

Paradigmas de Linguagens de Programação

Prof. Sérgio D. Zorzo

Departamento de Computação - UFSCar

1º semestre / 2013

Aula 8

Material adaptado do Prof. Daniel Lucrédio

Programação Orientada a Objetos

Programação Orientada a Objetos

- Origens na Engenharia de Software
- Linguagens de máquina
 - Endereçamento absoluto
- Linguagens de montagem
 - Endereçamento relativo (nomes de variáveis)
- Linguagens estruturadas
 - Controle de fluxo explícito na sintaxe
 - Abstração de processo
- Crise do software!

Programação Orientada a Objetos

- Programas começam a ficar maiores
 - Difíceis de lidar/administrar
 - Milhares de milhares de linhas de código
- Abstração de processo não é mais suficiente
 - Necessidade de técnicas de modularização
- Conceito de encapsulamento

Encapsulamento

- Princípio de agrupar subprogramas e dados relacionados em um mesmo módulo
 - É um critério de modularização
 - Em outras palavras, em uma única unidade, devem estar:
 - Dados
 - Código que manipulam esses dados
- Conceito de tipos abstratos de dados
 - Abstração de dados
- Número ponto-flutuante
 - Representação interna
 - Operações de ponto-flutuante

Encapsulamento

- Ex:

```
public class Posicao
{
    public double latitude;
    public double longitude;
}
```

Encapsulamento

- **Ex:**

```
public class PosicaoUtil {  
    public static double distancia(Posicao p1, Posicao p2) {  
        // Calcula e retorna distância entre dois  
        // pontos geográficos  
    }  
    public static double direcao(Posicao p1, Posicao p2) {  
        // Calcula e retorna a direção entre  
        // dois pontos geográficos  
    }  
}
```

Encapsulamento

- **Ex:**

```
Posicao minhaCasa = new Posicao();  
minhaCasa.latitude = 36.538611;  
minhaCasa.longitude = -121.797500;
```

```
Posicao padaria = new Posicao();  
padaria.latitude = 36.539722;  
padaria.longitude = -121.907222;
```

```
double dist = PosicaoUtil.distancia(minhaCasa, padaria);  
double dir = PosicaoUtil.direcao(minhaCasa, padaria);
```


Encapsulamento

- O código anterior segue o estilo da década de 70
 - Estruturas de dados (registros)
 - Bibliotecas (conjunto de subprogramas)
 - Abstração de processo
- Além de dificultar a modularização
 - O programador precisa conhecer a ordem dos parâmetros
 - Ex: `direcao(p1, p2)`
 - de p1 a p2 ou de p2 a p1?

Encapsulamento

- Ex:

```
public class Posicao {  
    public double distancia(Posicao p) {  
        // Calcula e retorna a distância  
        // entre este objeto e p  
    }  
    public double direcao(Posicao p) {  
        // Calcula e retorna a distância  
        // entre este objeto e p  
    }  
    public double latitude;  
    public double longitude;  
}
```

Encapsulamento

- **Ex:**

```
Posicao minhaCasa = new Posicao();  
minhaCasa.latitude = 36.538611;  
minhaCasa.longitude = -121.797500;
```

```
Posicao padaria = new Posicao();  
padaria.latitude = 36.539722;  
padaria.longitude = -121.907222;
```

```
double dist = minhaCasa.distancia(padaria);  
double dir = minhaCasa.direcao(padaria);
```

Encapsulamento

- Agora, “Posicao” é um tipo abstrato de dados
 - Semântica mais clara (sentido da direção fica mais óbvio)
 - Modularização melhor
 - Dados e operações ficam na mesma unidade sintática
 - O programador pode aproveitar os benefícios do sistema de tipos
 - Ex: detecção de erros de tipos, conversões implícitas/explicitas

Encapsulamento

- Com encapsulamento, tem-se um critério a mais de modularização
 - Não basta separar um programa em subprogramas
 - Deve-se também identificar os dados manipulados nos subprogramas
 - Deixá-los explícitos
 - Agrupá-los em unidades, junto com os subprogramas associados

Encapsulamento

- David Parnas, em 1972, publicou um artigo:
 - “On the Criteria to Be Used in Decomposing Systems into Modules” - Communications of the ACM, Volume 15, Issue 12, Dec. 1972.
- Ele desenvolveu o princípio que ficou conhecido como ocultação da informação
 - Atualmente, fala-se em coesão e acoplamento
 - Princípios relacionados

Ocultação da informação

- Motivação:
 - Software tem partes cujo projeto (estrutura) são mais ou menos suscetíveis a mudanças, ao longo do ciclo de vida do software
 - Existem estruturas que tendem a ficar fixas
 - E existem estruturas que tendem a ser alteradas mais constantemente
- O princípio da ocultação da informação diz que:
 - As estruturas que tendem a ser alteradas mais constantemente devem ficar “escondidas” atrás das estruturas que tendem a ficar fixas
- Resultados:
 - Maior flexibilidade, compreensibilidade e menor tempo de desenvolvimento

Ocultação da informação

- Esse princípio obriga os projetistas/arquitetos a pensarem primeiro naquilo que é essencial/imutável em um sistema
 - Detalhes devem aparecer só depois, já que estes não irão interferir na estrutura maior
- Princípio básico
 - Não pertence a um paradigma em particular
- Mas a orientação a objetos “abraçou” a idéia

Programação Orientada a Objetos

- O que é?
 - Abstração de processo
 - Abstração de dados
 - Encapsulamento
 - Outros mecanismos (que veremos depois)
 - Aplicados com o objetivo de garantir o princípio da ocultação de informação
- OO se mostrou uma forma adequada e natural de se implementar o princípio da ocultação da informação

POO

- Voltando ao exemplo:

```
public class Posicao {  
    public double distancia(Posicao p) {  
        // Calcula e retorna a distância  
        // entre este objeto e p  
    }  
    public double direcao(Posicao p) {  
        // Calcula e retorna a distância  
        // entre este objeto e p  
    }  
    public double latitude;  
    public double longitude;  
}
```

POO

- E se eu precisar alterar o sistema de geometria?
 - Geometria plana
 - Geometria esférica
 - Geometria elíptica
- Atualmente, o programador (usuário) precisa conhecer o tipo de geometria utilizada
 - Detalhes de implementação
- Para fazer tal alteração, seria necessário alterar em diversos locais do programa

POO

- Se eu estivesse usando ocultação da informação corretamente

```
public class Posicao {  
    ...  
    public void setLatitude(double latitude) {  
        this.latitude = latitude;  
    }  
    public void setLongitude(double longitude) {  
        this.longitude = longitude;  
    }  
    private double latitude;  
    private double longitude;
```

POO

- Com informação oculta:

```
Posicao minhaCasa = new Posicao();  
minhaCasa.setLatitude(36.538611);  
minhaCasa.setLongitude(-121.797500);
```

```
Posicao padaria = new Posicao();  
padaria.setLatitude(36.539722);  
padaria.setLongitude(-121.907222);
```

```
double dist = minhaCasa.distancia(padaria);  
double dir = minhaCasa.direcao(padaria);
```

POO

- Se eu estivesse usando ocultação da informação corretamente

```
public class Posicao {  
    ...  
    public void setLatitude(double latitude) {  
        this.phi = Math.toRadians(latitude);  
    }  
    public void setLongitude(double longitude) {  
        this.theta = Math.toRadians(longitude);  
    }  
    private double phi;  
    private double theta;
```

