

Paradigmas de Linguagens de Programação

Aula 9

Relacionamento de Objetos

Claudiane Maria Oliveira
claudiane@gmail.com

Introdução

2

- Sistemas em OO são desenvolvidos usando classes
- Classes são instanciadas em objetos que podem enviar mensagens entre si
- Sistemas orientados a objetos dependem muito de como os objetos se relacionam:
 - O planejamento/modelagem das classes se baseiam nesses relacionamentos.
- Objetos se comunicam através de mensagens, que são implementadas como métodos públicos de suas respectivas classes

Introdução

3

- Classes possuem relacionamentos entre elas (para comunicação)
 - Compartilham informações
 - Colaboram umas com as outras
- Principais tipos de relacionamentos
 - Associação
 - Agregação
 - Composição
 - Herança

Quais objetos fazem parte deste sistema?

4



- Como acham que os objetos se relacionam?
- O que cada um tem a ver com o outro?

Quais objetos fazem parte deste sistema?

5

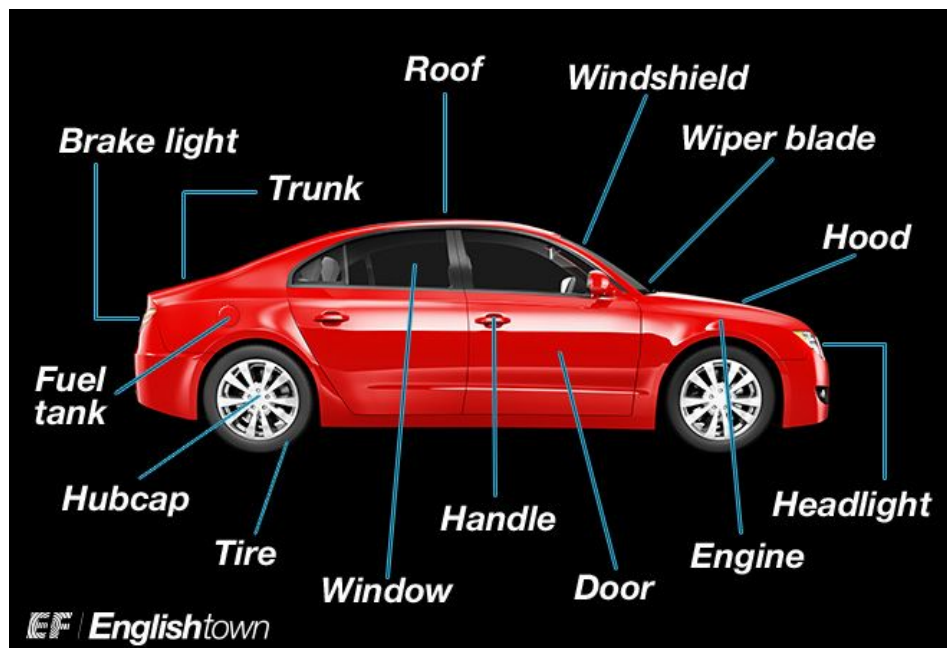


- Posso pensar que existiriam, por exemplo, os objetos:
 - Produto, vendedor e compra.
- Uma compra pode ser formada por um conjunto de produtos e estar relacionada a um vendedor

