métodos

Um outra questão relacionada a declaração de métodos (subrotinas) é a possibilidade de o método devolver mensagens ou não, isto é, "se o método é capaz de retornar algum tipo de valor".

Métodos em Java têm sua execução encerrada de duas maneiras possíveis:
1. quando um método não tem um valor de retorno (declarado com o tipo de retorno **void**): a execução é encerrada quando o bloco do corpo do método chega ao final;

```
public static void main(String args[]) {
    :
} // fim do corpo do método
```

2. encerra a execução do método através do comando **return**.

```
return;                    // sem valor de retorno.
return expressão;          // o valor de retorno é o resultado da expressão.
```

```
public static int soma(int a, int b) {
    return (a + b); // retorna a soma dos argumentos enviados
}
```

Definição do Corpo de Métodos

O corpo de um método é formado por declarações de variáveis locais e comandos da linguagem de programação delimitados por chave (‘{‘ e ‘}’).

A sintaxe de declaração de variáveis locais em Java é similar àquela de declaração de atributos de objetos, sem a opção dos modificadores de visibilidade, variáveis locais têm visibilidade restrita ao método, exclusivamente.

```
int a, b;
boolean ehPrimo = true;
```

Embora não seja obrigatório, é uma boa prática de programação manter todas as declarações de variáveis no início do método.

Uma exceção aceita refere-se a blocos delimitados por iteração com **for**, onde a forma: **for** (**int** i=1; i <=10; i++); é aceita. Neste caso, o escopo da variável de controle “i” está restrito ao bloco da iteração.

Comandos podem representar uma expressão (uma operação a ser realizada para determinar um valor) ou um comando de controle de fluxo de execução.

```
// Estrutura modular p/ exibir todos os números primos entre 1 e 100.
public class Modular {
  public static void main(String args[]) {
    for (int n=1; n<=100; n++) {
      if (primo(n) == true)    // chama a funcao "Primo" enviando o
                               // argumento "n"
         System.out.println(n); // para chamar esta função fora do
                                // escopo da classe "Modular":
                                // if (Modular.Primo(n) == true)
    }
  }

// Função que retorna verdadeiro se o valor do parâmetro formal "n"
// corresponde a um numero primo, falso, caso contrário.
  public static boolean primo(int n) {
    boolean ehPrimo = true;
    int i = 2;
    while ((ehPrimo == true) && (i <= (n / 2))) {
      if ((n % i) == 0)
         ehPrimo = false;      // encontrou um divisor, portanto,
                               // não eh primo
      else i++; // próximo divisor
    }
    return (ehPrimo); // retorna o valor da variável flag "ehPrimo"
  }
} // fim do corpo da classe "Modular"
```

2. Definição de classes em Java

Considerando novamente e, estendendo, as classes em Java são definidas através do uso da palavra-chave **class**.

Para definir uma classe, utiliza-se a construção:

```
[modificador] class NomeDaClasse {
    // corpo da classe
}
```

onde os elementos de declaração da classe representam:

- **modificador** (opcional); se presente, pode ser uma combinação de public e abstract ou final.

- **class** palavra chave da linguagem que indica a declaração de uma classe

- **NomeDaClasse** deve ser um identificador válido da linguagem.

Corpo da Classe

A definição (atributos e métodos) da classe propriamente dita está entre as chaves (‘{‘ e ‘}’). que delimitam blocos na linguagem Java.

A construção do corpo de uma classe usualmente obedece à seguinte seqüência de definição:

1. As variáveis de classe (static), iniciando pelas public, seguidas pelas protected, pelas com visibilidade padrão (sem modificador) e finalmente pelas private.

2. Os atributos (ou variáveis-membro) dos objetos dessa classe, seguindo a mesma ordenação definida para as variáveis de classe.

3. Os construtores de objetos dessa classe.

4. Os métodos da classe, geralmente agrupados por funcionalidade.

Toda classe pode também ter um método **main** associado, que será utilizado pelo interpretador Java para dar início à execução de uma aplicação.

3. Definição de classes em Java

Agora, de forma completa, são descritos os elementos sintáticos que podem ser combinados na construção, ou definição de classes em Java:

public ou sem especificador (pacote)	acesso, ou visibilidade da classe
abstract	a classe não pode ser instanciada
final	classe terminal, ou folha, a classe não pode derivar outras classes
class NomeDaClasse	nome, ou identificador da classe
extends Super	a classe é derivada, ou filha, da superclasse, ou classe pai (**Super**)
implements Interface	interfaces implementadas da classe

```
{
        // Corpo da Classe
}
```

O que é um Objeto ?

Um **objeto** é um elemento computacional que representa, no domínio da solução, alguma entidade (abstrata ou concreta) do domínio de interesse do problema sob análise. Objetos similares são agrupados em classes.

No paradigma de orientação a objetos, tudo pode ser potencialmente representado como um objeto. Sob o ponto de vista da programação, um objeto não é muito diferente de uma variável no paradigma de programação convencional. Por exemplo, quando define-se uma variável do tipo **int** em Java, essa variável tem:

- um espaço em memória para registrar o seu estado atual (um valor);
- um conjunto de operações associadas que podem ser aplicadas a ela, através dos operadores definidos na linguagem que podem ser aplicados a valores inteiros (soma, subtração, multiplicação, divisão, …).

Da mesma forma, quando se cria um objeto, esse objeto adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos, definidos pela classe) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).

Objeto, ou Instância (1/4)

- *Objeto* é uma **instância** de uma classe, é uma variável do tipo de dados definida pela classe.
- Objetos são entidades reais, existem no tempo, são mutáveis, tem estado, comportamento e podem ser criados e destruídos.
- Um programa OO consiste “somente” de objetos, que possuem métodos e dados.
- Um programa orientado a objetos é composto por um conjunto de objetos que interagem através de "trocas de mensagens". Na prática, essa troca de mensagem traduz-se na invocação de métodos entre objetos.

Classe é uma Abstração dos Objetos

- Enquanto o objeto constitui uma entidade concreta com tempo e espaço de existência.
- Classe significa, em programação, definir um modelo com os atributos e todas as operações associadas a essa entidade.
- Objeto é a representação de uma entidade real de uma determinada classe, é um exemplar ou instância da classe ⇔ Variável.

Objeto, ou Instância (2/4)

Segundo Furlan, um objeto é uma ocorrência específica (instância) de uma classe e é similar a uma entidade/ tabela no modelo relacional somente até o ponto onde representa uma coleção de dados relacionados com um tema em comum. [FURLAN, 1998]

Objeto, ou Instância (3/4)

Um objeto é uma entidade que possui:

1. **atributos**, ou *propriedades*, que descrevem o estado de um objeto no mundo real;
2. **ações**, ou *métodos*, que representam os processos associados ao objeto no mundo real;
3. um **identificador**, ou *nome*, que designa univocamente o objeto na representação.

Objeto, ou Instância (4/4)

1. “um objeto representa uma entidade, item ou unidade individual, identificável, sendo real ou abstrato, com regras bem definidas no domínio do problema.”

2. Um objeto possui **estado**, **comportamento** e **identidade**; a estrutura e o comportamento de objetos similares são definidos na sua classe comum.

Estado de um Objeto

- O estado de um objeto, engloba todas as propriedades (usualmente estáticas) de um objeto, mais os valores correntes (usualmente dinâmicos) de cada uma dessas propriedades.
- Uma propriedade, é uma característica, traço ou qualidade distinta ou inerente, que contribui para fazer com que um objeto seja único.
- Uma condição ou situação durante a vida de um objeto.

Comportamento de um Objeto

- Como um objeto age e reage, nos termos das suas mudanças de estado e passagem de *mensagens*.
- O comportamento de um objeto é completamente definido pelas suas ações.
- Uma operação é alguma ação que um objeto executa sobre outro, para obter uma reação.

Identidade de um Objeto

- É a propriedade de um objeto que o distingue de todos os outros objetos.
- Java, usa **variáveis** para distinguir objetos temporários, juntando endereçamento e identidade.
- Sistemas de **bancos de dados**, usam chaves identificadoras para distinguir objetos persistentes, juntando valores de dados e identidades.

Significado das Operações (1/2)

- Modificadora
  - uma operação que altera o estado do objeto, uma operação de escrita
- Seletora
  - uma operação que acessa sem alterar o estado de um objeto, uma operação de leitura
- De Iteração
  - uma operação que permite que partes de um objeto sejam acessadas em alguma ordem pré-definida

Significado das Operações (2/2)

- Construtora, ou de Instanciação
  - uma operação que cria um objeto e/ ou "inicializa" seu estado
  - operador **new**
- Destruidora
  - uma operação que libera o estado de um objeto e/ ou destrói o próprio objeto
  - coletor de lixo (*garbage colector*)

Instanciação (1/2)

A criação de um objeto é uma importante operação, que chamamos de "instanciação". Instanciar significa criar uma instância da classe (*class instance*), isto é, um novo objeto que pode ser descrito por meio dessa classe. Enquanto uma classe é um modelo abstrato de um objeto, uma instância representa um objeto concreto dessa classe.

Do ponto de vista da computação, a instaciação corresponde à "alocação dinâminca de memória" para armazenar informações sobre um certo objeto, ou seja, a reserva de uma porção de memória organizada internamente para guardar os valores associados aos atributos que descrevem um objeto de uma certa classe. A classe, por meio de sua definição, contém informações suficientes para que seja determinada a quantidade de memória necessária ao armazenamento de um objeto do seu tipo. O processo de ocupação de memória e de organização interna é absolutamente transparente para o programador, sendo responsabilidade da linguagem de programação utilizada. O programador, por sua vez, é quem determina quando e como instanciar novos objetos, ou seja, é o programador quem cria novos objetos para serem utilizados em um programa.

Instanciação (2/2)

Para instanciarmos um novo objeto, devemos utilizar o operador **new**, especialmente destinado à criação de novos objetos, como segue:

```
NomeDaClasse nomeDoObjeto;              // declaração do objeto
nomeDoObjeto = new NomeDaClasse();      // instanciação do objeto
```

Na primeira linha, temos a declaração de um objeto de forma semelhante à declaração de uma variável de tipo primitiva, exceto pelo fato de usarmos o nome de uma classe, em vez do nome de um tipo. Na segunda linha, temos a instanciação do objeto, na qual utilizamos o operador **new**, seguido do nome da classe que pertencerá o novo objeto. Com isso, efetuamos a criação de um novo objeto e guardamos uma referência para sua utilização na variável objeto.

De forma mais resumida, poderíamos ter escrito:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse();
```

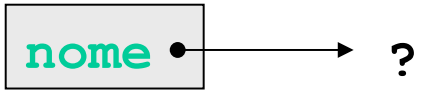
Aqui, o lado esquerdo corresponde à declaração do objeto, e o lado direito equivale à sua instanciação. Em ambos os casos, criamos um novo objeto da classe, ou seja, uma nova instância da classe, que recebe o nome que figura na declaração. A partir deste ponto, o nome do objeto pode ser utilizado para termos acesso tanto aos seus atributos como às suas operações.

Manipulação de Objetos (1/2)

Quando declara-se uma variável cujo tipo é o nome de uma classe, como em:

String **nome**;

não está se criando um objeto dessa classe, mas simplesmente uma "referência para um objeto" da classe String, a qual inicialmente não faz referência a nenhum objeto válido:



Quando um objeto dessa classe é criado, obtém-se uma referência válida, que é armazenada na variável cujo tipo é o nome da classe do objeto. Por exemplo, quando cria-se uma String:

nome = **new** String("POO/ Java");

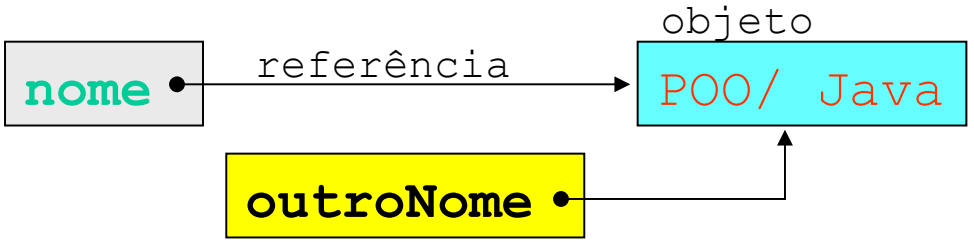
nome é uma variável que armazena uma referência para um objeto específico da classe String, objeto cujo conteúdo é POO/ Java:

Manipulação de Objetos (2/2)

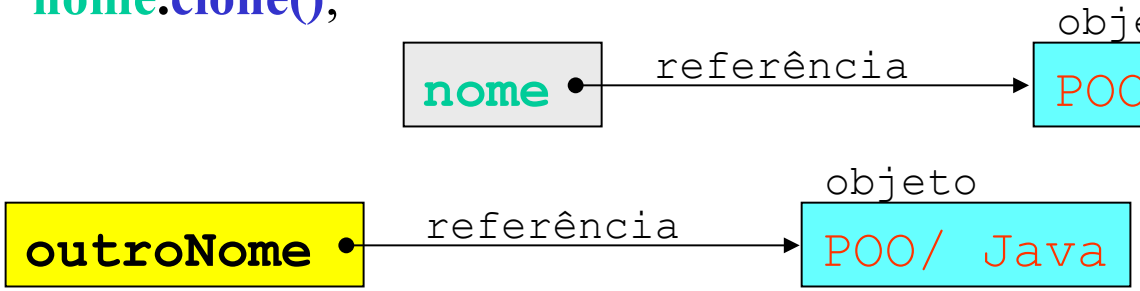
É importante ressaltar que a variável **nome** mantém apenas a referência para o objeto e não o objeto em si. Assim, uma atribuição como:

String **outroNome** = **nome**;

não cria outro objeto, mas simplesmente uma outra referência para o mesmo objeto:



O único modo de aplicar os métodos a um objeto é através de uma referência ao objeto: nomeDoObjeto.nomeDoMétodo. Seguindo com o mesmo exemplo, para criar um "novo" objeto com o mesmo conteúdo do objeto existente, utiliza-se o método **clone()**: String **outroNome** = **nome.clone()**;



Contrutores (1/2)

A criação de novos objetos, ou instanciação, tem a forma:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse();
```

O operador **new** sempre precede uma chamada, semelhante ao acionamento de um método comum, mas no qual o nome desse método é o mesmo que o da classe cujo objeto desejamos instanciar. Esse método especial é denominado "construtor do objeto" ou apenas construtor (*constructor*).