Coesão e acoplamento

- Conceitos derivados da ocultação da informação
- Coesão
 - O grau com o qual uma classe tem uma responsabilidade única e bem definida
- Acoplamento
 - O grau com o qual uma classe depende de outras para exercer sua responsabilidade
- Ideal:
 - Alta coesão e baixo acoplamento

Paradigma Orientado a Objetos

- Iremos estudar como a orientação a objetos ajuda na ocultação da informação
 - Atributos
 - Métodos
 - Modificadores de acesso
 - Construtores
 - Herança
 - Interfaces
 - Polimorfismo
 - Pacotes
 - Exceções

JAVA

JAVA

- Iremos explorar o paradigma OO na linguagem Java
 - Mas antes, veremos o básico, para ajudar no entendimento e nos exemplos
- Não iremos entrar em detalhes da plataforma completa
 - Há um mundo Java lá fora
 - Java SE
 - Java EE
 - Java ME
 - Java FX
 - Java isso...
 - Java aquilo...

JAVA – o básico

- Algumas características da linguagem:
 - Case sensitive
 - camelCase é a convenção
 - Classes começam com letra maiúscula
 - Atributos, métodos e variáveis começam com letra minúscula
 - Constantes não utilizam camelCase, e sim todas maiúsculas separadas pelo “sublinhado”

JAVA – o básico

- Variáveis
 - Tipo primitivo
 - Tipo referência
 - Arrays
- Tipos primitivos
 - Inteiros (byte, short, int, long)
 - Reais (float, double)
 - Caractere (char)
 - Booleano (boolean)
 - E só

JAVA – o básico

- Tipos referência
 - Armazenam endereço de memória
 - Toda variável não-primitiva é uma referência
 - Podem “apontar” para:
 - null (significa que não tem nenhum endereço válido)
 - Um objeto
 - Um array
 - Ex: String
- Arrays
 - Podem ser de tipos primitivos ou referências
 - De qualquer forma, a variável será uma referência

JAVA – o básico

- Operadores
 - Unários
 - `!, ++, --, +, -, ~, ()`
 - Aritméticos
 - `+, -, *, /, %`
 - Deslocamento
 - `<<, >>, >>>`
 - Relacionais
 - `<, <=, >, >=, ==, !=, instanceof`
 - Lógicos
 - `&, |, &&, ||`
 - Bit a bit
 - `&, |, ^`

JAVA – o básico

- Operadores
 - Atribuição
 - =, +=, -=, *=, /=, %=
 - Ternário
 - ?:

JAVA – o básico

- Controle de fluxo

- Bidirecional

```
if(<expressao booleana>
    // comandoThen
else
    // comandoElse
```

- Seleção múltipla

```
switch(<variável>) {
    case <valor> : // comandos
    case <valor> : // comandos
    ...
    default: // comandos
}
```

- Obs: variável deve ser char, byte, short ou int
 - Obs2: uso de break é necessário para sair de um bloco “case”

JAVA – o básico

- Controle de fluxo

- Iteração

- while, do while

```
while(<expressao booleana>)  
    // comando
```

```
do
```

```
    // comando
```

```
while(<expressao booleana>)
```

- for

```
for(<inicialização>;<condição>;<incremento>)  
    // comando
```

- Controle explícito de saída

- break, continue

JAVA – o básico

- Arrays

- Declaração

- `<tipo> [] <nomeVariavel>;`

- `<tipo> <nomeVariavel>[];`

- Inicialização

- `<nomeVariavel> = new <tipo>[<tamanho>;`

- Uso

- Indexado a partir do zero

- Outras formas:

- `int[] a = {1,0,2,-1,99,-10}; // junto com declaração`

- `a = new int[] {1,2,3,4,5}; // em qualquer momento`

JAVA – o básico

- Arrays de arrays

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

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

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

```
int[][] a = {{10,20},{30,40},{50,60}};
```

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

```
a[0] = new int[100];
```

```
a[1] = new int[200];
```

```
a[2] = new int[300];
```

```
...
```

JAVA – o básico

- A classe String
 - Em JAVA, strings são referências, e não tipos primitivos
 - Ao fazer:
 - `String a = "Olá pessoal";`
 - Está sendo criado um novo objeto, que armazena os caracteres "O", "l", "á", " ", ...
 - A representação exata está escondida!
 - "a" é uma referência ("ponteiro") para este objeto
- Característica principal: uma String é imutável!
 - Uma vez inicializado, o objeto não é alterado!

JAVA – o básico

- **Ex:**

```
String a = "123";
```

```
a += "456";
```

- Nesse código, foram criados três objetos:
 - "123" // pelo compilador
 - "456" // pelo compilador
 - "123456" // durante execução
- Strings criadas pelo compilador ficam em um "pool" para reutilização

```
String a = "123";
```

```
String b = "123";
```

```
String c = "123";
```

- Nesse código, apenas um objeto é criado

JAVA – o básico

- Atribuição de Strings

```
String a = "123";
```

```
String b = a;
```

- a e b são duas referências para o mesmo objeto
 - Ou seja, são apelidos
- Porém, como uma String é imutável, não há problemas na criação dos apelidos

JAVA – o básico

- Comparação
 - Operador ==
 - Compara os endereços
 - Ou seja, só irá retornar true se forem o mesmo objeto
 - Ex: duas strings iguais criadas pelo compilador
 - Ex2: duas variáveis que são apelidos para a mesma string
 - Método equals
 - `str1.equals(str2)`
 - Compara caractere a caractere
 - Método equalsIgnoreCase
 - `str1.equals(str2)`
 - Compara ignorando a diferença entre maiúsculas e minúsculas
 - Método compare
 - `str1.compare(str2)`
 - Retorna um inteiro indicando a relação lexicográfica

JAVA – o básico

- Problema da imutabilidade das strings
 - Manipulação de strings sempre gera novos objetos
 - Concatenação / divisão
 - Se a manipulação for excessiva, pode ser um problema
- Nesses casos, deve-se utilizar StringBuffer
 - Objeto string “mutável”
 - Método “append” modifica a representação interna
 - Ex:
 - `StringBuffer sb1 = new StringBuffer();`
 - `sb1.append("abc");`
 - `sb1.append("def");`
 - Nesse exemplo, sb1 é um objeto único que foi modificado

JAVA – o básico

- **Ex:**

```
String s1 = "";  
s1 += "abc";  
s1 += "def";
```

- Nesse exemplo, foram criados 5 objetos (três pelo compilador e dois durante a execução)
 - s1 apontou para 3 objetos diferentes ao longo da execução

- **Ex:**

```
StringBuffer sb1 = new StringBuffer();  
sb1.append("abc");  
sb1.append("def");
```

- Nesse exemplo, foram criados 3 objetos (dois pelo compilador e um durante a execução)
 - sb1 aponta um objeto único que foi modificado ao longo da execução

OO em Java

Classes e objetos

- Uma classe implementa o princípio do encapsulamento
 - Unidade sintática que agrega dados e operações sobre os dados
 - Objetivo é modularizar e implementar a abstração de dados
- Importante: uma classe não necessariamente implica em ocultação da informação
 - Que é um princípio de projeto mais amplo

Classes e objetos

- Ou seja, uma classe tem:
 - Dados / atributos / campos / variáveis
 - Operações / métodos / procedimentos / subprogramas
- Okay:
 - Dado um problema qualquer...
 - Como saber separar as classes?
 - Como saber onde vai cada atributo?
 - Como saber onde vai cada método?
 - Qual é o melhor jeito de ocultar a informação?
 - Como saber quais partes do projeto mudam mais e quais mudam menos?