Exemplos de Relacionamentos

6

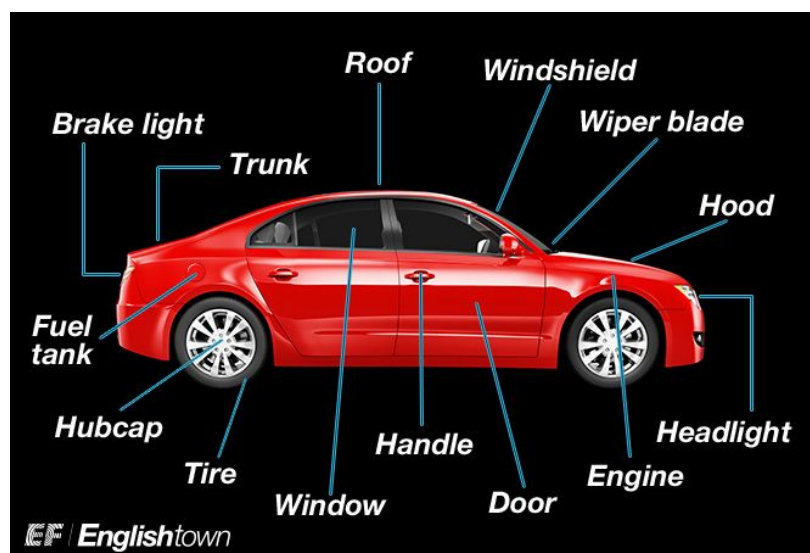
- Veremos que os tipos de relacionamento entre objetos são muito importantes para modelar um software OO
 - Qual é o relacionamento entre um carro e um motor?



Exemplos de Relacionamentos

7

- Qual é o relacionamento entre um carro e um motor?
 - Podemos dizer que um carro tem um motor?
- E os pneus?
 - Da mesma forma, um carro tem vários pneus



FAGAMMON
FACULDADE PRESBITERIANA GAMMON

Exemplos de Relacionamentos

8

- Qual o relacionamento entre os itens e a compra?
 - Da mesma forma, uma compra tem vários itens..



FAGAMMON
FACULDADE PRESBITERIANA GAMMON

Composição

9

- Chamamos esses relacionamento de Composição!!!
- A composição implementa o relacionamento **TEM-UM (HAS-A)**
- Eles se caracterizam por ser um relacionamento:
 - **Todo / Parte!**
- Se não tenho um carro, não faz sentido criar as rodas
- Se o banco faliu, não preciso mais de agências
- Sem uma compra não tenho itens de compra

Composição

10

- A **composição** acontece quando duas classes possuem relacionamento tem um ou todo/parte.
- Nela o tempo de vida dos objetos é controlado pelo todo.

Exemplo de Composição

11

```
class Conta
{
    // atributos da minha classe
    private float saldo;
    private int numero;

    // construtores
    Conta(int numero)
    {
        saldo = 0;
        this.numero = numero;
    }

    Conta(int numero, float saldoInicial)
    {
        this.numero = numero;
        saldo = saldoInicial;
    }

    // métodos
```

```
class Agencia
{
    private String nome;
    private Conta[] contas;
    private int contasAtivas;

    public Agencia(String nome, int nroContas)
    {
        this.nome = nome;
        contas = new Conta[nroContas];
        contasAtivas = 0;
    }

    public int criarConta()
    {
        // usando variavel para deixar mais claro o codigo
        int proxNroConta = contasAtivas+1;

        contas[contasAtivas] = new Conta(proxNroConta);
        contasAtivas++;

        return proxNroConta;
    }
}
```



Composição

12

- Composição denota relacionamentos todo/parte, onde o todo é constituído de partes.
- Exemplo: relacionamento entre Banco e Agência.



Banco é o todo e os objetos Agência são as partes. Os objetos Agência não podem existir independentemente do objeto Banco. Se o banco for excluído (encerrar suas atividades) as agências também serão excluídas. O inverso não é necessariamente verdadeiro.