Os construtores são metodos especiais, acionados pelo sistema no momento da criação de um objeto; assim sendo, o operador **new** apenas indica que o método especial construtor de uma certa classe será utilizado. O resultado da chamada de um construtor é uma referência à área de memória na qual foi criado o objeto, ou seja, o construtor é o responsável pela alocação de memória e pelo preparo do funcionamento do objeto. Pelo construtor, podem ser inicializados seus atributos com valores consistentes e adequados ao objeto que está sendo criado e, portanto, também podem ser criados outros objetos necessários, por meio do uso de construtores de outras classes, possibilitando a construção de objetos bastantes sofisticados.

Contrutores (2/2)

Os construtores podem, efetivamente, criar objetos concretos, isto é, estruturas de dados e operações que representam uma entidade real, a partir de uma classe (um modelo abstrato de objetos de um certo conjunto). Toda e qualquer classe tem ao menos um construtor. Todos os construtores de uma classe têm, obrigatoriamente, sempre o mesmo nome da classe a que pertencem. Outro ponto importante é que os construtores sempre devem ser especificados "sem valor de retorno" e como públicos (public), uma vez que eles sempre resultam na criação de um novo objeto.

Para classes em que não exista um construtor explicitamente definido, a linguagem Java assume a existência de um construtor denominado *default*, que é um método de mesmo nome que a classe, sem parâmetros, que inicializa automaticamente os atributos existentes na classe, da seguinte forma:

Tipo da variável	Valor *default* de inicialização
byte, short, int, long, char	0
float, double	0
boolean	false
referências para objetos	*null*

Destrutores e a Coleta de Lixo (1/5)

Os destrutores (*destructor*) também são métodos especiais que liberam as porções de memória utilizada por um objeto, ou seja, enquanto os construtores são responsáveis pela alocação de memória inicial e pelo preparo do objeto, os destrutores encerram as operações em andamento liberando a memória utilizada e todos os demais recursos alocados do sistema, removendo todos os vestígios da existência prévia do objeto.

A importância dos destrutores reside no fato de que devemos devolver (alocação dinâmica) ao sistema operacional os recursos utilizados, pois, caso contrário, tais recursos podem ficar esgotados, impedindo que esse programa e os demais existentes completem suas tarefas.

Um dos maiores problemas do desenvolvimento de aplicações é justamente garantir a correta devolução dos recursos alocados do sistema, concentrando esse problema na devolução da memória ocupada, ou seja, nas operações de eliminação dos objetos que não são mais necessários. Quando um programa não devolve ao sistema a quantidade integral e memória alocada, é como se o sistema estivesse ‘perdendo’ sua memória (*memory leakage*).

Destrutores e a Coleta de Lixo (2/5)

Cientes desses problemas de gerenciamento de memória, os criadores do Java incorporaram ao projeto da máquina virtual (JVM) um gerenciador automático de memória, que é executado paralelamente ao programa Java.

Cada vez que um objeto deixa de ser referenciado, isto é, sai do escopo do programa, ficando 'perdido' na memória, é marcado para eliminação futura. Quando o programa fica ocioso, esperando por uma entrada de dados, por exemplo, o gerenciador automático de memória elimina os objetos perdidos, recuperando a memória perdida. Se a quantidade de memória disponível para a execução do programa também é esgotada, esse mecanismo procura imediatamente por objetos perdidos para recuperar a memória do sistema.

Esse inteligente mecanismo de eliminação de objetos perdidos e a conseqüente reciclagem da memória do sistema é conhecido como coletor automático de lixo *(automatic garbage collector* ou *garbage collector* ou abreviadamente, *gc*).

Destrutores e a Coleta de Lixo (3/5)

Para eliminar um objeto em Java, basta perder sua referência, o que pode ser feito por meio de uma nova inicialização da variável objeto que a referência

```
nome = new String("Omero.");      // nome recebe a referência
:
nome = new String("Alzemiro");    // nome recebe uma nova
                                  // referência e a referência
                                  // anterior é perdida
```

ou atribuindo-se o valor **null** para tal variável.

```
nome = null;
```

Dessa forma, não é necessário liberar memória explicitamente nos programas, libertanto os programadores dessa tarefa usualmente complexa e problemática.

Destrutores e a Coleta de Lixo (4/5)

A presença da coleta automática de lixo no Java torna o conceito de destrutores um pouco diferente de seus equivalentes em outras linguagens orientadas a objetos. Para todos os objetos, existem destrutores *default*, tais como os contrutores *default*. Como o conceito e o propósito desses destrutores são ligeiramente diferentes em Java, tais destrutores são denominados finalizadores (*finalizers*). Esses métodos especiais são acionados pelo sistema quando um objeto perdido é efetivamente selecionado para destruição.

Assim como para os contrutores, é possível criar explicitamente os finalizadores para a realização de tarefas mais sofisticadas. Em uma estrutura com

```
protected void finalize() {
// código para preparar a efetiva destruição do objeto
}
```

os finalizadores têm o acesso definido como protected. Eles não possuem parâmetros e não têm valor de retorno (void), podendo ser explicitamente acionados, embora isso não garanta que o objeto será imediatamente eliminado, mas apenas marcado para eliminação futura.

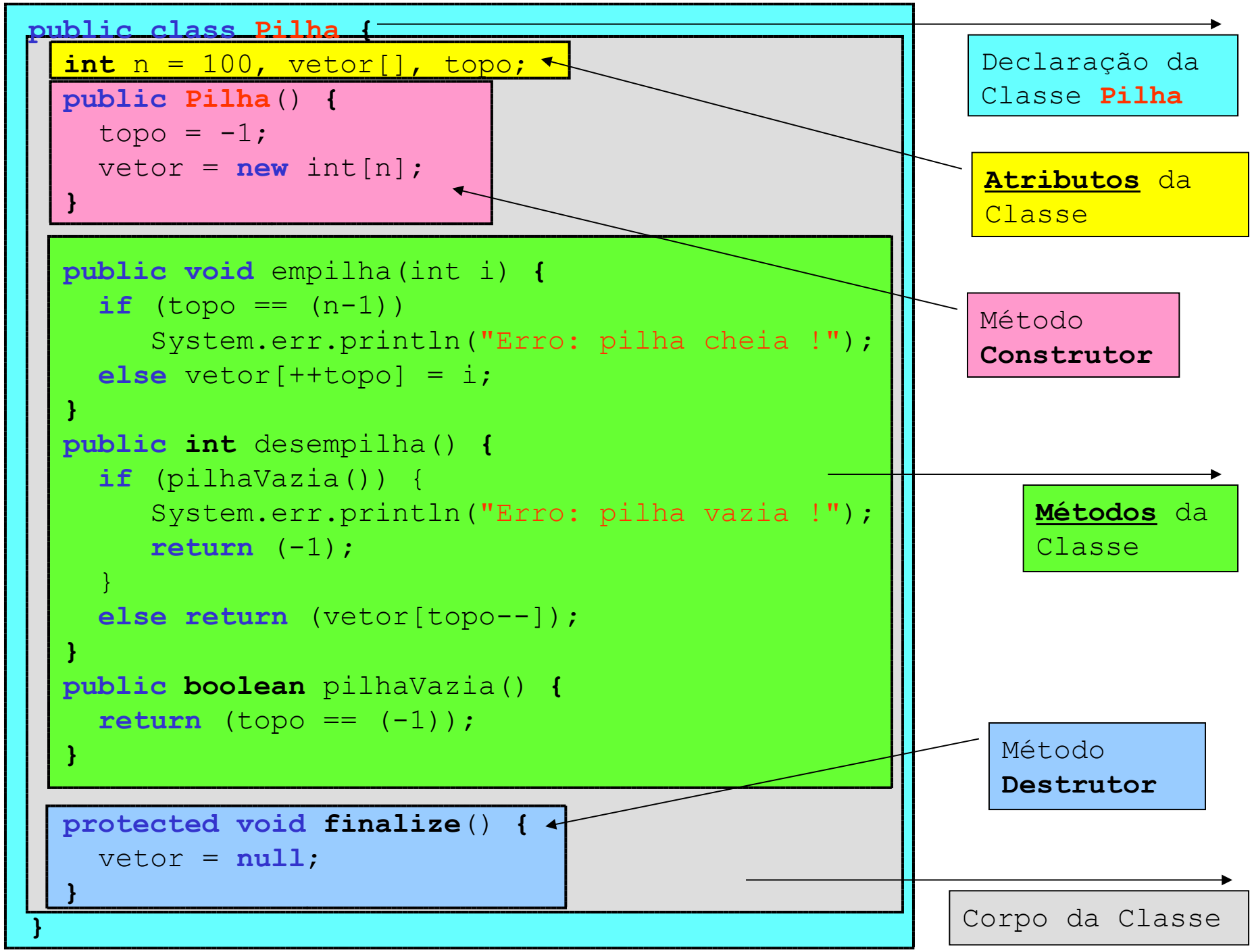
Destrutores e a Coleta de Lixo (5/5)

A coleta de lixo pode ser forçada programaticamente, isto é, pode ser acionada por meio de programas (em tempo de execução), por uma chamada explícita de um método estático existente na classe **System**, pertencente ao pacote **java.lang**:

```
System.gc();        // aciona a coleta automática de lixo
```

Atenção:

É importante ressaltar que um objeto não é imediatamente destruído ao sair do escopo do programa. O coletor de lixo apenas efetua uma marcação para eliminação futura, isto é, ele assinala o objeto fora do escopo para ser eliminado quando se torna necessário recuperar memória para o sistema, evidenciando o caráter assíncrono da coleta de lixo.


public class Pilha {
int n = 100, vetor[], topo;
public Pilha() {
topo = -1;
vetor = new int[n];
}
public void empilha(int i) {
if (topo == (n-1))
System.err.println("Erro: pilha cheia !");
else vetor[++topo] = i;
}
public int desempilha() {
if (pilhaVazia()) {
System.err.println("Erro: pilha vazia !");
return (-1);
}
else return (vetor[topo--]);
}
public boolean pilhaVazia() {
return (topo == (-1));
}
protected void finalize() {
vetor = null;
}
}
Declaração da Classe Pilha
Atributos da Classe
Método Construtor
Métodos da Classe
Método Destrutor
Corpo da Classe

```
public class UsandoPilha {
// corpo da classe
  public static void main(String args[]) {
    int x;
// declara e instancia o objeto "objPilha" a partir da classe "Pilha"
    Pilha objPilha = new Pilha();
    while (true) {
      System.out.println("Informe um valor inteiro:");
      System.out.println("obs. (-1) para encerrar.");
      x = Input.readInteger();
      if (x == (-1))
         break;
// ativa o método "empilha" para empilhar o item 'x' no topo da pilha
      objPilha.empilha(x);
    }
    System.out.println("Imprimindo os itens armazenados na Pilha");
// enquanto a pilha não estiver vazia, retira o item que esta
// no topo da pilha
    while (! objPilha.pilhaVazia()) {
      x = objPilha.desempilha();
      System.out.println(x);
    }
// ativa o método destrutor para liberar os recursos (vetor) do
// objeto o coletor de lixo marcar o objeto para eliminação futura
    objPilha.finalize();
  }
}
```

Ciclo de Vida de um Objeto

- (1) declara o objeto a partir da classe, (2) instância ou cria o objeto e (3) perder a referência ao objeto criado (o coletor de lixo marca o objeto para eliminação futura)
- O tempo de vida de um objeto vai desde o tempo em que ele é instanciado ou criado (assim consumindo espaço), até quando sua referência é perdida (*garbage collector* recupera o espaço).
- Persistência = armazenamento permanente dos objetos em disco.

Em Síntese: Classe X Objeto

Classe = grupo de objetos similares que compartilham atributos e comportamento semelhantes.

```
public class MinhaClasse
{
  // corpo da Minha Classe
}
```

Objeto = uma instância, ou uma ocorrência, ou um exemplar de uma Classe.

```
MinhaClasse meuObjeto = new MinhaClasse();
```

Resumindo: O que é ! E o que não é uma Classe !