Classes e objetos

- A resposta verdadeira mesmo é:
 - É necessário experiência!
- Mas no paradigma OO, a resposta é:
 - Faça aquilo que for mais próximo do mundo real
- As estruturas de dados devem representar elementos da vida real
 - As características desses elementos devem ser os atributos
 - As computações sobre as características devem ser os métodos
 - ... pelo menos em tese!

Classes e objetos

- Esse é um bom ponto de partida
 - Na dúvida, o jeito mais natural deve ser adotado
 - Em alguns casos, é fácil identificar
 - Características e/ou comportamento óbvios
 - Ex:
 - Um produto em um sistema de comércio eletrônico
 - A posição em um sistema de localização
 - Um criptografador em um sistema de segurança
 - Em outros casos, a classificação exata pode não estar óbvia
 - Ex:
 - Pessoa e dependente

Classes e objetos

- Devemos também considerar o contexto
 - Dependendo do software que estamos projetando, mais ou menos detalhes são necessários
- Ex:
 - Sistema de estoque de uma oficina mecânica
 - Vale a pena detalhar que um carro possui motor
 - Um motor possui peças
 - Peças possuem código
 - Peças possuem subpeças
 - Sistema de vendas
 - Vale a pena detalhar o motor de um carro
 - Mas não é necessário detalhar as peças do motor

Classes e objetos

- NÃO entraremos em detalhes sobre COMO projetar um sistema
 - Tópico complexo e abrangente
- ESTUDAREMOS como as linguagens (no caso Java) oferece suporte à OO

Classes

- Em JAVA

```
<modificadores> class <nome> {  
    ... atributos e métodos ... // em qualquer ordem  
}
```

- Normalmente:
 - Uma classe pública em cada arquivo
 - Nome do arquivo idêntico ao nome da classe + “.java”
- Ex: arquivo Pessoa.java

```
public class Pessoa {  
  
}
```

Classes

- Atributos
 - “Variáveis” definidas dentro da classe
- Ex:

```
class Pessoa {  
    long rg;  
    String nome, sobrenome;  
    Date dataNascimento;  
    String[] enderecos;  
}
```

Classes e objetos

- Se uma classe define uma categoria de elementos do mundo real
 - Um objeto é uma ocorrência em particular de uma classe
 - Ex:
 - classe Produto / objeto Lavadora Electrolux
 - classe Pessoa / objeto João Paulo
 - classe Posição / objeto Coordenadas DC – UFSCar

Classes e objetos

- Um objeto instancia uma classe
 - Significa que ele possui valores para todos os atributos daquela classe

- Ex:

```
class Qualquer {  
    int a, b;  
    float[] c;  
    boolean d;  
    char e;  
    String f;  
}  
  
Qualquer q = new Qualquer();  
// q possui valores para a, b  
// c, d, e f  
// (Ainda que valores implícitos)
```

Classes e objetos

- Regras de inicialização de atributos
 - byte, short, int, long \rightarrow 0
 - float, double \rightarrow 0.0
 - char \rightarrow `'\u0000'`
 - boolean \rightarrow false
 - referências \rightarrow null
- Ou seja, se não for feita uma inicialização explícita, esses serão os valores dos atributos em uma instância

Classes e objetos

- Acessando atributos de um objeto
- Operador ponto “.”
 - Desde que haja visibilidade (veremos mais adiante)
- Ex:

```
class Qualquer {  
    int a, b;  
    float[] c;  
    boolean d;  
    char e;  
    String f;  
}
```

```
Qualquer q = new Qualquer();  
System.out.println(q.a);  
if(!q.d)  
    q.a = 20;  
else q.a = 10;  
q.b = q.a + 5;
```

Classes e objetos

- Um objeto possui um estado
 - Valores dos atributos em um determinado momento
- Ex:

```
Qualquer q = new Qualquer();
```

```
System.out.println(q.a);
```

```
if(!q.f)
```

```
    q.a = 20;
```

```
else q.a = 10;
```

```
q.b = q.a + 5;
```

Estado
1

Estado
2

Estado
3

Estado
4

Classes

- Métodos
 - Subprogramas definidos dentro da classe
 - O código dos métodos pode acessar normalmente os atributos, como se fossem variáveis locais

- Ex:

```
class Pessoa {  
    String nome, sobrenome;  
    void imprimeNomeCompleto() {  
        System.out.println(nome+" "+sobrenome);  
    }  
    void imprimeNomeBibliográfico(PrintStream p) {  
        p.println(sobrenome.toUpperCase()+  
            ", "+nome.toUpperCase());  
    }  
}
```


Classes

- Retorno de métodos
 - Métodos com tipo de retorno diferente de void DEVEM especificar o que é retornado
 - Em TODOS os caminhos do subprograma

- Ex:

```
class Pessoa {  
    String nome, sobrenome;  
    String montarNomeCompleto() {  
        if(nome != null && sobrenome != null) {  
            return nome + " " + sobrenome;  
        }  
        // erro: método deve retornar uma String  
        aqui
```

Classes

- Chamadas de métodos
 - Semelhante ao acesso a atributos
 - Operador ponto “.”
- Ex:

```
class Pessoa {  
    String nome, sobrenome;  
    void imprimeNomeCompleto() {  
        System.out.println(nome+" "+sobrenome);  
    }  
    void imprimeNomeBibliográfico(PrintStream p) {  
        p.println(sobrenome.toUpperCase()+  
            ", "+nome.toUpperCase());  
    }  
}
```

```
Pessoa p = new Pessoa();  
p.nome = "Daniel";  
p.sobrenome = "Lucrédio";  
p.imprimeNomeCompleto();
```

Classes

- Passagem de parâmetros
 - Em JAVA, a passagem é por valor (cópia na chamada)
 - Para tipos primitivos, modificar o parâmetro formal não altera o parâmetro real
 - Para tipos referência, modificar o parâmetro formal com atribuição não altera o parâmetro real
 - Mas modificar o parâmetro formal por meio de métodos/atributos pode alterar o estado do parâmetro real

Classes

- Passagem de parâmetros

- Ex:

```
void soma(int a, int b, int c) {  
    c = a + b;  
}
```

...

```
int x = 0;
```

```
soma(2, 2, x);
```

```
System.out.println(x); // imprime 0
```

Classes

- Passagem de parâmetros
- Ex:

```
class Inteiro {  
    int valor;  
    void mudaValor(int valor) {  
        this.valor = valor;  
    }  
}  
  
...  
  
void soma(Inteiro a, Inteiro b, Inteiro c) {  
    c.mudaValor(a.valor + b.valor);  
}
```

Classes

- Passagem de parâmetros
 - Tipos String são referências
 - Mas são imutáveis, portanto se comportam como primitivos
 - Para alterar Strings em um método, deve-se utilizar a classe StringBuffer