Composição

13

```
public class Ponto {  
    private double x; // coordenada x  
    private double y; // coordenada y  
    public Ponto (double x, double y)  
        this.x = x; this.y = y;  
}  
//...
```



```
public class Quadrado {  
    private Ponto p1, p2, p3, p4;  
    public Quadrado (Ponto p1, Ponto p2, Ponto p3, Ponto p4) {  
        //...  
    }  
    //...  
}
```



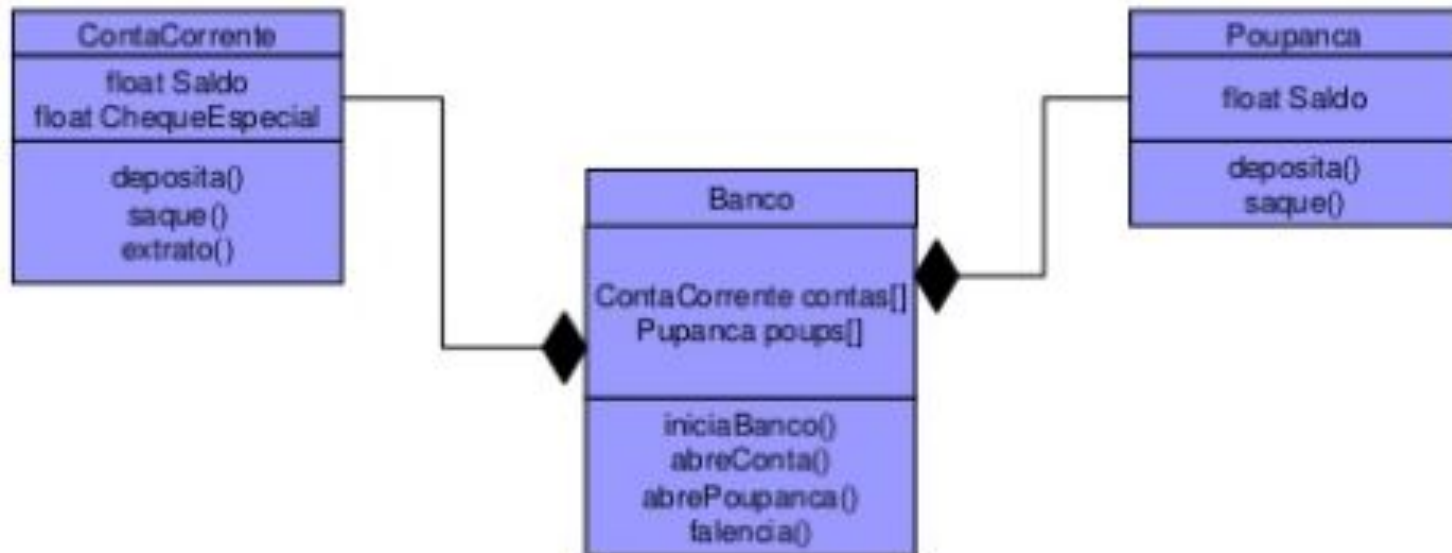
Composição

14

- É um relacionamento todo/parte.
- Podemos dizer também que é um relacionamento tem-um ou tem-vários.
- Objetos-parte têm que pertencer ao objeto-todo
 - O todo não existe (ou não faz sentido) sem as partes.
 - Ou, as partes não existem sem o todo
 - E as partes pertencem a um único todo.
- Por isso, é o todo quem controla o tempo de vida das partes.

Composição

15



Composição

16

```
public class Poupanca {
    float Saldo;

    void saque() {
        Saldo -= 10.0f;
        System.out.println("Novo Saldo →" + Saldo);
    }
    void deposito() {
        Saldo += 10.0f;
        System.out.println("Novo Saldo →" + Saldo);
    }
}

public class ContaCorrente {
    float Saldo;

    void saque() {
        Saldo -= 100.0f;
        System.out.println("Novo Saldo →" + Saldo);
    }
    void saque() {
        Saldo -= 100.0f;
        System.out.println("Novo Saldo →" + Saldo);
    }
}
```

```
public class Banco {
    Poupanca[] pops;
    ContaCorrente[] cc;
    int numConta, numPoupanca;
    void iniciaBanco() {
        pops = new Poupanca[100];
        cc = new ContaCorrente[100];
        numConta = 1;
        numPoupanca = 1;
    }
    void abreConta() {
        cc[ numConta ] = new ContaCorrente();
        numConta++;
    }
    void abrePoupanca() {
        pops[ numConta ] = new Poupanca();
        numPoupanca++;
    }
    void falencia() {
        for (int i = 0; i < 100; i++) {
            pops[ i ] = null;
            cc[ i ] = null;
        }
    }
}
```



Agregação

17

- No nosso exemplo anterior, qual seria o relacionamento entre os Produtos e o Estoque?