Os conceitos de classe e objeto são intercalados, já que não podemos falar sobre um objeto sem mencionar de qual classe ele é uma instância. Enquanto um objeto é uma entidade concreta que existe no tempo e espaço, uma classe representa somente uma abstração, a "essência" de um objeto como ele é. Por exemplo, se estamos falando da classe dos mamíferos, precisamos identificar qual especialmente. Para isso, precisamos falar sobre "este mamífero" ou "aquele mamífero".

No contexto da POO, podemos definir:
Uma classe como um "conjunto de objetos que compartilham estrutura e comportamento comuns". [BOOCH, 1991]
Um objeto é uma entidade do espaço-tempo, ou simplesmente uma instância (ou um exemplar) da classe.

Concluindo então, um Objeto **não** é uma Classe.

package (pacote)

No desenvolvimento de pequenas atividades ou aplicações Java, é viável manter o código e suas classes no diretório corrente. No entanto, para grandes aplicações é preciso organizar as classes de maneira a:

1. evitar problemas com nomes duplicados de classes, e
2. localizar o código da classe de forma eficiente.

Em Java, a solução para esse problema está na organização de classes e interfaces em pacotes (package).

Um pacote ou package em Java nada mais é do que um conjunto de classes (*framework*). Usualmente, são colocadas em um package classes relacionadas, construídas com um propósito comum; assim, sob certos aspectos, os *packages* reproduzem a idéia das bibliotecas de código (*libraries* e *unit's*), de outras linguagens de programação.

Definindo um pacote (1/2)

Para definirmos um pacote, basta criar um diretório apropriado e adicionar uma ou mais classes ao mesmo. As classes que serão adicionadas devem ser iniciadas pela diretiva package, seguida do nome do pacote, tendo a sintaxe:

package nomedopacote;

Exemplo da criação de uma classe pertencente a um pacote definido pelo usuário.

```
package pacotes.java; // pacotes.java.FuncaoMatematica.java
public class FuncaoMatematica {
  public static int Fatorial(int n) {
    int f = 1;
    for (int i=1; i<=n; i++)
      f *= i;
    return (f);
  }
}
```

Definindo um pacote (2/2)

Assim sendo, a classe **FuncaoMatematica** passa a pertencer ao pacote **pacotes.java**, ou seja, seu nome completamente qualificado é:

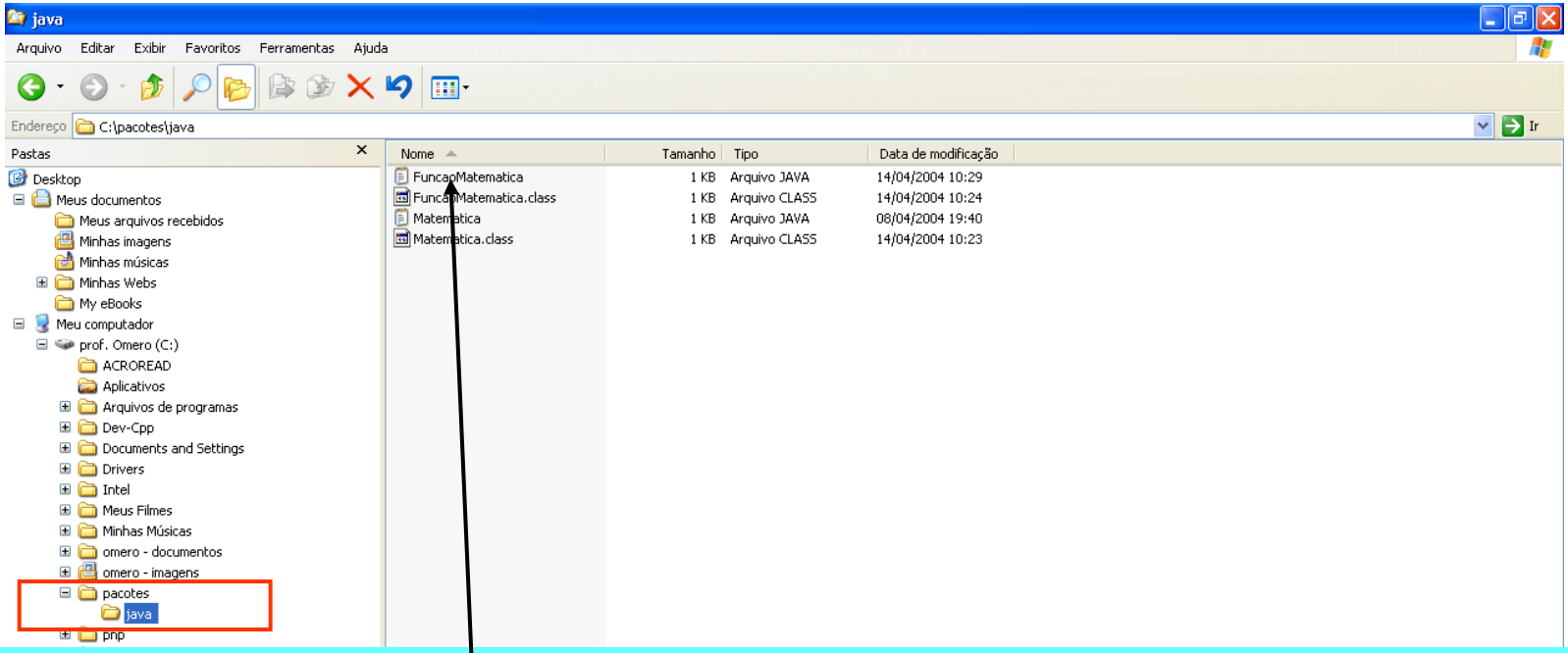
pacotes.java.FuncaoMatematica

O arquivo **.class** gerado deve, obrigatoriamente, estar localizado em uma árvore de diretórios de nome **pacotes\java** para que essa classe possa ser utilizada.

Para facilitar esse procedimento, o compilador pode ser informado do nome do diretório que deve ser utilizado como raiz para o diretório pacotes. Veja o exemplo da compilação através da janela do prompt do MS-DOS:

```
path = C:\...\jdk1.6.0_04\bin;c:\pacotes\java;%path%
javac –d FuncaoMatematica.java
```

Diretório do pacote de classes "pacotes.java".




```java
// pacotes.java.FuncaoMatematica.java

package pacotes.java;

public class FuncaoMatematica {

  public static int Fatorial(int n) {
    int f = 1;
    for (int i=1; i<=n; i++)
      f *= i;
    return (f);
  }
}
```

Importando Classes no Java (1/3)

Para importar uma ou mais classes de um pacote conhecido, utilizamos a diretiva import (importar), que tem a seguinte sintaxe:

```
import nomedopacote.nomedosubpacote.NomeDaClasse;
```

Os pacotes incluídos no Java, isto é, aqueles que fazem parte da API (*Application Programming Interface*) padrão, são sempre iniciados pela palavra **java** (reservada para denominação de pacotes da *Sun*) seguida do nome do pacote e do nome da classe desejada, "separada por um ponto", tal como:

```
import java.awt.Button;
```

Assim, estamos indicando que desejamos utilizar a classe (ou componente) **Button**, destinada a execução de alguma seqüência de comandos em aplicações com interface gráfica, que é oferecida pelo Java por meio do pacote **java.awt** (*Abstract Window Toolkit*).

Importando Classes no Java (2/3)

Quando “várias classes” do mesmo pacote serão importadas, é mais prático utilizar a seguinte forma:

```
import java.awt.*;
```

A denominação dos pacotes reflete, na verdade, a estrutura de diretórios utilizada para a organização das classes.

Quando indicamos o nome **java**, estamos informando ao compilador que as classes da API padrão devem ser utilizadas; assim, ele as procura no diretório apropriado de sua instalação.

A importação de classes em pacotes, definidas pelo usuário, é semelhante, diferindo apenas na localização dos diretórios nos quais tais classes estão armazenadas.

Importando Classes no Java (3/3)

Essencialmente, uma classe **Xyz** que pertence a um pacote nomedopacote tem o nome completo nomedopacote**.Xyz** e o compilador Java espera encontrar o arquivo **Xyz.class** em um subdiretório nomedopacote. Este, por sua vez, deve estar localizado sob um dos diretórios especificados na variável de ambiente **CLASSPATH**.

Mesmo que apenas uma classe seja importada, a forma:

import nomedopacote**.*;**

simplifica a redação do programa, e não implica na importação de classes desnecessárias. As classes importadas não são incluídas nas classes que as utilizam, pois cada classe Java é sempre um arquivo independente, e são utilizadas assim.

Regras para a Denominação de Pacotes

Os nomes dos pacotes devem utilizar apenas letras minúsculas, podendo ser compostos de várias partes separadas por um ponto (**.**), tal como **java.awt**.

Para facilitar o compartilhamento de pacotes que serão distribuídos, recomenda-se que a denomição dos pacotes utilize o nome reverso do domínio na Internet da organização que originou o pacote (Gosling, 2000, pág. 115), agregado ao nome do pacote, pois os nomes de domínios na Internet são únicos.

Assim, um pacote que desejamos denominar como **toolbox**, originado pela empresa XYZ, cujo domínio é **www.xyz.com.br**, teria o seguinte nome: **br.com.xyz.toolbox**.

Mesmo que uma outra empresa resolva criar e distribuir outro pacote de nome **toolbox**, sendo seu domínio na Internet diferente, não haverá ‘colisão’ entre os nomes desses pacotes, que poderão ser instalados e utilizados em um mesmo sistema.

Vocabulário da POO, Parte II

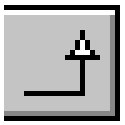
- Herança
- Acessibilidade (especificadores de acesso)
- Encapsulamento
- Classe final
- Classes Abstratas
- Interfaces

Relacionamento entre Classes, ligam as classes/objetos entre si criando relações lógicas entre estas entidades

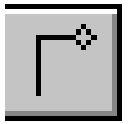
Há quatro tipos básicos de relacionamentos entre classes:
1. generalização: superclasse (pai) - subclasse (filho)
2. agregação: todo - parte
3. associação: classes não correlatas, cliente - pedidos
4. dependência: independente - dependente

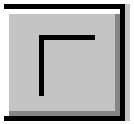
Muitas técnicas em linguagens de programação foram desenvolvidas para expressar os relacionamentos entre classes:
herança (generalização, agregação e associação)
uso, **instanciamento** (criando objetos) e
metaclasses.

Relacionamentos (1/3)

- Generalização/ especificação 
  - indica relacionamento entre um elemento mais geral e um elemento mais específico (respectivamente, superclasse e subclasse)
  - também conhecido como herança ou classificação (por exemplo, um enfermeira “é um tipo de” pessoa)

Relacionamentos (2/3)

- Agregação 
  - usada para denotar relacionamentos todo/parte
  - por exemplo, um Item de compra é parte de um Pedido

- Associação 
  - é um relacionamento que descreve um conjunto de vínculos, onde cada vínculo é definido como uma conexão semântica entre tuplas de objetos
  - por exemplo, um Cliente possui Conta Corrente

Relacionamentos (3/3)

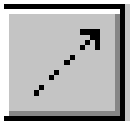
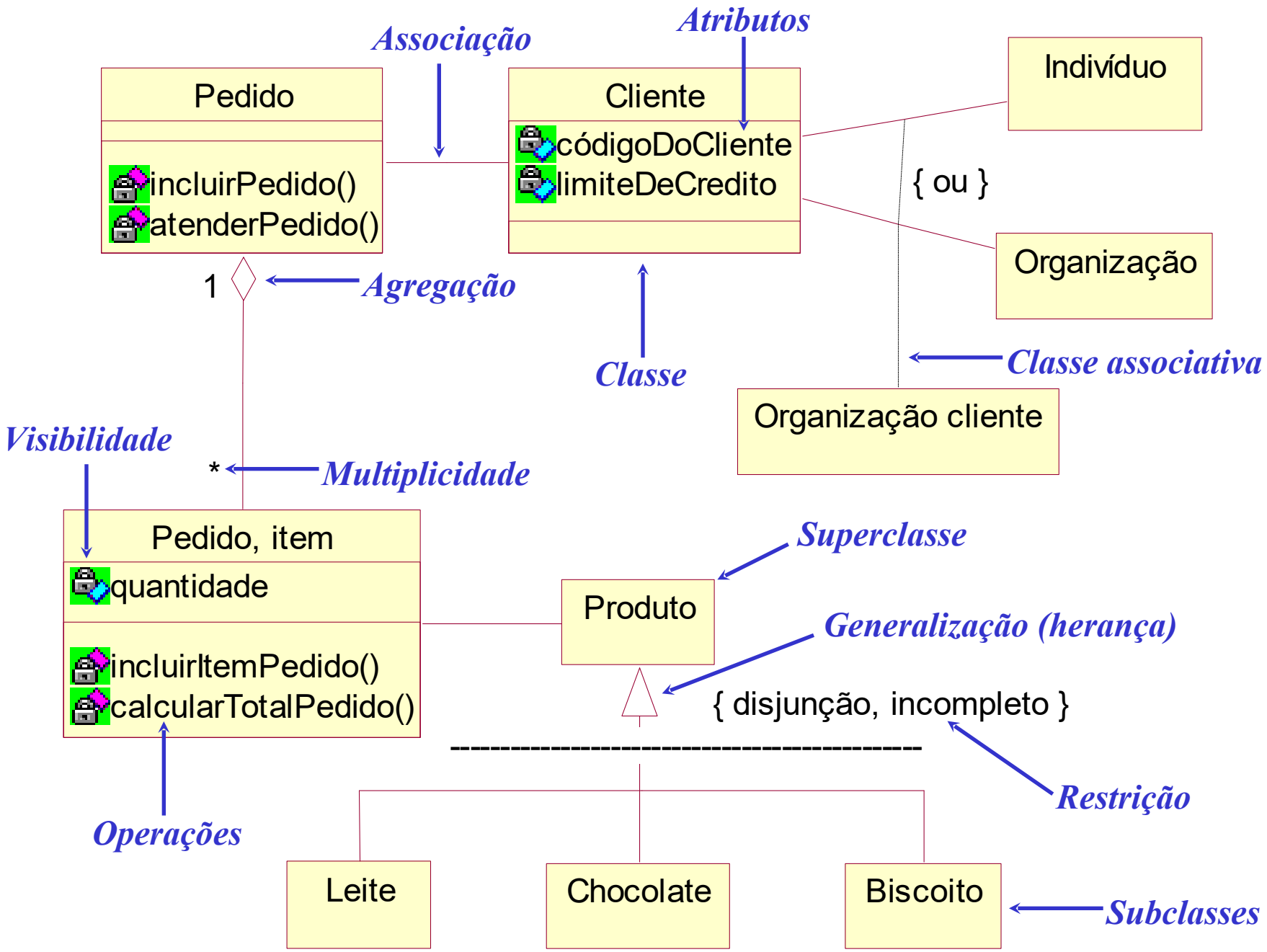
- Dependência 
  - é um relacionamento entre elementos, um independente e outro dependente
  - uma modificação no elemento independente afetará diretamente os elementos dependentes do anterior
  - é um relacionamento de utilização