Encapsulamento de atributos

Ocultação da informação

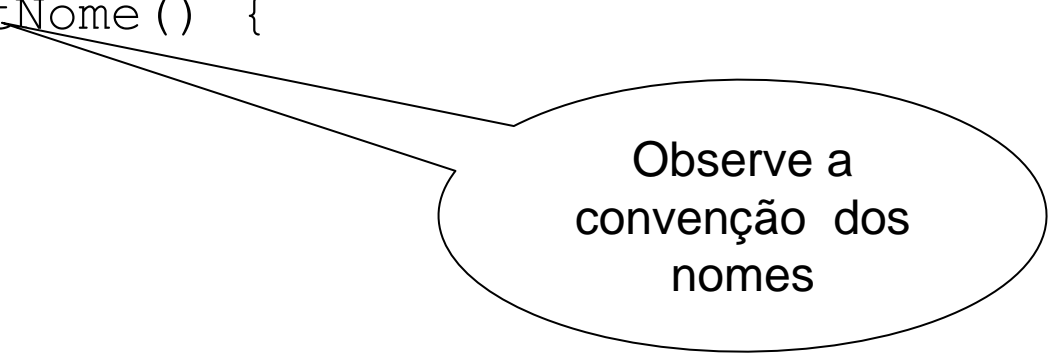
- Como já discutido, ocultar a informação é um princípio de projeto
 - Envolve uma análise criteriosa sobre quais partes de um projeto se alteram mais ou menos frequentemente
- Em Java, esse princípio levou a uma convenção
 - Na verdade, é discutível a sua real necessidade
 - Mas tornou-se um padrão de fato
- Essa convenção é chamada de encapsulamento de atributos (erroneamente)
 - Padrão de projeto importante também por outros motivos
 - Padronização / Interoperabilidade / Legibilidade

Getters e setters

- Regra:
 - Para cada atributo, cria-se um método para leitura (getter) e um método para escrita (setter)

- Ex:

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



Observe a
convenção dos
nomes

Getters e setters

- Para tipos booleanos, pode-se utilizar “is” ao invés de “get”
- Ex:

```
boolean aposentado;
```

```
void setAposentado(boolean aposentado) { ... }
```

```
boolean isAposentado() { ... }
```

Getters e setters

- Para completar o encapsulamento, deve-se adicionar o modificador “private” aos atributos
 - De forma que o acesso a eles é feito SOMENTE através dos getters/setters
 - O compilador proíbe o acesso direto

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

Getters e setters

- Garantem uma forma uniforme de acesso aos atributos
 - O programador da classe tem como garantir regras de acesso aos dados
- Ex:
 - latitude somente aceita valores entre -90 e 90

```
void setLatitude(double lat) {  
    if(lat < -90) lat = -90;  
    if(lat > 90) lat = 90;  
    this.lat = lat;  
}
```

Getters e setters

- A representação real dos atributos pode ser modificada sem impacto
 - Causa menor acoplamento

- **Ex:**

```
char valor = 'v';  
void setFlag(boolean flag) {  
    if(flag)  
        valor = 'v';  
    else valor = 'f';  
}  
boolean getFlag() {  
    return valor == 'v';  
}
```

Getters e setters

- Importante
 - O “encapsulamento” de atributos é apenas UMA forma de ocultar informação
 - O princípio deve ser aplicado em todo o projeto

Construtores

Construtores

- Um construtor é um método especial
 - Sem retorno
 - Nome idêntico ao da classe
 - Pode ter parâmetros ou não
- Chamado na criação de uma instância da classe

```
class Pessoa {  
    String nome;  
    Pessoa() {  
        nome = "Fulano";  
    }  
}
```

```
}
```

```
...
```

```
Pessoa p = new Pessoa();
```

```
System.out.println(p.nome); // imprime Fulano
```


Construtores

- Construtor padrão (default)
 - Construtor sem parâmetros
- Quando nenhum construtor é declarado pelo programador
 - O compilador insere o construtor padrão automaticamente
 - Sem nenhum comportamento definido
- Garante que toda classe tem pelo menos um construtor

Construtores

- Servem para que o programador possa GARANTIR que um ou mais atributos sejam inicializados
- Ex:

```
class Pessoa {  
    String nome, sobrenome;  
    public String getNomeCompleto() {  
        if(nome == null || sobrenome == null)  
            throw new RuntimeException("Precisa  
inicializar nome e sobrenome antes de  
chamar este método!");  
        return nome + " " + sobrenome;  
    }  
}
```

Construtores

- Ex:

```
class Pessoa {  
    String nome, sobrenome;  
    Pessoa(String nome, String sobrenome) {  
        this.nome = nome;  
        this.sobrenome = sobrenome;  
    }  
    public String getNomeCompleto() {  
        return nome + " " + sobrenome;  
    }  
}  
  
...  
  
Pessoa p = new Pessoa(); //erro!
```

Sobrecarga de construtores

- Quando há múltiplas formas de inicializar um objeto, todas válidas

```
class Pessoa {  
    String nome, sobrenome;  
    Pessoa(String nome) {  
        this.nome = nome;  
        this.sobrenome = "de tal";  
    }  
    Pessoa(String nome, String sobrenome) {  
        this.nome = nome;  
        this.sobrenome = sobrenome;  
    }  
    public String getNomeCompleto() {  
        return nome + " " + sobrenome;  
    }  
}  
  
...  
Pessoa p = new Pessoa("Fulano");  
System.out.println(p.getNomeCompleto()); // Imprime Fulano de tal
```

Sobrecarga de construtores

- Referência “this”
- Usado para fazer referência ao objeto atual
- Ex:

```
class Qualquer {  
    int a = 2;  
    void metodo() {  
        int a = 3;  
        System.out.println(a);  
        System.out.println(this.a);  
    }  
}  
  
...  
Qualquer q = new Qualquer();  
q.metodo(); // imprime 3 e depois 2
```

Sobrecarga de construtores

- `this()` pode ser utilizado para facilitar a sobrecarga de construtores
 - Evitar duplicação de código

```
class Pessoa {  
    String nome, sobrenome;  
    Pessoa(String nome) {  
        if(nome.length > 6) nome = nome.substring(0,6);  
        this.nome = nome;  
        this.sobrenome = "de tal";  
    }  
    Pessoa(String nome, String sobrenome) {  
        if(nome.length > 6) nome = nome.substring(0,6);  
        this.nome = nome;  
        if(sobrenome.length > 6) sobrenome =  
sobrenome.substring(0,6);  
        this.sobrenome = sobrenome;  
    }  
}
```

Sobrecarga de construtores

```
class Pessoa {  
    String nome, sobrenome;  
    Pessoa(String nome) {  
        this(nome, "de tal");  
    }  
    Pessoa(String nome, String sobrenome) {  
        if(nome.length > 6) nome = nome.substring(0,6);  
        this.nome = nome;  
        if(sobrenome.length > 6) sobrenome =  
sobrenome.substring(0,6);  
        this.sobrenome = sobrenome;  
    }  
    ...  
}
```

Sobrecarga de construtores

- `this()` só pode ser usado como a primeira instrução
 - Caso haja outra instrução antes, o compilador irá acusar um erro
- Regra geral:
 - Um construtor geral, completo, que inicializa todos os atributos
 - Pode ser `public` ou `private`, dependendo da necessidade
 - Outros construtores mais restritos, que definem um valor padrão para atributos omitidos na inicialização
 - Obs: apenas uma sugestão
 - Siga seus instintos!