- Um estoque tem vários produtos. E um produto é parte de um estoque. Então à primeira vista parece composição!!

Agregação

18

- E os produtos que vão para as prateleiras?
- E os produtos que chegam ao supermercado e vão direto para as prateleiras?
 - Nesse caso não é o estoque que controla o tempo de vida dos produtos. Ou seja, não é o todo que controla o tempo de vida das partes.
- Esse relacionamento é o que chamamos de Agregação!!!



Agregação

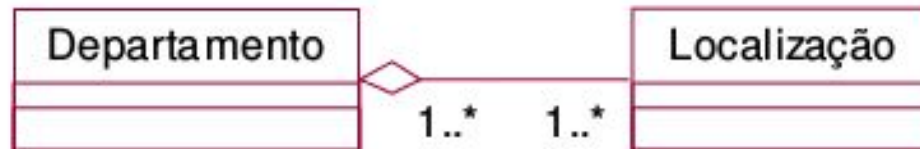
19

- A **Agregação** também modela relacionamentos todo/parte (tem-um).
- Diferentemente da composição, os objetos em uma agregação podem existir independentemente uns dos outros.
- Alguns autores dizem que a agregação é um tipo menos rigoroso de composição. Denota uma **composição menos rigorosa**.

Agregação

20

- Exemplo: relacionamento entre Departamento e Localização.



- Um departamento possui várias localizações, mas uma localização pode pertencer a mais de um departamento e portanto não é excluída se um departamento for excluído.
- Representada por um losango aberto do lado todo.

Agregação e Composição

21

- Agregação e Composição são implementadas usando um atributo privado
- Como consigo saber qual está sendo usada em um sistema?
- Sintaticamente são parecidas, mas não semanticamente.
- Na agregação o todo não controla o tempo de vida da parte

Associação

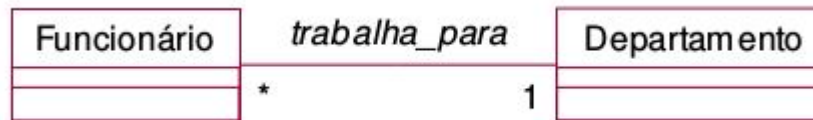
22

- Suponha que um professor que dá aulas particulares, resolva fazer um sistema para controlar suas atividades.
- Ele criou uma classe Professor e uma classe Aluno. Qual é o relacionamento entre essas classes?
- O professor dá aulas para o aluno e o aluno tem aulas com o professor. Mas não faz sentido dizer que o aluno é parte do professor ou que o professor é formado por alunos.
- Neste exemplo, temos uma **associação**...

Associação

23

- A **Associação** indica que um objeto contém ou está conectado a outro objeto. Um objeto usa outro objeto.
- **Exemplo:** relacionamento entre Funcionário e Departamento, indicando que um funcionário trabalha para um departamento e que em um departamento trabalham vários funcionários.



- Os objetos **Funcionário** e **Departamento** existem independentemente um do outro e não denotam todo/parte.

Associação

24

- Exemplo Funcionário e Departamento
- O lado '1' do relacionamento é declarado como um atributo simples e o lado 'muitos' é declarado como uma coleção (array, Vector, ...).

```
public class Funcionário {  
    private Departamento depto;  
    ... }
```

```
public class Departamento {  
    private Funcionário[] funcs;  
    ... }
```

Associação

25

- **Agregação e Composição** são subtipos de Associação que ajudam a refinar mais os modelos.