Diagrama de Classes UML



Herança

A herança (*inheritance*) pode ser entendida como uma técnica para que uma classe passe a utilizar atributos e métodos, ou operações, definidos em outra classe.

Rigorosamente falando, a herança é o compartilhamento de atributos e métodos entre classes, com base em um relacionamento hierárquico do tipo 'pai e filho', ou seja, a classe pai é tomada como referência, contendo definições que poderão ser utilizadas nas classes definidas como filhas.

A classe pai é o que se denomina classe base (*base class*) ou superclasse (*superclass*), e as classes filhas são também chamadas de classes derivadas (*derived classes*) ou subclasses (*subclasses*).

Implementando a Herança

Em Java, a indicação da construção de uma classe pela herança é feita juntamente com sua declaração por meio da palavra reservada **extends**, seguida do nome da classe que será utilizada como base, conforme o trecho esquemático de código, apresentado a seguir. A superclasse não recebe nenhuma indicação especial, podendo ser utilizada para a construção de tantas subclasses quantas forem desejadas.

```
public class SuperClasse {
// Corpo da SuperClasse
}


public class SubClasse extends SuperClasse {
// Corpo da SubClasse
}
```

Herança (1/4)

- Todos os objetos criados a partir da mesma classe vão compartilhar (herdar) as características descritas por ela.
- Usando a herança é possível montar uma classe nova (subclasse ou classe filho), definindo-a em termos de outra classe existente (superclasse ou classe pai).

Herança (2/4)

- Uma subclasse herda os atributos e métodos da superclasse.
- Na subclasse é possível adicionar novos métodos e variáveis de instâncias aos originais, além de redefinir ou modificar a implementação dos métodos herdados sem que a superclasse precise ser modificada.

Herança (3/4)

- A herança permite que novas classes sejam criadas e adicionadas a uma hierarquia. Na raiz da hierarquia, temos a classe base ou superclasse da hierarquia e, no outro extremo, temos novas subclasses que podem ampliar, especializar ou mesmo restringir o comportamento das classes originais.
- Observando tal hierarquia a partir da raiz, ocorre a generalização das diferentes classes relacionadas. Isso significa que todas as classes pertencentes a uma mesma família de classes podem ser genericamente tratadas como uma classe do tipo mais primitivo existente, ou seja, da própria superclasse da hierarquia.

Herança (4/4)

- Especialização, você pode considerar uma subclasse, ou classe filha, um caso especial da superclasse, ou classe pai.
- *Generalização*, você pode usar a classe pai para *compartilhar os elementos comuns*, e também parte do código.
- Na prática, a herança é uma maneira de evitar repetição de código.

Compartilhamento e Especificação (1/4)

A idéia central da herança é de que os atributos e operações definidos em uma certa classe podem ser utilizados em outras classes.

Isso representa um mecanismo sofisticado de compartilhamento, no qual muitas classes passam a compartilhar um mesmo conjunto de atributos e operações.

Como cada classe derivada pode adicionar outros atributos e operações específicos, a herança também é, portanto, um mecanismo de extensão, pois uma dada subclasse tem tudo aquilo definido pela sua superclasse, além de atributos e operações localmente adicionados para implementar um novo comportamento.

Compartilhamento e Especificação (2/4)

Atributos e operações definidos na classe base não precisam ser repetidos em uma classe derivada, dessa forma, a orientação a objetos auxilia a redução da repetição de código dentro de um programa, ao mesmo tempo em que possibilita que classes mais genéricas sejam reutilizadas em outros projetos, sob a forma de *toolkit*, o que compõe um dos aspectos da possibilidade de reutilização de código.

Quando o projeto de uma hierarquia de classe é suficientemente genérico e amplo, caracteriza-se o desenvolvimento de uma solução generalizada. Com isso, torna-se possível mais do que a simples reutilização de código, mas também a reutilização do projeto. Uma hierarquia de classes que pode ser genericamente utilizada em outros projetos chama-se *framework*.

Compartilhamento e Especificação (3/4)

Outro aspecto importante é que o fato de uma subclasse herdar um conjunto de atributos e operações não implica em que os mesmos são imutáveis; ou seja, é possível a modificação dos membros herdados por meio da sobrecarga (*overload*) ou da sobreposição (*overhide*).

A sobrecarga, permite que novas formas de uma mesma operação sejam implementadas, ou seja, podem ser adicionados nas subclasses métodos com os mesmos nomes, desde que suas assinaturas sejam diferentes.

A "sobreposição" é uma operação restrita a classes construídas por herança, na qual "atributos e/ou operações declarados na superclasse são substituídos por outros", declarados nas subclasses.

Compartilhamento e Especificação (4/4)

Dessa forma, o mecanismo de especialização pode conduzir a duas situações distintas:

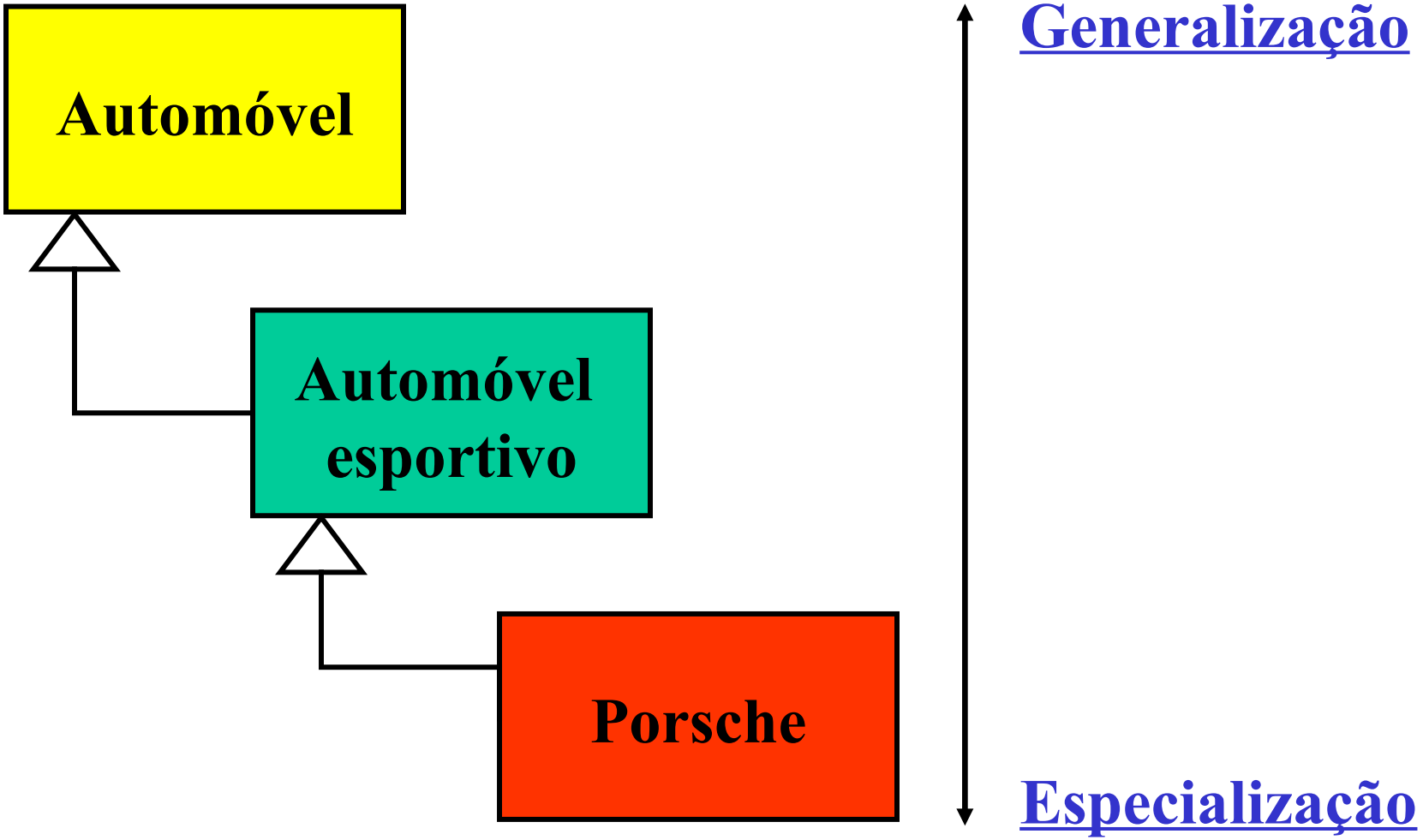
1. Extensão

A herança pode ser utilizada para a construção de novas classes que ampliam, de forma especializada, as operações e os atributos existentes na classe base. Com isso, temos a adição de novos elementos a uma classe.

2. Restrição

As subclasses podem ocultar ou alterar de modo consistente operações e atributos da superclasse (*overriding*), sem modificar a interface proposta. Com isso, temos a modificação de elementos já existentes numa classe para a adequação com os novos elementos adicionados.

Hierarquia de Classes



Resumo Herança

SuperClasse Classe Pai Classe Base Generalização	**[e]**	SubClasse Classe Filho Classe Derivada Especialização

Reutilização de Código

Classe Object (1/2)

A classe **java.lang.Object** é a superclasse de toda a hierarquia de classes Java, ou é "raiz" a partir da qual todas as classes são definidas. Desse modo, os métodos dessa classe estão disponíveis para objetos de todas as demais classes.

- protected Object clone()
  criar e retorna um "novo" objeto com o mesmo conteúdo do objeto existente

- public boolean equals(Object obj)
  permite comparar objetos por seus conteúdos

- protected void finalize()
  método destrutor, faz o objeto perder a "referência"

Classe Object (2/2)

- public String toString()

  permite converter uma representação interna do objeto em uma *string* que pode ser apresentada ao usuário

- public final Class getClass()

  retorna um objeto que representa a classe à qual o objeto pertence.

  A partir do objeto retornado, da classe **java.lang.Class**, é possível obter:
  - o nome da classe usando o método getName(); e
  - o nome da superclasse usando o método getSuperclass();

  estes métodos retornam uma String com o nome da classe ou da superclasse, respectivamente.

Invocando, ou Chamando Construtores

Java é muito organizada com relação à inicialização de objetos, com o intuito de garantir a segurança. Quando uma instância de um determinado objeto é criada, ocorre a seguinte seqüência de ações:

1. O espaço para o objeto é alocado e seu conteúdo é inicializado (*bitwise*) com zeros ou valores *default*.
2. O construtor da classe base é invocado.
3. Os membros da classe são inicializados para o objeto, seguindo a ordem em que foram declarados na classe.
4. O restante do corpo do construtor é executado.

Seguir essa seqüência é uma necessidade de forma a garantir que, quando o corpo de um construtor esteja sendo executado, o objeto já terá à disposição as funcionalidades mínimas necessárias, quais sejam aquelas definidas por seus ancestrais.

O primeiro passo garante que nenhum campo do objeto terá um valor arbitrário, que possa tornar erros de não inicialização difíceis de detectar.

Construção de Objetos Derivados

Durante a construção de um objeto de uma classe derivada, o construtor de sua superclasse é executado (implicitamente ou explicitamente) antes de executar o corpo de seu construtor.