Modificador static

Static

- Aprendemos antes que a palavra “estático” se relaciona a “tempo de compilação”
 - Ou seja, o compilador consegue alocar variáveis estáticas
 - Ou melhor, definir uma área de memória para ela em tempo de compilação
- Em Java, temos o modificador static
 - O conceito é parecido, mas não exatamente implementado dessa forma
- Pode ser aplicado a atributos, métodos e blocos de código

Atributos static

- Associados à classe, e não a uma instância em particular
 - Também chamados de atributos (ou variáveis) de classe
 - Em contraste com atributos de instância (não-estáticos)
 - Acesso via nome da classe (e não objeto)
 - Apesar de ser possível
- Exemplo clássico: contador de objetos

```
class Qualquer {  
    static int contador;  
    public Qualquer() {  
        contador++;  
    }  
}
```

...

```
System.out.println(Qualquer.contador);
```

Atributos static

- Normalmente, são utilizados para constantes
 - Ou seja, independem de uma instância
 - Combinados com o modificador final, é impossível alterar seu valor
- Ex:

```
class Buffer {  
    public final static byte TAMANHO = 256;  
    int[] buffer;  
    public Buffer() {  
        buffer = new int[TAMANHO];  
    }  
}
```

Atributos static

- Também é utilizado para objetos do tipo Singleton (padrão de projeto)
- Ex:

```
class ConexaoBD {  
    static ConexaoBD c = new ConexaoBD();  
}
```

Métodos static

- Métodos de classe (em contraste com métodos de instância)
 - Não associados a uma instância em particular
- Naturalmente, métodos static só podem acessar atributos static
 - Compilador previne o contrário
- Exemplo mais comum:

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

Métodos static

- Usados para encapsulamento de atributos static
 - Ocultação da informação
- Exemplo: singleton anterior não prevenia a criação de instâncias adicionais

```
class ConexaoBD {  
    static ConexaoBD c = new ConexaoBD();  
}
```

- Nesse caso, algum outro objeto poderia criar outra instância e atribuir a c explicitamente

Métodos static

- **Ex:**

```
class ConexaoBD {  
    private static ConexaoBD c;  
    public static ConexaoBD getConexaoBD() {  
        if(c == null) c = new ConexaoBD();  
        return c;  
    }  
}
```


Blocos de código static

- Blocos de código utilizados para inicialização de atributos estáticos
- Ex:

```
class Qualquer {  
    private static int[] valores;  
    static {  
        valores = new int[100];  
        for(int i=0;i<100;i++) {  
            valores[i] = i+1000;  
        }  
    }  
    ...  
}
```

Relacionamento entre classes

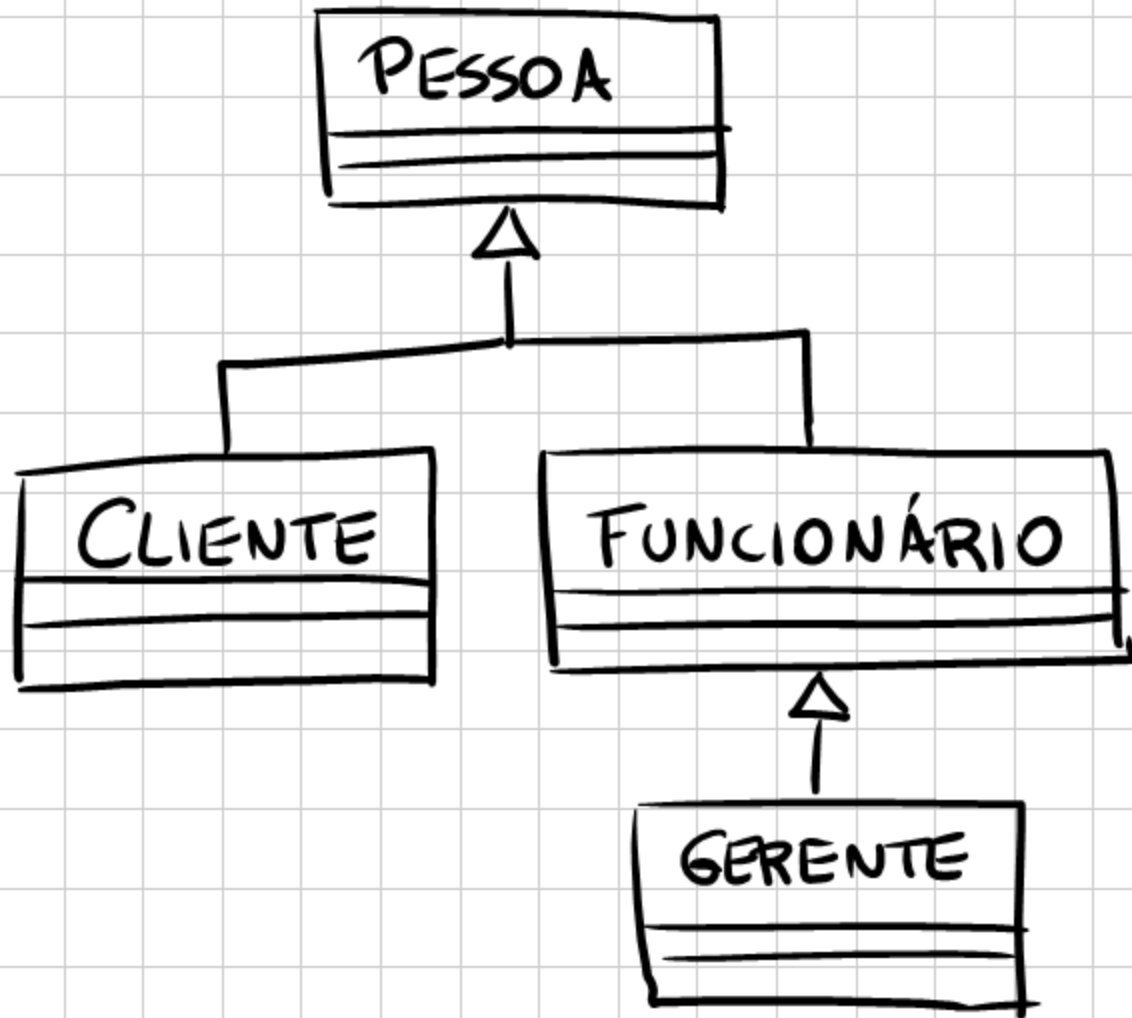
Relacionamento entre classes

- Este é um aspecto onde Java não implementa OO em sua plenitude
- Classes podem se relacionar entre si de diversas formas
 - Herança
 - Composição
 - Agregação
 - Associação
 - Dependência
- São conceitos do desenvolvimento de software orientado a objetos
 - Isto é: análise, projeto e implementação

Herança

- É a relação mais forte
 - Semântica: “é um”
 - Exs:
 - Cliente “é uma” Pessoa
 - Funcionário “é uma” Pessoa
 - Gerente “é uma” Pessoa
 - Gerente “é um” Funcionário