Associação

26

- Se dois objetos têm relação um com o outro... Mas, um não é parte do outro. Então eles possuem uma relação de **associação**.
- Se a associação passa a ter características todo/parte, mas o todo não controla o tempo de vida das partes, a relação é de **agregação**.
- Mas, se além de ser todo/parte, a parte não existe sem o todo: temos uma relação de **composição**.



Multiplicidade

27

0..1	No máximo um. Indica que os objetos da classe associada não precisam obrigatoriamente estar relacionados.
1..1	Um e somente um. Indica que apenas um objeto da classe se relaciona com os objetos da outra classe.
0..*	Muitos. Indica que podem haver muitos objetos da classe envolvidos no relacionamento
1..*	Um ou muitos. Indica que há pelo menos um objeto envolvido no relacionamento.
3..5	Valores específicos.

Herança

28

- Herança denota especialização.
- Herança permite a criação de novos tipos com propriedades adicionais ao tipo original.
- Em geral, se uma classe B estender (herdar) uma classe A, a classe B vai herdar métodos e atributos da classe A e implementar alguns recursos a mais.
- Herança implementa o relacionamento É-UM (IS-A).

Herança

29

- Exemplos:
 - Classes Mamífero e Cachorro
 - Cachorro É-UM Mamífero
 - Classes Veículo e Carro
 - Carro É-UM Veículo
 - Classes Empregado e Gerente
 - Gerente É-UM Empregado

Herança

30

- A classe herdeira é chamada de subclasse (ou classe derivada) e a classe herdada é chamada de superclasse (ou classe base).
 - Carro é subclasse de Veículo
 - Veículo é superclasse de Carro
 - Carro é a classe derivada da classe base Veículo

Herança

31

- Facilita a evolução (extensibilidade):
 - Criar novas classes é muito mais rápido.
 - Diminui custos e riscos de novos desenvolvimentos.
- Facilita a manutenção:
 - Ao arrumar um problema comum a todos as classes, será arrumado apenas uma vez e valerá pra todas.

Herança

32

- Melhora a reutilização:
 - Os códigos comuns (que ficaram na classe base) estão sendo reutilizados nas demais classes.
- Melhora a robustez:
 - Dessa forma, códigos já testados/validados estão sendo usados mais de uma vez, ao invés de se criar novos códigos que podem ter erros.