Assim, ao se construir um objeto de uma classe derivada, o método construtor da superclasse será inicialmente invocado. Este, por sua vez, invocará o construtor de sua superclasse, até que o contrutor da classe raiz de toda a hierarquia de objetos, a classe Object, seja invocado.

Como Object não tem uma superclasse, seu construtor é executado e a execução retorna para o construtor de sua classe derivada. Então executa-se o restante do construtor de sua classe derivada e a execução retorna para o construtor de sua classe derivada e assim sucessivamente, até que finalmente o restante do construtor da classe para a qual foi solicitada a criação de um objeto seja executada.

Construtores da superclasse podem ser explicitamente invocados usando a palavra-chave super (definida na seção sobre polimorfismo).

Conceitos adicionais

Adicionalmente, ao conceito de herança, há os conceitos de dependência e coesão de classes.

1. **Dependência** refere-se ao conhecimento que uma classe possui de outra - o objetivo é o de minimizar a dependência para evitar impactos em uma classe decorrentes de modificações em outra classe.

2. **Coesão** é uma medida de integridade conceitual de uma classe - o objetivo, nesse caso, é o de maximizar a coesão para assegurar agrupamento de operações em classes base ou gerais. O tratamento generalizado de classes permite escrever novas classes que podem ser mais facilmente extensíveis.

Contração

A contração é uma uma variante do mecanismo de herança onde a subclasse elimina métodos da superclasse com o objetivo de criar uma “classe mais simples”.

A eliminação pode ocorrer pela redefinição de métodos com corpo vazio. O problema com este mecanismo é que ele viola o princípio da substituição, pois a subclasse já não pode mais ser utilizada em todos os pontos onde a superclasse poderia ser utilizada.

Se a contração parece ser uma solução adequada em uma hierarquia de classes, provavelmente a hierarquia deve ser re-analisada para detecção de inconsistências (problema pássaros-pinguins). De modo geral, o mecanismo de contração deve ser evitado.

Classe Interna (inner class) (1/4)

Uma classe interna é uma classe que é subordinada a uma outra classe, sendo definida no corpo dessa outra classe. Dependendo de onde e como se dá essa definição, a classe interna pode ser de diferentes tipos.

Uma classe membro tem seus objetos sempre associados a objetos da classe topo. Os objetos da classe membro têm acesso aos membros da classe topo. Em geral, esse tipo de classe define algum tipo de serviço de apoio que complementa a funcionalidade da classe topo. Uma classe membro não pode conter membros internos que sejam static.

```
public class ClasseTopo {
  // definição de membros (atributos e métodos)

  protected class ClasseMembro {
    // membros internos
  }

} // fim do corpo da classe "ClasseTopo"
```

Classe Interna (inner class) (2/4)

Uma classe membro estática (ou uma interface membro estática) é similar a uma classe membro, mas tem acesso apenas aos membros declarados como static na classe topo.

Sendo estática, seus membros internos são acessíveis pela classe topo ou por outras classes membros estáticas.

```
public class ClasseTopo {
  // definição de membros (atributos e métodos)

  public static class ClasseMembroEstática {
    // membros internos
  }

} // fim do corpo da classe "ClasseTopo"
```

Classe Interna (inner class) (3/4)

Uma classe local é um tipo de classe interna definida no escopo de um bloco de código, ou seja, dentro de um método. Além das facilidades associadas a uma classe membro, um objeto de uma classe local tem acesso a variáveis do método que tenham sido declaradas como final.

```
public class ClasseTopo {
  // definição de membros (atributos e métodos)
  public class UmaClasse {
    public UmaClasse umMetodo() {
      // ...
      class ClasseLocal extends UmaClasse {
        // definição da classe local
      } // fim do corpo da classe "ClasseLocal"
      ClasseLocal obj = new ClasseLocal();
      // ...
      return (obj);
    } // fim do corpo do método "umMetodo"
  } // fim do corpo da classe "UmaClasse"
} // fim do corpo da classe "ClasseTopo"
```

Classe Interna (inner class) (4/4)

Classes anônimas oferecem uma forma de definir uma classe local para a qual apenas um objeto será criado. Neste caso, o nome da classe é irrelevante e pode portanto ser omitido.

A sintaxe para definição de uma classe anônima já combina a criação de um objeto dessa classe.

```
public class ClasseTopo {
  // definição de membros (atributos e métodos)

  public UmaInterface outroMetodo() {
    // ...
    return new UmaInterface() {
      // definição da classe local, anônima
    };
  } // fim do corpo do método "outroMetodo"

}  // fim do corpo da classe "ClasseTopo"
```

Glossário

- Classe básica, classe mais geral em uma estrutura (hierarquia) de classes (Object no Java e TObject no Delphi).
- SuperClasse, classe que servirá de base para outras classes.
- SubClasse, classe derivada da superclasse.
- Framework, hierarquia de classes que pode ser genericamente utilizada em outros projetos.

Uma classe tem tipicamente dois tipos de clientes:

1. **instâncias, ou objetos**
2. **subclasses**

```
// Classe a ser instanciada, ou ainda:
// classe base, classe pai ou superclasse.
public class MinhaClasse
{
  // corpo da MinhaClasse
}

public class MinhaSubClasse extends MinhaClasse
{
  // corpo da MinhaSubClasse
}

// instanciação, ou criando o objeto
MinhaClasse meuObjeto = new MinhaClasse();
```

Visões de uma Classe

- Visão externa
  - interface da classe: declarações das propriedades e operações aplicáveis aos objetos da classe, podem incluir declarações de outras classes, constantes, variáveis e exceções necessárias para completar a abstração.

- Visão interna
  - implementação do comportamento: operações pré-definidas na interface da classe.

Acessibilidade (1/3)

A acessibilidade de classes, métodos e atributos é a forma com que tais elementos podem ser vistos e utilizadados por outras classes.

Este conceito é mais conhecido como encapsulamento ou *data hiding* (ocultação de dados), sendo muito importante dentro da orientação a objetos.

A determinação da acessibilidade de uma classe ou membro de classe é feita pelos especificadores de acesso (*access specifiers*), ou seja, por palavras reservadas da linguagem, que definem o grau de encapsulamento exibido por uma classe e seus elementos, isto é, são especificações para a restrição do acesso às declarações pertencentes a uma classe ou à própria classe com um todo.

Acessibilidade (2/3)

Restringido o acesso a certas partes de uma classe, possibilita-se um controle mais sofisticado de seu uso, o que oferece algumas vantagens:

• o usuário da classe (por exemplo, aquele que instancia objetos desse tipo) passa a conhecer apenas aquilo que é necessário para o uso dos objetos criados.

• detalhes da forma de implementação são ocultos dos usuários da classe, permitindo resguardar esforços de desenvolvimento ou tecnologias proprietária.

• pode-se assegurar que certos atributos assumam valores restritos a um conjunto desejado, mantendo-se, dessa forma, a consistência e a integridade dos dados armazenados na classe.

Acessibilidade (3/3)

As vantagens agora citadas são possíveis porque, por meio dos especificadores de acesso existentes, podemos ocultar algumas partes da classe, enquanto outras partes selecionadas são visíveis aos seus usuários (objetos ou subclasses).

Os especificadores de acesso são definidos pelo uso das palavras reservadas:

• public (público)	atributos e métodos podem ser livremente acessados
• *sem especificador*	a todos os membros que fazem parte
(visibilidade padrão)	do mesmo pacote
• protected (protegido)	só podem ser acessados pela classe e por novas classes criadas por meio do mecanismo de herança
• private (privado)	uso exclusivo da classe

public (público) (1/2)

Em Java, a visibilidade padrão de classes, atributos e métodos está restrita a todos os membros que fazem parte de um mesmo pacote.

A palavra-chave **public** modifica essa visibilidade de forma a ampliá-la, deixando-a sem restrições.

Uma classe definida como pública pode ser utilizada por qualquer objeto de qualquer pacote. Em Java, uma unidade de compilação (um arquivo fonte com extensão .java) pode ter no máximo uma classe pública, cujo nome deve ser o mesmo do arquivo (sem a extensão). As demais classes na unidade de compilação, não públicas, são consideradas "classes de suporte" para a classe pública e têm a visibilidade padrão.

public (público) (2/2)

Atributos é métodos públicos de uma classe representam as declarações da "interface" de uma classe que são visíveis a qualquer parte do ambiente de referenciamento (escopo) da classe, ou seja, a própria Classe, as classes derivadas ou Subclasses e as Instâncias ou objetos.

```
Declaração de atributos públicos:
public static int a, b;
public float raio;

Declaração de métodos públicos:
public void empilha(int i) { ... }
public int desempilha() { ... }
public boolean pilhaVazia() { ... }
```

protected (protegido)

O especificador de acesso correspondente a palavra-chave **protected** restringe a visibilidade do membro modificado, atributo ou método, apenas à própria classe e àquelas derivadas (subclasses) desta.

```
Declaração de atributos protegidos:
protected int n, tamanho;


Declaração de métodos protegidos:
protected boolean anoBissexto(int a) { ... }
protected int diasNoMes(int m) { ... }
```

private (privado)

O especificador de acesso correspondente a palavra-chave private restringe a visibilidade do membro modificado, atributo ou método, exclusivamente a classe, e se relacionam com sua implementação e com aspectos que não desejamos divulgar.

```
Declaração de atributos privados:
private int n = 100, topo, vetor[];
private int dia, mes, ano;


Declaração de métodos privados:
private int totalDeDias() { ... }
```

Acessibilidade da Interface da Classe

1. **public** (pública)

   uma declaração que é parte da interface de uma classe e é visível a qualquer parte do ambiente de referenciamento (escopo) da classe

   (Classe, Subclasses e Instâncias)

2. **protected** (protegida)

   uma declaração que forma parte da interface de uma classe, mas não é visível para quaisquer outras classes ou instâncias, exceto suas subclasses.

   (Classe, Subclasses)

3. **private** (privada)

   uma declaração que forma parte da interface de uma classe, mas não é visível para quaisquer outras classes ou instâncias.

   (Classe)

Encapsulamento (1/3)

- Consiste em separar os aspectos externos de um objeto, que são acessíveis para outros objetos, dos detalhes internos de implementação do objeto. [Rumbaugh]
- Qualquer mecanismo que nos permita "esconder" a implementação do objeto fazendo com que outros componentes do sistema não tenham conhecimento do conteúdo interno dos dados armazenados no objeto. [Yourdon]

Encapsulamento (2/3)

- É o resultado (ou ato) de se esconder os detalhes de implementação de um objeto do usuário. [Martin]
- O "**empacotamento**" de dados (atributos) e operações (métodos, funções ou serviços) em um objeto é chamado de encapsulamento.
- Impede que o cliente enxergue os detalhes de implementação do objeto, o que é chamado de ocultação da informação (*information hiding*).

Encapsulamento (3/3)

- Conseqüências:
  - comunicação entre os objetos é feita através de mensagens enviadas para as operações
  - implementação de um objeto pode ser mudada sem afetar as aplicações que o usam
  - facilidade de depuração, portabilidade, etc
- Lei da OO
  - os dados não podem ser acessados diretamente (são encapsulados)

Vantagens do Encapsulamento

• o código torna-se mais claro, pois ficam evidentes os membros que oferecem funcionalidades reais daqueles que são auxiliares;

• classes semelhantes podem exibir interfaces semelhantes, pois os detalhes de implementação permanecem ocultos, facilitando sua utilização;

• é oferecida facilidade de modificação (como a classe é conhecida pela sua interface, modificações que não afetem a interface são transparentes para os usuários da classe).

Tais benefícios são explorados na criação de famílias de componentes de software reutilizáveis, que são robustos, fáceis de usar e ainda permitem a criação de extensões que os modifiquem conforme desejado, conferindo grande versatilidade à sua utilização. Os pacotes, ou bibliotecas de componentes para interface gráfica para usuário e de capacidades gráficas que constituem a AWT (*Abstract Window Toolkit*) e a JFC (*Java Foundation Classes*), são alguns exemplos de conjuntos reutilizáveis de componentes Java que exploram tais conceitos e benefícios.

Classe final

Classe terminal ou folha. Uma classe definida com a palavra-chave final não pode ser estendida, ou seja, ela não pode ser usada para derivar outras classe.

```
public final class ClasseFolha {
:
}
```

```
// A declaração da classe NovaClasse vai
// ocassionar um erro de compilação porque
// ClasseFolha foi declarada como final,
// portanto, não pode ser estendida.
public class NovaClasse extends ClasseFolha {
:
}
```

Classes Abstratas (1/3)

Em algumas circunstâncias, desejamos orientar como uma classe deveria ser implementada, ou melhor, como deveria ser a interface de uma certa classe. Em outros casos, o modelo a ser representado é tão amplo, que certas classes se tornam gerais, não sendo possível ou razoável que elas sejam implementadas, ou mesmo que possuam instâncias. Para esses casos, dispomos das classes abstratas (*abstract classes*).

O uso das classes abstratas é dirigido para a construção de classes que constituirão um modelo, isto é, servirão como especificações básicas de novas classes, que serão implementadas por meio do mecanismo de herança. Assim, uma classe abstrata deve ser estendida, ou seja, deve ser a classe base de outra, mais específica, que contenha os detalhes que não puderam ser incluídos na superclasse (abstrata).