Herança

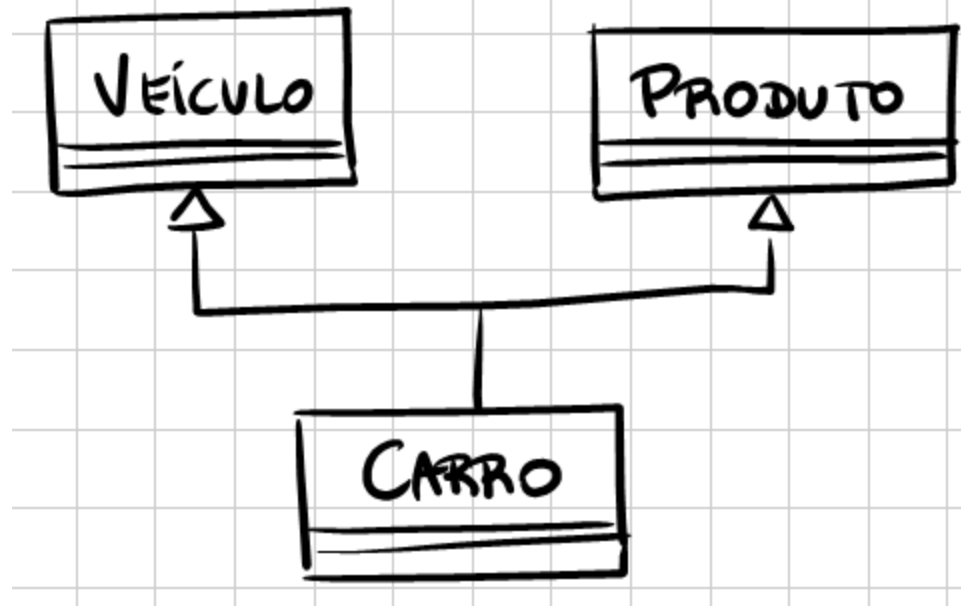


Herança

- Significado
 - Características e comportamento são “herdados” pelos filhos
 - Atributos e métodos
 - Ou seja
 - Se Pessoa tem nome, Cliente tem nome (assim como Funcionário e Gerente)
 - Se Pessoa tem um método cadastrar, Cliente tem um método cadastrar (assim como Funcionário e Gerente)

Herança múltipla

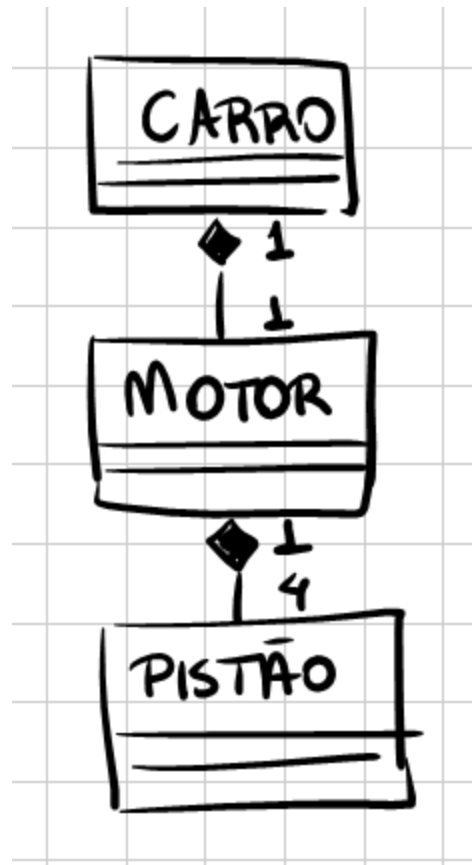
- Ex:
 - Em uma loja de veículos
 - Carro é, ao mesmo tempo, um veículo e um produto



Composição

- Semântica: “composto de”
 - Relação todo-parte, onde a parte não existe sem o todo
- Ex: oficina mecânica
 - Motor é parte de um carro
 - Pistão é parte de um motor
 - Não existe motor sem carro
 - Não existe pistão sem motor

Composição



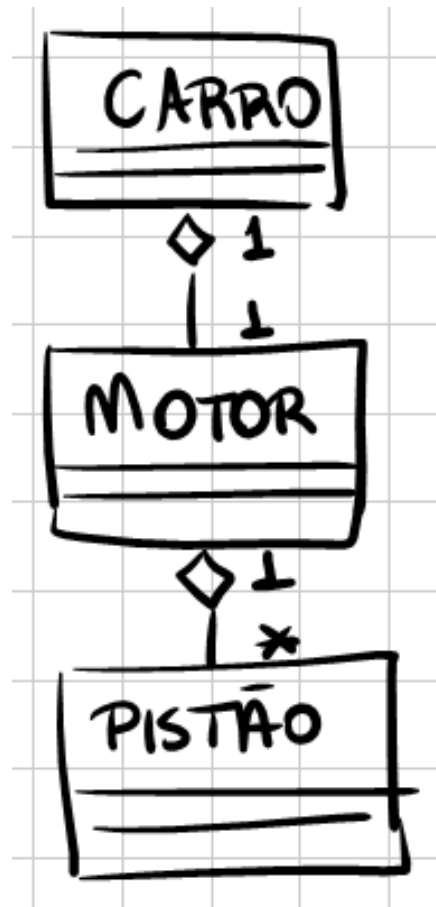
Composição

- Outro exemplo:
 - Uma universidade é composta de centros
 - Um centro é composto de departamentos
 - Não faz sentido um departamento sem um centro
 - Não faz sentido um centro sem uma universidade
- Outro exemplo ainda:
 - Um pedido é composto de itens de pedido
 - Não faz sentido um item de pedido sem um pedido associado

Agregação

- Semântica: “tem”
 - Relação todo-parte, onde a parte pode existir sem o todo
- Ex: revendedora de peças
 - Motor é parte de um carro
 - Pistão é parte de um motor
 - Mas eu posso vender motores e pistões separadamente

Agregação



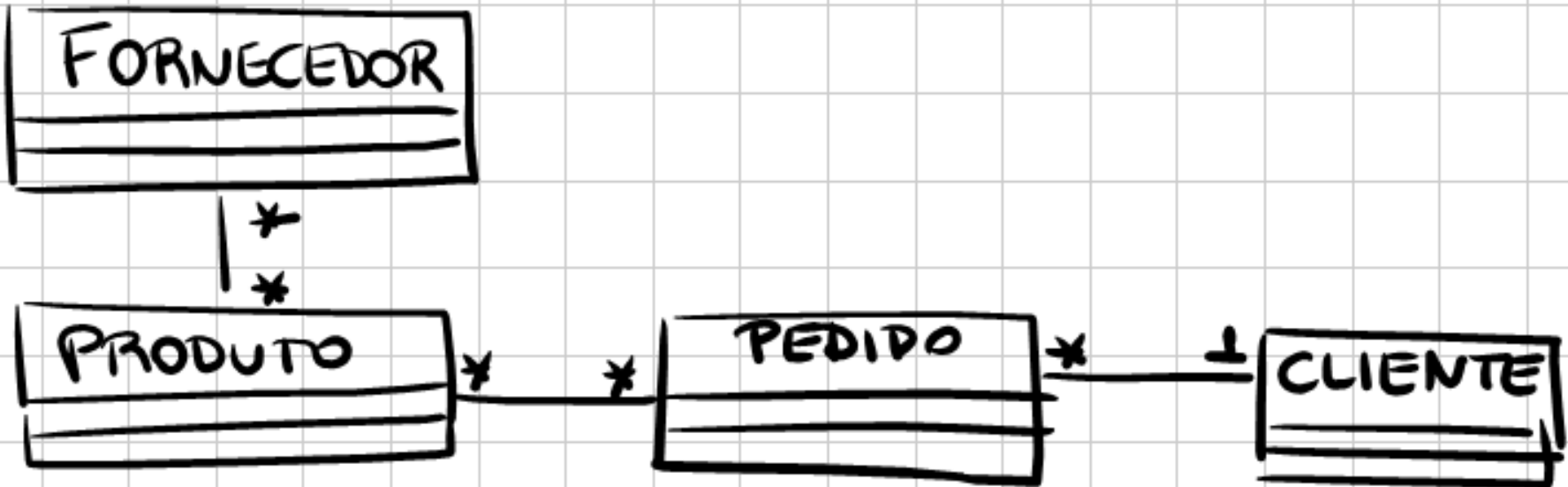
Agregação

- Outro exemplo:
 - Uma turma tem (é composta de) alunos
 - Mas o aluno pode participar de outras agregações
 - Ex: uma disciplina, uma agremiação, uma comissão, etc.