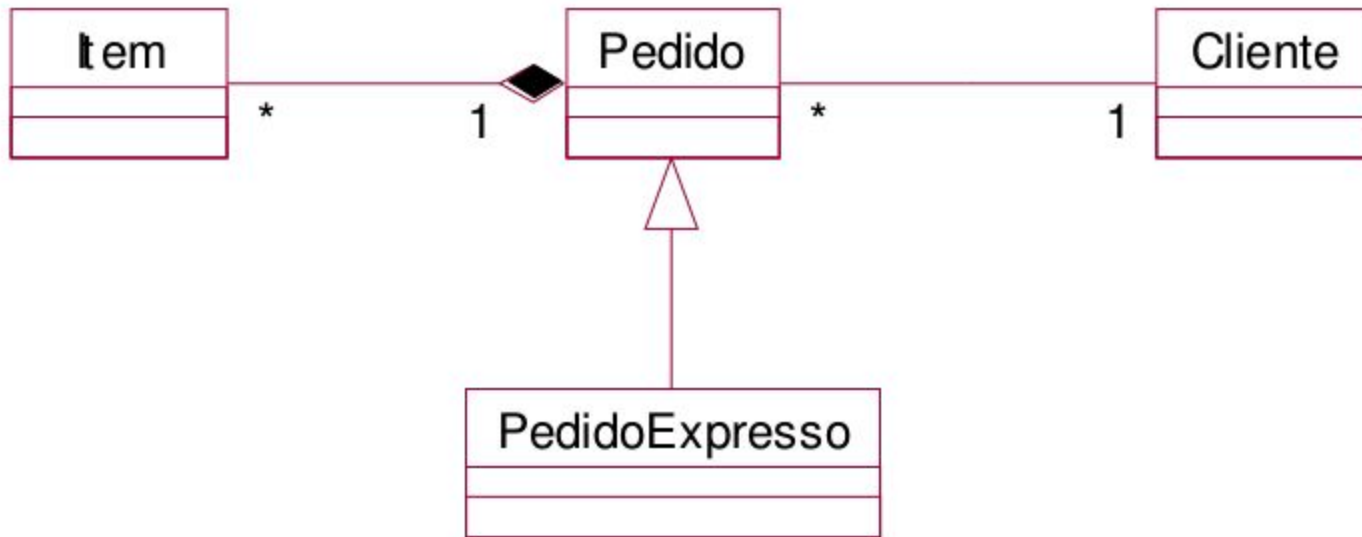
Herança

33

- Através da herança nós então conseguimos:
 - Criar novas classes a partir de classes existentes.
 - Tais classes absorvem características e comportamentos.
 - Mas também adicionam novos recursos.
- A classe a partir da qual outras classes herdam suas características e comportamentos é chamada de superclasse (ou classe pai).
- As classes que herdam das superclasses são chamadas de subclasses (ou classes filhas).

Herança

34



- Herança: PedidoExpresso É-UM Pedido
- Composição: Pedido TEM-UM (tem vários) Item
- Associação: Pedido está associado a um Cliente

Exercício

35

1) Considerando os conceitos de OO, como você relacionaria as classes abaixo?

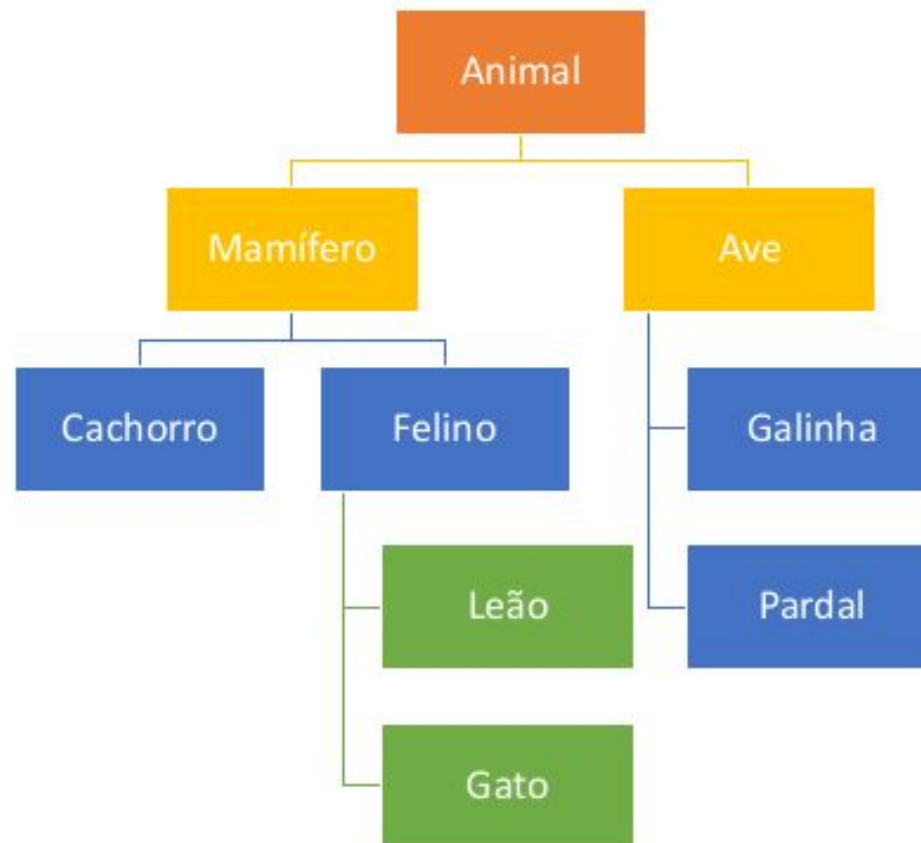
Pessoa, Empregado, Empresa, PequenaEmpresa, GrandeEmpresa.