Classes Abstratas (2/3)

Outra possível aplicação das classes abstratas é a criação de um ancestral comum para um conjunto de classes que, se originados dessa classe abstrata, poderão ser tratados genericamente pelo polimorfismo. Para declaramos uma classe abstrata, basta adicionar o qualificador abstract em sua declaração, imediatamente após o especificador de acesso:

```
<access specifier> [abstract] class NomeDaClasse {
        // corpo da classe
}
```

Os métodos abstratos podem ser declarados como segue:

```
<access specifier> [abstract] nomeDoMétodo(listaDeParâmetros);
```

Note que esses métodos não têm uma implementação, isto é, não possuem um corpo delimitado por chaves contendo qualquer código.

Classes Abstratas (3/3)

O fato de uma classe ser declarada abstrata não significa que não possa conter métodos implementados, isto é, métodos acompanhados de um corpo que contém código.

Como geralmente as classes abstratas pertencem ao nível superior de uma hierarquia de classes, recomenda-se que contenham tanto código quanto for possível, deixando para suas subclasses apenas as implementações específicas dos métodos abstratos.

O qualificador abstract não pode ser utilizado em algumas circunstâncias:
- construtores não podem ser declarados como abstratos;
- métodos declarados como abstratos não podem ser privados (private);
- classes abstratas não podem conter métodos estáticos (static).

Outra restrição ao uso da denominação abstrata é que essas classes não podem ser instanciadas, isto é, não é possível a criação de objetos do seu tipo.

Interfaces (1/3)

O Java permite a definição de um tipo especial de classe, denominada interface.

Uma interface é uma construção peculiar, cujo uso permite obter resultados semelhantes aos obtidos com a herança múltipla, isto é, permite o compartilhamento das interfaces das classes envolvidas, sem o compartilhamento de suas implementações, levando ao extremo o conceito da orientação a objetos.

Isso significa que, embora o Java não ofereca a herança múltipla (quando uma classe compartilha a interface e a implementação de várias classes base simultaneamente), é possível obtermos a melhor parte da herança múltipla (ao compartilhamentos apenas a interface da classe), por meio desse novo mecanismo.

O nome interface tem origem exatamente do fato de que o que se pretende é uma definição compartilhável da interface propriamente dita de uma classe, mas não de sua implementação efetiva.

Interfaces (2/3)

Uma interface é como a definição de um contrato entre classes, isto é, a interface corresponde a uma especificação do que a classe deve oferecer e implementar em termos de métodos, o que resulta em uma poderosa abstração.

A construção das interfaces é semelhante à das classes abstratas, embora se diferenciem por dois aspectos: nenhum dos métodos declarados pode ser implementado; e, se existirem, todas das variáveis-membro devem ser estáticas e finais.

Uma interface pode ser declarada como é exemplificado a seguir:

```
interface NomeDaInterface {
        // declaração de atributos final static
        :
        // declaração de métodos (sem implementação)
}
```

Interfaces (3/3)

Para indicarmos que uma classe implementa uma interface, usamos a palavra reservada implements, como segue:

```
public class ClasseContratante implements InterfaceContratada {
        // corpo da classe
}
```

Também é possível criarmos uma interface estendendo outra, isto é, a criação de novas interfaces por meio da herança simples de outra interface, mas não de uma classe simples, e vice-versa, daí:

```
interface NomeDaInterface extends InterfaceBase {
        // declaração de atributos e assinatura de métodos
}
```

A denominação de interfaces deve seguir as mesmas regras que a denominação de classes. Todo o processamento de eventos das aplicações GUI baseia-se no uso intensivo de interfaces.

Vocabulário da POO, Parte III

- Abstração
- Representação
- Interface, Mensagem e Resposta
- Polimorfismo de sobrecarga e de sobreposição
- palavra-chave this
- coerção (cast) em objetos

Abstração (1/2)

- Qualquer mecanismo que nos permita representar a realidade complexa em termos de um modelo simplificado. [Yourdon]

- Consiste em enfocar nos aspectos essenciais de uma entidade e ignorar suas propriedades internas. Em desenvolvimento de software, isto significa enfocar no que um objeto é e faz, antes de decidir como ele dever ser implementado. [Rumbaugh]

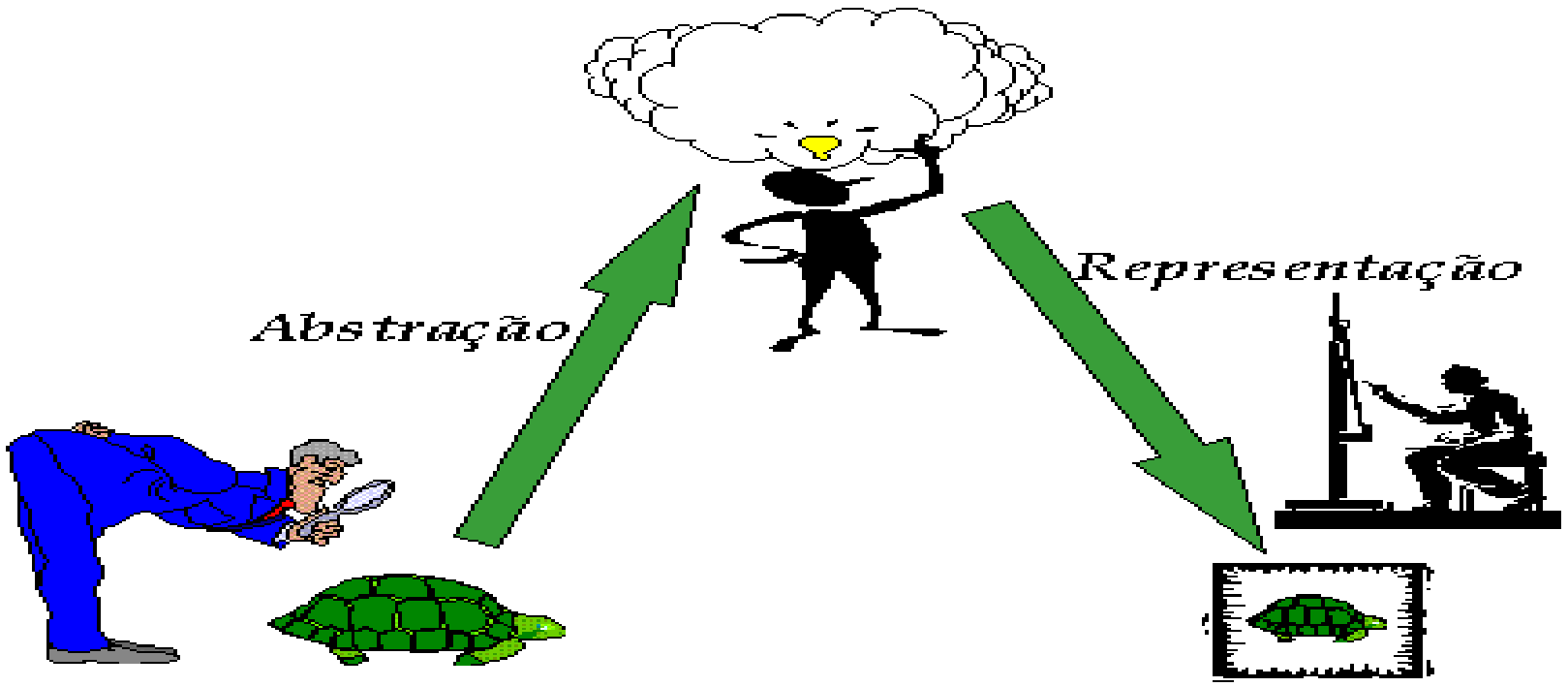
Abstração (2/2)

- O mecanismo básico utilizado para realização da análise do domínio da aplicação é a ***abstração***, através da qual um indivíduo observa a realidade (domínio) e procura capturar sua estrutura (***abstrair*** entidades, ações, relacionamento, etc, que forem consideradas relevantes para a descrição deste domínio).
- O resultado deste processo de abstração é conhecido como ***Modelo Conceitual***.

Representação

- Por um processo de ***representação*** este modelo conceitual pode, então, ser materializado segundo alguma convenção (um desenho, uma maquete, um texto, um diagrama, etc).
- No caso específico da informática, as representações mais convencionais são as linguagens de programação e as notações auxiliares na forma de diagramas e figuras.

Abstração e Representação



Abstração e Representaçao em Modelagem Conceitual
(adaptado de Takahashi e Liesenberg)

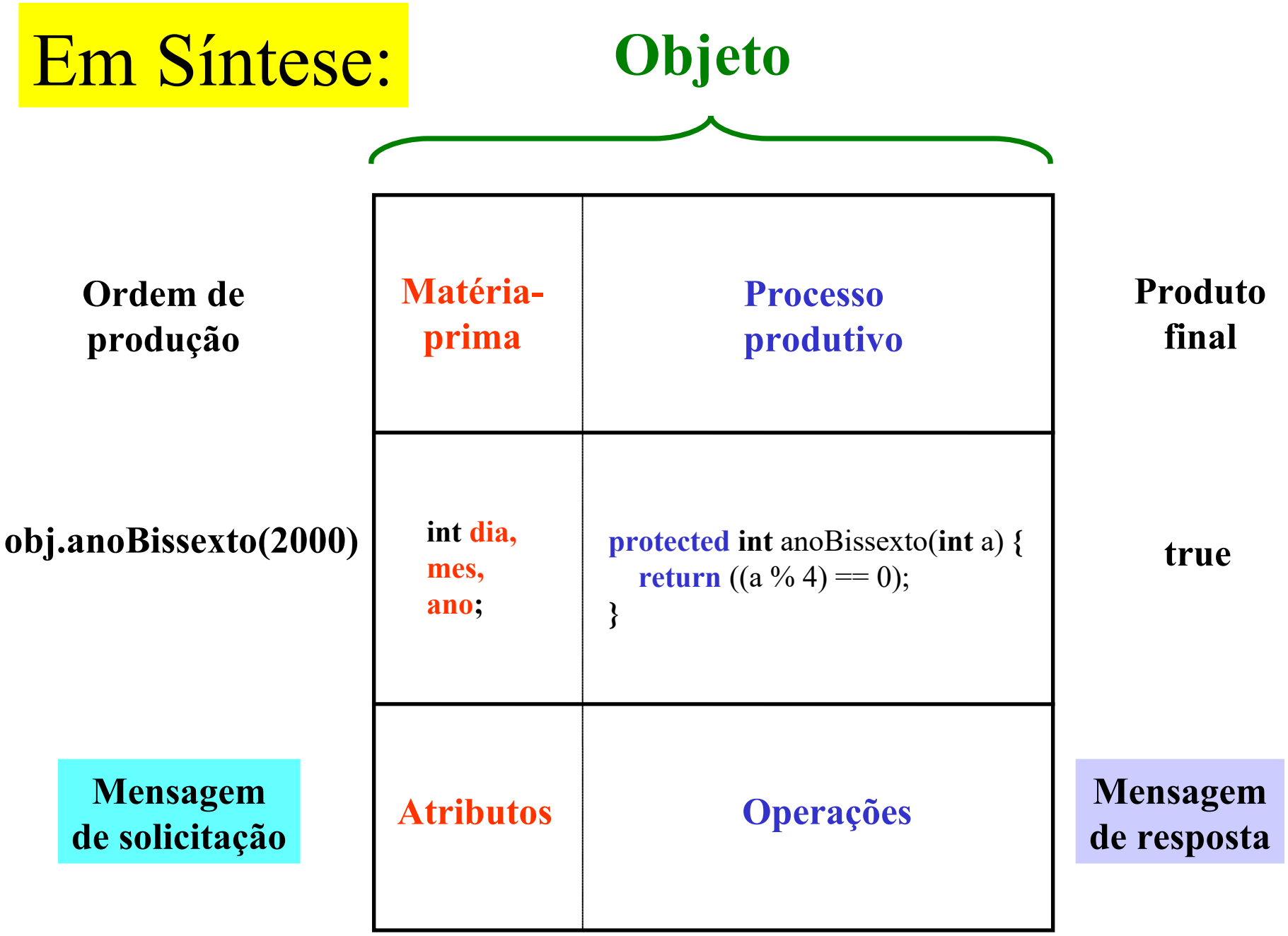
Interface, Mensagem e Resposta

- Interface, ou Assinatura
  - descreve as características externas do método, sua parte visível como: *Nome* e *ListaDeParâmetros*.
  - obs. o *Valor de retorno* não é considerado parte da assinatura de um método.
- Mensagem
  - nomeDoObjeto.nomeDoMétodo(<rel. de parâmetros>);
- Resposta
  - resultado da execução do método correspondente

Comunicação entre Objetos

- Objetos se comunicam através de mensagens, isto é, um sinal enviado de um objeto a outro requisitando um serviço através da execução de uma operação. Trata-se de um ciclo completo onde uma mensagem é enviada a um objeto, operações são executadas dentro com base nos dados de seu alcance na hierarquia de classes, e uma mensagem contendo o resultado da operação é enviada ao objeto solicitante.