Associação

- Relação mais fraca entre classes
 - Semântica: “se relaciona a”
 - Relação entre classes que não é herança, nem composição e nem agregação
- Exemplo:
 - Cliente faz pedido
 - Fornecedor vende produto

Associação



Dependência

- Relação indireta
 - Semântica: “depende de”
 - Sentido mais funcional, ou seja, uma classe depende de outra para funcionar
 - Nível mais baixo de abstração
- Exemplo:
 - Classe CadastroCliente depende de classe ConexaoBD
 - Classe FolhaDePagamento depende da classe Math

Dependência



Relacionamentos entre classes

- JAVA
 - Somente herança simples
 - Sem distinção entre composição / agregação / associação
 - Dependências implícitas
- Essas limitações não são graves
 - É possível facilmente contorná-las sem muita perda na legibilidade / capacidade de escrita
- Veremos agora como JAVA implementa esses relacionamentos
 - Usaremos o caminho inverso, ou seja, dos relacionamentos mais fracos para os mais fortes

Dependência

- Caracterizada pelo uso de uma classe dentro de outra
 - Seja pela chamada de um método estático, ou acesso a algum atributo
- Ex:

```
class CadastroCliente {  
    public void cadastrar() {  
        ConexaoBD conexao = ConexaoBD.getConexaoBD();  
        conexao.executarComandoSQL("INSERT ...");  
        ...  
    }  
}
```

Dependência

- Problema:
 - Não fica facilmente visível na ESTRUTURA da classe que ela depende de outra
 - É necessário observar o código atentamente
 - Por exemplo, se uma classe depende de outra que está em uma unidade de compilação diferente (outro arquivo), é necessário incluir tal arquivo no projeto
- Solução 1:
 - Se as classes estiverem em pacotes diferentes (veremos mais adiante)
 - É possível deixar a declaração de uso em evidência
- Solução 2:
 - Deixar o compilador/JVM detectar erros
 - Ao tentar compilar/executar uma classe que depende de outra, mas cujo arquivo está ausente, o compilador/JVM irá exibir uma mensagem

Dependência

- Solução 3:
 - Usar comentários (texto normal) explicitando a dependência
- Solução 4:
 - Utilizar ferramentas de análise de código
- Solução 5:
 - Remover as dependências do código!
 - E inseri-las posteriormente
 - Utilizando um artefato que deixa as dependências explícitas
 - Padrão de projeto conhecido como injeção de dependência
 - Frameworks como Spring e Pico implementam este padrão

Associação / agregação

- Uma classe que possui um atributo de outra classe
 - No nível de programação, não há diferença entre associação e agregação
 - A distinção é apenas semântica
- Ex:

```
class Produto {  
    private Fornecedor fornecedor;  
    public Fornecedor getFornecedor() {  
        return fornecedor;  
    }  
    public void setFornecedor(Fornecedor fornecedor) {  
        this.fornecedor = fornecedor;  
    }  
}
```

Associação / agregação

- **Ex:**

```
class Pedido {  
    private Produto[] produtos;  
    public Pedido(int qtdeItens) {  
        produtos = new Produto[qtdeItens];  
    }  
    public Produto[] getProdutos() {...}  
    public void addProduto(Produto p) {  
        produtos[tamanho++] = p;  
    }  
    ...  
}
```

Associação / agregação

- A linguagem não garante, por exemplo, que uma determinada multiplicidade seja garantida
 - Ex: um motor pode ter no máximo 4 pistões
 - É necessário programar manualmente
 - Métodos setters / getters são o melhor local para esse tipo de restrição

Composição

- Em termos de código, é idêntico à associação
 - Ou seja, uma classe que tem um atributo que é do tipo de outra classe

- Ex:

```
class Pedido {  
    ItemPedido[] itens;  
}  
  
class ItemPedido {  
    ...  
}
```

- Nesse caso, não há garantia de que todo ItemPedido terá um pedido associado

Composição

- Uma solução é adicionar um construtor que exige, na criação de ItemPedido, que seja informado o Pedido correspondente
 - Não é a melhor solução, mas pode ser útil em alguns casos

- Ex:

```
class ItemPedido {  
    ItemPedido(Pedido p) {  
        // verifica se p existe mesmo  
        // caso contrário, acusa um erro  
    }  
}
```

Composição

- Outra solução é encapsular a criação de ItemPedido na classe Pedido

```
public class Pedido {  
    public class ItemPedido {  
        Pedido p;  
        private ItemPedido(Pedido p) {  
            this.p = p;  
        }  
    }  
  
    public ItemPedido criarItemPedido() {  
        ItemPedido ip = new ItemPedido(this);  
        return ip;  
    }  
}  
  
...  
Pedido p = new Pedido();  
Pedido.ItemPedido ip = p.new ItemPedido(); // erro  
Pedido.ItemPedido ip2 = p.criarItemPedido();
```

Herança

Herança

- Uma das características “marcantes” da orientação a objeto
- Uma das formas mais simples de reutilizar software
- Possibilita um aumento na legibilidade e capacidade de escrita
 - Acarreta em menor desempenho na teoria, mas atualmente isso não é mais um problema real
- Relaciona uma classe pai (superclasse) com uma classe filha (subclasse)

Herança

- **Em JAVA:**

```
class Pessoa {  
    String nome;  
    void cadastrar() {  
        ...  
    }  
}  
  
class Cliente extends Pessoa {  
    int conta;  
    void pagamento() {  
        ...  
    }  
}
```

Herança

- Em termos de código, significa que os atributos e métodos da classe pai são “copiados” para a classe filha
 - Ou seja, objetos da classe filha podem ter valores para atributos que foram definidos somente na classe pai
 - Além de ser possível chamar métodos que foram definidos somente na classe pai

Herança

- Em Java, TODA classe (com exceção de uma, especial) tem exatamente uma superclasse associada
 - Não é possível definir uma classe sem que ela tenha uma superclasse
 - Isso porque, caso o programador não defina uma superclasse, o compilador define automaticamente que a superclasse será “java.lang.Object”
- Ou seja, toda classe é descendente de java.lang.Object
 - Exceto, é claro, a própria java.lang.Object

Herança

- Uma das coisas mais interessantes da herança deriva do fato de que
 - Sistema de tipos JAVA considera a hierarquia de classes
 - Objetos da subclasse são também consideradas instâncias de objetos da superclasse
 - Isso vale para atribuições, passagem de parâmetros, retorno de funções, chamadas de métodos, etc.