2) Dada a classe Conta abaixo, defina as classes ContaPoupanca e ContaEspecial. Suponha que poupança possui rendimento mensal de 0.5% em uma única data fixa e que a ContaEspecial permite empréstimo com juros de 10% ao mês

Conta
Numero saldo
getNumero getSaldo depositar sacar transferir

Hierarquia de Herança

36



Exercício

37

1) Crie uma hierarquia de herança para as seguintes classes:

Docente, AlunoGraduacao, ProfessorTitular, ComunidadeAcademica, ProfessorAssistente, AlunosPósGraduacao, TecnicoAdministrativo, Aluno, Reitor.

2) Apresente uma hierarquia de herança para alimentos..
ex. Frutas, verduras, carboidratos, etc..

Herança

38

- Suponha que eu tenha uma classe veículo com os atributos: nome, marca, anoFabricacao, peso, nroRodas, valorDeVenda e com os métodos calcularImposto e trocarPneu
- Posso criar a classe carro que herde da classe veículo e não adiciona nada?

Herança

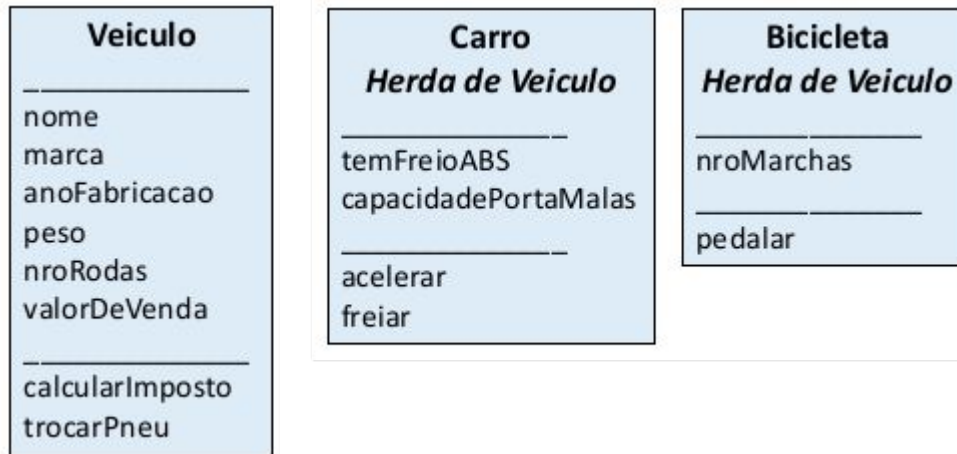
39

- Suponha que eu tenha uma classe veículo com os atributos: nome, marca, anoFabricacao, peso, nroRodas, valorDeVenda e com os métodos calcularImposto e trocarPneu
- Posso criar a classe carro que herde da classe veículo e não adiciona nada? Posso!!!
- O grande poder da herança está em:
 - Criar características novas.
 - E/ou criar comportamentos novos.
 - E/ou modificar comportamentos.