Em Síntese:


Objeto
Ordem de produção
Matéria-prima
Processo produtivo
Produto final
obj.anoBissexto(2000)
int dia,
mes,
ano;
protected int anoBissexto(int a) {
return ((a % 4) == 0);
}
true
Mensagem de solicitação
Atributos
Operações
Mensagem de resposta

Tipos de Relações

- Objetos contribuem para o comportamento de um sistema, colaborando com outros.
  - Relações de Uso
  - Relações de Continência (conter)

- Alguns autores denominam de relações de *precedência* e *pai/filho*, respectivamente.

Relações de Uso

Papel dos objetos nas relação:

1. **Ator ou Objeto Ativo**: objeto que pode operar sobre outros, mas que não sofre ação de outros objetos;

2. **Servo ou Objeto Passivo**: um objeto que nunca opera sobre outro; ele sempre sofre ação de outros objetos;

3. **Agente**: um objeto que pode tanto atuar sobre um outro objeto ou sofrer ação de outros objetos

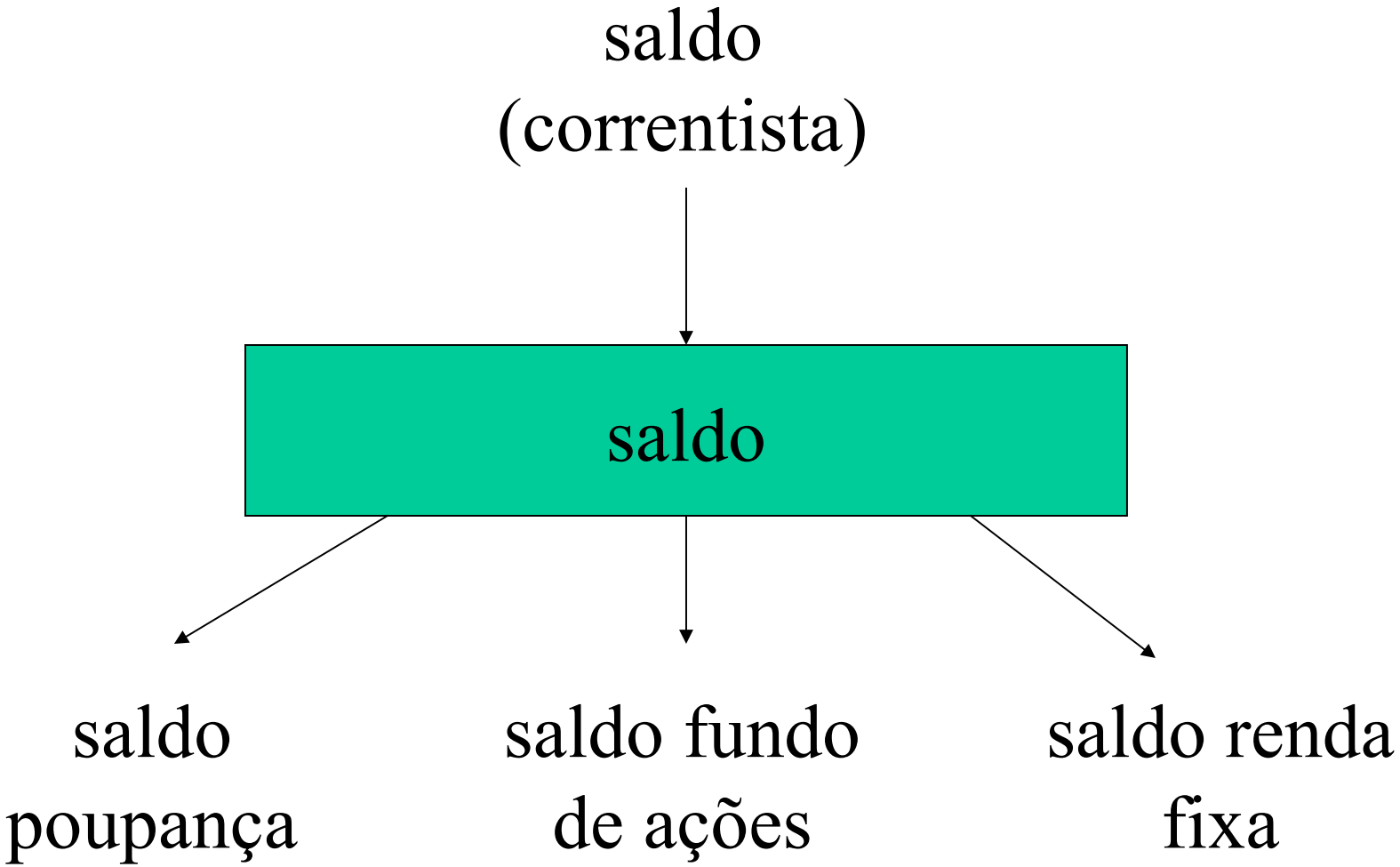
Polimorfismo (1/2)

- Linguagens convencionais, como Pascal e C são baseadas na idéia de que funções têm um tipo único: são linguagens *monomórficas*.
- Linguagens *polimórficas*, por sua vez, permitem que funções de mesmo nome, desde que pelo menos um de seus parâmetros perfaça uma distinção (quantidade, nome ou tipo), sejam invocadas para objetos diferentes.

Polimorfismo (2/2)

- Significa que a mesma operação pode se comportar diferentemente em classes diferentes. [Rumbaugh]
- O termo **polimorfismo** vem do grego *poli morfos* e significa **muitas formas**. Em programação orientada a objetos, este termo se refere a uma determinada classe que possui a capacidade de alterar o comportamento de um método para adequá-lo a necessidade solicitada.

Em Síntese: Polimorfismo



Tipos de Polimorfismo

1. ***Overloading***, ou sobrecarga de métodos:
   - Um método que possui diversos tratamentos diferentes de acordo com os parâmetros da chamada. Na prática temos dois ou mais métodos com o mesmo nome, mas aceitando parâmetros diferentes (ou, com assinaturas diferentes).
2. ***Overriding***, ou sobreposição (ou redefinição) de métodos:
   - Um objeto possui um método alterado, a partir de um método herdado de um objeto pai. Na prática, temos um método herdado na classe filho que é reescrito, geralmente com objetivo de especializar este objeto.

Sobrecarga de Métodos (1/3)

O polimorfismo é um dos aspectos mais importantes das linguagens orientadas a objetos. A sua forma mais simples, oferecida pela linguagem Java, é por meio da sobrecarga de métodos (*method overload*).

A sobrecarga de métodos é a possibilidade de existirem vários métodos com o mesmo nome em uma mesma classe.

Para que estes métodos homônimos possam ser distinguidos, eles devem ter uma assinatura diferente. A assinatura de um método é definida pelo nome do método é complementada pela lista que indica o nome e o tipo de todos os seus parâmetros formais; sendo assim, métodos com o mesmo nome são considerados diferentes quando recebem um diferente número ou diferentes tipos de argumentos, e têm, portanto assinaturas diferentes.

O valor de retorno não faz parte da assinatura, pois se admite a conversão automática de tipos, o que impede o compilador de identificar o método adequado.

Sobrecarga de métodos (um parâmetro – tipos diferentes)

```
public class Overload {
  public int dobro(int n) {
    return (2 * n);
  }
  public float dobro(float n) {
    return (2 * n);
  }
  public int dobro(String n) {
    return (2 * Integer.parseInt(n));
  }
}
```

Na classe **Overload** temos a implementação de três métodos denominados **dobro()**, que tomam um único argumento retornando um valor que é o dobro do valor do argumento recebido. Por meio da sobrecarga, foram implementadas três versões desse método na mesma classe, diferenciados pelo fato de que cada uma recebe um tipo diferente de argumento. O efeito produzido é a existência de um método **dobro()**, capaz de processar argumentos de tipos diferentes, simplificando o uso da classe **Overload**:

```
int i = 10; float f = 5.35f;
Overload meuObjeto = new Overload();
int j = meuObjeto.dobro(i);
System.out.println(i + " * 2 = " + j);
System.out.println(f + " * 2 = " + meuObjeto.dobro(f));
System.out.println("17 * 2 = " + meuObjeto.dobro("17"));
```

Sobrecarga de métodos (múltiplos parâmetro)

outra situação possível é a implementação de métodos sobrecarregados pelo uso diferentes listas de argumentos:

```
public class Overload {
  public int somatorio(int n) {
    int sm = 0;
    for (int i=1; i<=n; i++)
      sm += i;
    return (sm);
  }
  public int somatorio(int n, int incremento) {
    int sm = 0;
    for (int i=1; i<=n; i += incremento)
      sm += i;
    return (sm);
  }
}
```

Neste exemplo, temos a classe **Overload**, que tem dois métodos **somatorio()**: um efetua a soma dos primeiro **n** números naturais, e o outro efetua a soma dos primeiros **n** números naturais espaçados de **incremento**. A seleção de um desses métodos se dá pelo diferente número de argumentos utilizados.

```
Overload meuObjeto = new Overload();
System.out.println(meuObjeto.somatorio(10));
System.out.println(meuObjeto.somatorio(9, 2));
```

Sobrecarga de Métodos (2/3)

A API do Java utiliza intensivamente o mecanismo de sobrecarga; por exemplo, a classe **java.lang.String**, na qual o método **indexOf()** tem várias implementações. O método **indexOf()** se destina a localizar algo dentro da cadeia de caracteres contida nos objetos Strings, isto é, determinar a posição (índice) de um caracter ou substring. Veja as diferentes possibilidades para seu uso:

Método	Descrição_
indexOf(char ch)	retorna a posição, dentro da string, da primeira ocorrência do caracter (ch) especificado
indexOf(char ch, int n)	retorna a posição, dentro da string, da primeira ocorrência do caracter (ch) especificado a partir da posição (n) dada
indexOf(String s)	retorna a posição, dentro da string, da primeira ocorrência da substring (s) especificada
indexOf(String s, int n)	retorna a posição, dentro da string, da primeira ocorrência da substring (s) especificada a partir da posição (n) dada

Sobrecarga de Métodos (3/3)

É transparente para o usuário da classe a existência de métodos sobrecarregados, permitindo a escolha da alternativa mais adequada.

Ao utilizarmos uma classe na qual são implementados vários métodos sobrecarregados, fica a cargo do compilador verificar se existe um método apropriado em função da lista de argumentos indicada nas chamadas.

Durante a execução, essa checagem é realizada novamente para selecionar e acionar o método apropriado, o que é chamado de ligação dinâmica (*dynamic binding*) ou ligação tardia (*late binding*)

Ligação Tardia

Quando o método a ser invocado é definido durante a compilação do programa, o mecanismo de **ligação prematura** *(early binding)* é utilizado.

Para a utilização de polimorfismo, a linguagem de programação orientada a objetos deve suportar o conceito de **ligação tardia** *(late binding)*, onde a definição do método que será efetivamente invocado só ocorre durante a execução do programa.

O mecanismo de ligação tardia também é conhecido pelos termos *dynamic binding* (ligação dinâmica) ou *run-time binding* (ligação em tempo de execução).

Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia exceto em dois casos:

1. métodos declarados como final não podem ser redefinidos e portanto não são passíveis de invocação polimórfica da parte de seus descendentes; e
2. métodos declarados como private são implicitamente finais.

Sobrecarga de Construtores (1/3)

Da mesma forma que os métodos de uma classe podem ser sobrecarregados, uma classe pode ter mais de um construtor, e cada um deles é diferenciado pela assinatura.

O Java oferece para cada classe um construtor que não recebe parâmetros e que é denominado *default*. Esse construtor não precisa ser declarado e implementado quando apenas inicializar variáveis da classe.

Fazemos a sobrecarga de construtores (*construtor overload*) do mesmo modo que é feito em qualquer outro método, ou seja, criamos novos construtores, agora denominados construtores-cópia (*copy constructors*), cuja assinatura é diferente dos demais construtores.

Exemplo de Sobrecarga de construtores

A classe Sobrecarga tem dois atributos inteiros, **prim** e **sec**, que são inicializados com valor zero por meio do construtor *default*. Para que o atributo **prim** seja inicializado com outro valor, podemos utilizar o construtor-cópia, que recebe um parâmetro inteiro. Para que tanto o atributo **prim** como **sec** sejam inicializados diferentemente, dispõe-se ainda de um segundo construtor-cópia.

```
public class Sobrecarga {
// atributos
  public int prim;
  public int sec;
// construtores
  public Sobrecarga() {
    prim = sec = 0;
  }
  public Sobrecarga(int p) {
    prim = p; sec = 0;
  }
  public Sobrecarga(int p, int s) {
    prim = p; sec = s;
  }
}
```

Sobrecarga de Construtores (2/3)

O uso de construtores sobrecarregados é bastante conveniente, pois possibilita reduzir a quantidade de código que deve ser escrito para a utilização de um objeto, como exemplificado:

```
Sobrecarga s = new Sobrecarga();  // instanciação do objeto
s.prim = 10;                      // inicialização de prim
```

O trecho anterior pode ser substituído por:

```
Sobrecarga s = new Sobrecarga(10);
```

Outro caso possível:

```
Sobrecarga s = new Sobrecarga();  // instanciação do objeto
s.prim = 5; s.sec = -2;           // inicialização de prim e sec
```

Que pode ser substituído com vantagem por:

```
Sobrecarga s = new Sobrecarga(5, -2);
```

Sobrecarga de Construtores (3/3)

A API Java utiliza, extensivamente, o mecanismo de sobrecarga de construtores para oferecer aos programadores a maior flexibilidade possível.