Herança

- **Ex:**

```
class Animal {
    double peso;
}

class AnimalDeRaca extends
    Animal {

    String nome;
    int idade;
    String raca;
}

class AnimalSelvagem
    extends Animal {

    String regioao;
}
```

```
class Peixe extends
    Animal {

    String nome;
    String especie;
}

class Cachorro extends
    AnimalDeRaca {

}

class Cavalo extends
    AnimalDeRaca {

}

class Onca extends
    AnimalSelvagem {

}
```

Herança

```
Cachorro c = new Cachorro();  
c.nome = "Bidu";  
c.peso = 10.0;  
c.idade = 3;  
c.raca = "Schnauzer";  
Animal a = c;  
System.out.println(a.peso);  
System.out.println(a.nome); // erro
```

Herança

```
void examinar(Animal a) {  
    System.out.println(a.peso);  
}  
  
void vacinar(Cachorro c) {  
    System.out.println("Vacinando "+c.nome);  
    ...  
    examinar(c); // ok  
    vacinar(a); // erro
```

Herança

```
Animal buscarPeloNome(String nome) {  
    // consulta o banco  
    Cachorro c = new Cachorro();  
    return c;  
}  
  
...  
Animal x = buscarPeloNome("Bidu");
```

Herança

- Quando o programador sabe que um objeto é de determinado tipo, mas o compilador não, é possível dar uma “ajudinha”

- Ex:

```
Cachorro c = new Cachorro();
```

```
Animal a = c;
```

```
...
```

```
// em outro trecho do programa
```

```
vacinar(a); // erro
```

```
Cachorro c2 = (Cachorro)a;
```

```
vacinar(c2); // ok
```

```
vacinar((Cachorro)a);
```

Herança

- É também possível programar a detecção de tipo
- Ex:

```
Animal buscarPeloNome(String nome) {  
    // consulta o banco  
    Animal encontrado = new Cachorro();  
    if(encontrado instanceof Cachorro) {  
        Cachorro c = (Cachorro)encontrado;  
        vacinar(c);  
        return c;  
    }  
    else return encontrado;  
}
```

Herança

- Essa funcionalidade permite a criação de “containers” genéricos
 - Arrays, listas, matrizes, etc.

- **Ex:**

```
Animal[] cadastro = new Animal[20];
```

```
cadastro[0] = new Cachorro();
```

```
cadastro[1] = new Cavalo();
```

```
cadastro[2] = new Onca();
```

```
...
```


Herança

- **Ex:**

```
Object[] coisas = new Object[20];  
coisas[0] = new Cachorro();  
coisas[1] = "Uma string";  
coisas[2] = new Integer(20);  
coisas[3] = new  
    java.util.Connection();  
...
```

Herança

- Nesse tipo de container genérico, é necessário fazer a conversão explícita ao utilizar os elementos armazenados
- Ex:

```
for(int i = 0; i < cadastro; i++) {  
    Animal a = cadastro[i];  
    if(a instanceof AnimalDeRaca) {  
        AnimalDeRaca adr = (AnimalDeRaca)a;  
        // código específico de animais de raça;  
    }  
    else if(a instanceof AnimalSelvagem) {  
        AnimalSelvagem as = (AnimalSelvagem)a;  
        // código específico de animais selvagens;  
    }  
}
```

Private vs protected

- Atributos e métodos “private” só podem ser acessados de dentro da própria classe
- Atributos e métodos “protected” podem ser acessados de dentro da própria classe, mas também de dentro das subclasses
- Nem private nem protected permitem acesso externo
 - Um método protected não pode ser chamado por uma outra classe

Private vs protected

- Ex:

```
class Pessoa {
    private String cpf;
    protected String nome;
    public void imprimirDeclaracao() {
        System.out.println("Eu, "+nome+", portador de CPF "+cpf+", declaro estar
        ciente das condições deste contrato.");
    }
}

class Cliente extends Pessoa {
    protected double saldo;
    public void imprimeRecibo() {
        if(saldo < 0) {
            System.out.println("Eu, "+nome+", portador de CPF "+cpf+", declaro estar
            devendo "+saldo+" reais para a loja.");
        }
    }
}

...

Cliente c = new Cliente();
System.out.println(c.cpf);
System.out.println(c.nome);
System.out.println(c.saldo);
```

Herança e construtores

- Lembrando:
 - Toda classe tem um construtor
 - Definido pelo programador ou pelo compilador
 - Toda classe tem uma superclasse
 - Definida pelo programador ou pelo compilador
- Regra:
 - Todo construtor deve, na sua primeira linha:
 - Fazer uma chamada para outro construtor
 - Já vimos isso, com a chamada `this(...)`
 - Fazer uma chamada para um construtor da superclasse
 - Usando uma chamada `super(...)`
 - Se o programador não fizer isso, o compilador fará:
 - Inserindo uma chamada `super()` automaticamente

Herança e construtores

- Situação 1: Superclasse tem construtor padrão

```
class Pessoa {  
    String nome, cpf;  
    public Pessoa() {  
        this.nome = "";  
        this.cpf = "";  
        System.out.println("Construindo pessoa");  
    }  
}  
  
class Cliente extends Pessoa {  
    double saldo;  
    public Cliente(String nome, String cpf, double saldo) {  
        // Compilador insere super(); aqui  
        this.nome = nome;  
        this.cpf = cpf;  
        this.saldo = saldo;  
        System.out.println("Construindo cliente");  
    }  
}
```

Herança e construtores

- Situação 2: Superclasse tem construtor padrão e outros

```
class Pessoa {
    String nome, cpf;
    public Pessoa() {
        this.nome = "";    this.cpf = "";
        System.out.println("Construindo pessoa");
    }
    public Pessoa(String nome, String cpf) {
        // Compilador insere super(); aqui
        this.nome = nome;    this.cpf = cpf;
        System.out.println("Construindo pessoa 2");
    }
}

class Cliente extends Pessoa {
    double saldo;
    public Cliente(String nome, String cpf, double saldo) {
        super(nome, cpf); // se o programador "esquecer" aqui, o compilador insere super();
        this.saldo = saldo;
        System.out.println("Construindo cliente");
    }
}
```

Herança e construtores

- Situação 3: Superclasse não tem construtor padrão

```
class Pessoa {  
    String nome, cpf;  
    public Pessoa(String nome, String cpf) {  
        this.nome = nome;  
        this.cpf = cpf;  
    }  
}  
  
class Cliente extends Pessoa {  
    double saldo;  
    public Cliente(String nome, String cpf, double saldo) {  
        super(nome, cpf); // se o programador "esquecer" aqui,  
                           // o compilador irá inserir super(),  
                           // gerando um erro  
        this.saldo = saldo;  
    }  
}
```


Resumo

Resumo

- Vimos alguns conceitos da OO
 - Encapsulamento
 - Ocultação da informação
 - Coesão
 - Acoplamento
- Vimos como Java oferece suporte à OO
 - Conceitos básicos de Java
 - Classes e objetos
 - Encapsulamento de atributos
 - Construtores
 - Static
 - Relacionamentos entre classes

Resumo

- Na próxima aula
 - Sobrescrita de métodos
 - Classes abstratas
 - Interfaces
 - Tratamento de exceções
 - Pacotes

Fim