Um exemplo disso é a classe **java.lang.Float**, que tem três diferentes construtores:

Construtor	Descrição
Float(**float** value)	cria um novo objeto **Float** contendo o valor *float* informado
Float(**double** value)	cria um novo objeto **Float** contendo o valor convertido do *double* informado
Float(**String** value)	cria um novo objeto **Float** contendo o valor convertido do *String* informado.

Exemplo da Ativação de Construtores em SubClasses

```
abstract class Base {
  abstract void m();
  public Base() {
    System.out.println("Base: início construção");
    m();
    System.out.println("Base: fim construção");
  }
}
class Derivada extends Base {
  int valor = 1;
  void m() {
    System.out.println("Derivada.m: " + valor);
  }
  public Derivada() {
    System.out.println("Derivada: início construção");
    System.out.println("Derivada.m: " + valor);
    System.out.println("Derivada: fim construção");
  }
  public static void main(String[] args) {
    Derivada meuObjeto = new Derivada();
  }
}
```

Resultado da Execução

```
Base: início construção
Derivada.m: 0
Base: fim construção
Derivada: início construção
Derivada.m: 1
Derivada: fim construção
```

Sobreposição de Métodos

Método herdado da classe pai, ou superclasse, que é reescrito, geralmente com objetivo de especializar o comportamento da classe filho, ou subclasse.

```
public class Superclasse {
   public String mensagem() {
      return ("Alô Mundo !");
   }
}
```

```
public class Subclasse extends SuperClasse {
   public String mensagem() {
      return ("Hello World !");
   }
}
```

super

Java não oferece o mecanismo de "herança múltipla", ou seja, não é possível criar uma classe derivada com mais de uma classe base. Por esse motivo, é simples fazer na classe derivada (ou subclasse) uma referência explicita para o método construtor da sua classe base (ou superclasse); o mecanismo para tal é o uso da palavra-chave super.

A chamada super(lista de argumentos); se presente, deve estar na primeira linha do construtor da classe base e pode conter qualquer número de argumentos apropriado para os vários construtores disponíveis na superclasse, ou classe pai.

Implicitamente, o compilador faz a invocação do construtor super() default (sem argumentos) para cada superclasse da hierarquia.

Outro uso dessa palavra-chave é como prefixo para referenciar métodos da superclasse "sobrepostos" na classe base.

```
super.nomeDoMétodo(lista de argumentos);
```

this

Quando um método é aplicado a um objeto, de algum modo deve ser sinalizado ao método a qual objeto a invocação está referindo-se. Por exemplo:

```
String s1 = "java";
String s2 = "linguagem de programação orientada a objetos";
int x1 = s1.length(); // tamanho da cadeia eh 4
int x2 = s2.length(); // tamanho da cadeia eh 44
```

Nesse exemplo, o método aplicado às duas strings é o mesmo, length(). Como o método sabe qual o objeto que ele deve "olhar" para saber o que fazer?

Por trás da resposta a essa pergunta está a presença de um parâmetro secreto, que é passado a todo método, que é uma referência ao objeto que está realizando a invocação. Em termos de uma linguagem procedimental, isso seria o equivalente a ter para as duas últimas linhas do exemplo acima:

```
int x1 = length(s1);
int x2 = length(s2);
```

Se necessário, o próprio método pode ter acesso a essa referência em seu código. Essa referência é passada através da palavra-chave this, reservada para esse propósito.

this - exemplos de uso

A palavra-chave this é utilizada principalmente em dois contextos:

1. Diferenciar atributos de objetos de parâmetros ou variáveis locais de mesmo nome;
2. Acessar o método construtor a partir de outros construtores.

```
class ThisSample {
  int x, y;

// exemplo do primeiro caso:
  public ThisSample(int x, int y) {
    this.x = x;
    this.y = y;
  }

// exemplo do segundo caso:
  public ThisSample( ) {
    this(1, 1);
  }
}
```

Coerção (cast) em objetos (1/2)

O uso de polimorfismo está intimamente relacionado ao mecanismo de *upcast*, onde parte da informação sobre um objeto torna-se inacessível, ou seja, informação é momentaneamente perdida. Esse processo é seguro do ponto de vista da orientação a objetos pois a interface da classe base nunca é maior que a interface da classe derivada.

Há situações onde é interessante recuperar a referência para o tipo original (ou classe pai) de um objeto, de modo a obter acesso à sua funcionalidade completa. Para tanto, o mecanismo de *downcast* precisa ser utilizado:

```
Tipo_orig novoObjeto = (Tipo_orig) Ref_atual;
```

O problema com *downcasting* é que é preciso verificar, através do operador instanceOf, se o objeto que está tendo sua referência convertida é realmente do tipo especificado, ou caso contrário seria impossível garantir sua manipulação correta após a conversão.

Em Java, todas as operações de *downcasting* são verificadas através do mecanismo de RTTI, *Run-Time Type Identification* suportado pela linguagem.

O operador instanceOf

Como é possível deslocar objetos utilizando referências às respectivas classes pai, em determinados momentos convém saber o que realmente esta acessível. Essa é a finalidade do operador instanceOf. Vamos supor que nossa hierarquia de classes seja ampliada da seguinte maneira:

```
public class Empregado; // implicitamente estende a classe raiz (Object)
public class Gerente extends Empregado;
public class Contato extends Empregado;
```

Se objeto enviado através de uma referência do tipo Empregado, pode ser que ele venha a ser ou não um Gerente ou um Contato:

```
public void metodo(Empregado emp) {
  if (emp instanceOf Gerente) {
     // tratá-lo como um objeto Gerente: (Gerente)emp }
  else if (emp instanceOf Contato) {
     // tratá-lo como um objeto Contato: (Contato)emp }
  else // tratá-lo como um objeto Empregado: emp
}
```

Coerção (cast) em objetos (2/2)

• o *casting* não transforma objetos, só os "reclassifica"

• os cast "para cima" na hierarquia de classes são sempre válidos e, na verdade, não exigem o operador de cast, podendo ser efetuadas através de uma simples atribuição: Empregado = Gerente; // funciona

• para cast "para baixo", o compilador precisa estar certo de que o cast seja, pelo menos, possível: Gerente = Contato; // inválido
A classe para a qual a coerção está sendo efetuada deve ser subclasse do tipo de referência atual, ou seja, Contato é subclasse da classe Empregado e não da classe Gerente.

• Se o compilador permitiu a coerção, o tipo de referência será verificado em tempo de execução. Se a verificação com o operador instanceOf for omitida e o objeto que está sofrendo a coerção não é de fato um objeto do tipo da classe que irá recebê-lo, ocorrerá uma "exceção" (*rum-time error*).

Vocabulário da POO, Parte IV

- Protocolo
  - conjunto de métodos implementados por uma classe
- Variáveis de instância
  - coleção de dados privados de uma classe
- Estado interno
  - conjunto de valores assumidos pelas variáveis de instância de um objeto

Recomendações para o Projeto de Classes (1/7)

1. Mantenha os atributos da classe com acesso privado. Dados privados seguem à risca o conceito de encapsulamento, pois são inacessíveis aos usuários da classe que, dessa forma, não podem atribuir valores inconsistentes aos atributos assim especificados. Tornam-se necessários métodos denominados de acesso, isto é, métodos que permitam ler o conteúdo desses atributos privados e métodos que permitam modificar o conteúdo desses atributos.

```
private double frequencia; // atributo
// método de escrita (mutator method)
public void setFrequencia(double f) {
   if (f >= 0)
      frequencia = f;
}

// método de leitura (acessor method)
public double getFrequencia() {
   return (frequencia);
}
```

Para um atributo denominado **atributo**, é prática comum de programadores Java denominar seu *acessor method* como **getAtributo()** e seu *mutator method* correspondente como **setAtributo()**.

Recomendações para o Projeto de Classes (2/7)

2. Nem todos os atributos necessitam de métodos de acesso. Mesmo sendo privados, nem todos os atributos necessitam de métodos de acesso para a leitura e escrita de dados. Avalie caso a caso e implemente apenas os métodos de acesso necessários.

3. Se os atributos não podem ser definidos como privados, defina-os como protegidos. Se for desejável que certos atributos sejam modificados em classes derivadas ou se for necessário utilizar diretamente tais atributos em classes derivadas, escolha o nível de acesso protegido (protected).

4. Parametrize os construtores. Se os atributos somente são modificados na criação do objeto, crie construtores parametrizados para realizar tal tarefa.

Recomendações para o Projeto de Classes (3/7)

5. Sempre inicialize atributos e variáveis. Embora o Java inicialize as variáveis-membro de uma classe, o mesmo não acontece com as variáveis locais (declaradas dentro de um método ou bloco de código). Assim, recomenda-se a inicialização explícita de todas as variáveis nos seus respectivos blocos, métodos ou construtores. Esse procedimento torna o código mais claro e não conta com facilidades especiais da máquina virtual Java.

6. Divida classes com muitas responsabilidades. Se uma classe tem muitos atributos ou métodos, estude sempre a possibilidade de dividir essa classe em duas ou três, tornando cada uma dessas classes mais simples e, ao mesmo tempo, mais especializada $\Leftrightarrow$ Dividir-para-conquistar (gereciamento da complexidade).

Recomendações para o Projeto de Classes (4/7)

7. Evite o uso de muitos tipos básicos em uma mesma classe. Quando vários atributos de tipos básicos declarados em um classe estão relacionados, deve ser analisada separadamente a possibilidade de transformar tais atributos em uma classe.

```
public class Pessoa {
  static String nome;
  static Data dataDeNascimento;
  public static void main(String args[]) {
    Pessoa objPessoa = new Pessoa();
    objPessoa.Nome = "Galbas Milleo Filho";
    objPessoa.dataDeNascimento.dia = 30;
    objPessoa.dataDeNascimento.mes = 11;
    objPessoa.dataDeNascimento.ano = 1965;
  }
}
static class Data {
  static int dia, mes, ano;
}
```

Recomendações para o Projeto de Classes (5/7)

8. Padronize a estrutura de suas classes. Recomenda-se que as classes sejam escritas da mesma forma, ou seja, que se utilize um estilo consistente de escrita de programas para todas as classes, incluindo-se o formato dos comentários e a distribuição destes dentro do código, bem como o posicionamento da declaração de atributos e de métodos.

A declaração de atributos pode ser colocada no início da classe.

Os métodos podem ser ordenados alfabeticamente ou, como é mais útil, divididos em categorias: primeiro, os públicos; depois, os protegidos; e, finalmente, os privados.

Recomendações para o Projeto de Classes (6/7)

9. Utilize nomes representativos na denominação de classes, métodos e atributos. As escolher um nome para uma classe, método ou atributo, procure um que signifique, o mais precisamente possível, o propósito desse elemento. Evite nomes incomuns, dúbios, sem significado ou abreviaturas não usuais.

As classes e os atributos devem utilizar substantivos como nomes simples ou substantivos e adjetivos como nomes duplos.

Os métodos devem dar preferência para verbos no infinitivo ou substantivos.

Recomendações para o Projeto de Classes (7/7)

10. Documente. Não economize comentários. Anote, mesmo que de forma simples e concisa, as razões que levaram àquela implementação em particular. Dê preferência aos comentários de documentação (/* ... **/), que podem ser extraídos automaticamente pelo programa javadoc.

11. Não reinvente a roda. Verifique se o problema já não tem uma solução. Procure utilizar soluções conhecidas e experimentadas. Indentifique a ocorrência de padrões dentro do problema e, quando possível, utilize padrões de projeto (*design patterns*) proposto.

Essas dicas não são as únicas que poderiam ser dadas nem tampouco definitivas, mas podem auxiliá-lo num bom projeto e na redação de suas classes.

Vantagens potenciais da OO

- Reusabilidade = reutilização de código através dos conceitos de herança (*Manutenibilidade*).
- Extensibilidade = reaproveitamento da implementação dos métodos da superclasse e adição de novos recursos ao sistema.
- A reusabilidade, extensibilidade e manutenibilidade são benefícios técnicos que conduzem as vantagens econômicas.

Link's Interessantes

- Universidade Federal de Santa Catarina
  - http://www.inf.ufsc.br/poo/conceitos/index.html
- MundoOO
  - http://www.mundooo.com.br/
- Cetus Links mais de 18.000 links relacionados a Tecnologia de Objetos
  - http://www.cetus-links.org/
- OMG – *Object Management Group*
  - http://www.omg.org/

Referências (1/2)

- Introdução ao Java.
  - Peter Jandl Junior.
  - São Paulo: Berkeley - 2002.
  - Capítulo 3: Java e a Orientação a Objetos, pág. 55..114.
- Ivan Luiz Marques Ricarte. POO: uma abordagem com Java.
  - http://www.dca.fee.unicamp.br/courses/PooJava/

Referências (2/2)

- Programando em Java 2-Teoria & Aplicações.
  - Rui Rossi dos Santos.
  - Rio de Janeiro: Axcel Books - 2004.
  - Capítulo 4: Programação Orientada a Objetos, pág. 143..241.