Herança

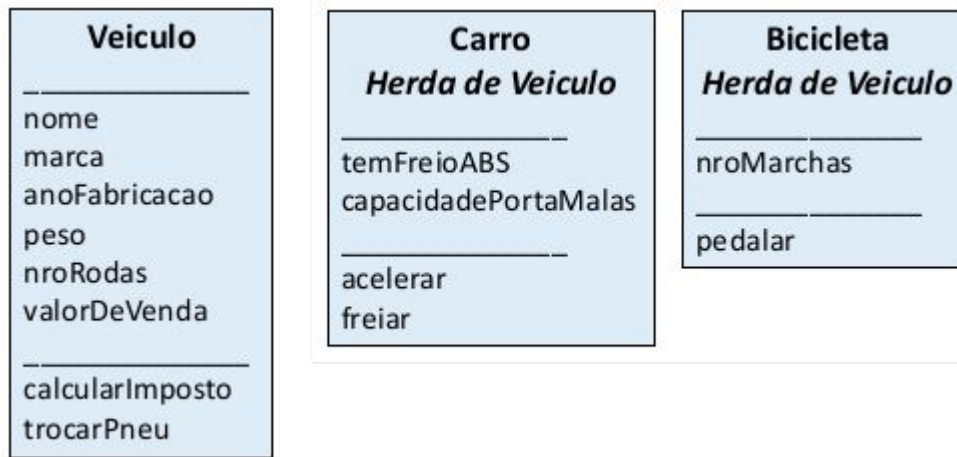
40



- A subclasse “é maior” que a superclasse.
- Pois tem tudo que a superclasse tem e ainda acrescenta mais coisas.

Herança

41

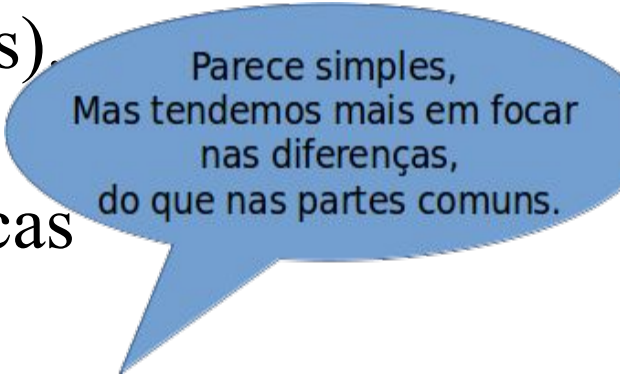


- Além disso, a subclasse é mais especializada que a superclasse.
- Ela representa um número menor de objetos (possíveis).
Ex: a classe **Veiculo** pode representar qualquer tipo de veículo (carros, bicicletas, caminhões, etc.). Já a classe **Carro** representa um tipo específico veículo (carros).

Herança

42

- Para modelar um programa Orientado a Objetos começamos identificando os objetos.
- E, em seguida, identificamos suas características (atributos) e comportamentos (métodos).
- E agora adicionamos o seguinte ...
 - Identificar objetos com características e comportamentos comuns.
 - E caso eles tenham relação é um, devemos modelá-los utilizando herança



Parece simples,
Mas tendemos mais em focar
nas diferenças,
do que nas partes comuns.

Herança

43



Sintaxe:

- Palavra-chave **extends**.
- Utiliza-se na definição da subclasse a palavra-chave **extends** seguida pelo nome da super-classe.

```
class SuperClasse {  
    ...  
}  
class SubClasse extends SuperClasse {  
    ...  
}
```



Herança - Exemplo 1

44

```
class Funcionario {  
    String nome;  
    String cpf;  
    double salario;  
    // métodos devem vir aqui  
}
```

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
    public boolean autentica(int senha) {  
        ...  
    }  
}
```



Herança - Exemplo 2

45

```
class TestaGerente {  
    public static void main(String[] args) {  
        Gerente gerente = new Gerente();  
  
        // podemos chamar metodos do Funcionario:  
        gerente.setNome("João da Silva");  
  
        // e tambem metodos do Gerente!  
        gerente.setSenha(4231);  
    }  
}
```



Herança 3

46

- Dizemos que a classe **Gerente** herda todos os atributos e métodos da classe mãe, no nosso caso, a Funcionario
- Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente

Herança 4

47

- E se precisamos acessar os atributos que herdamos?
- Atributos de Funcionario como Public?

Herança

48

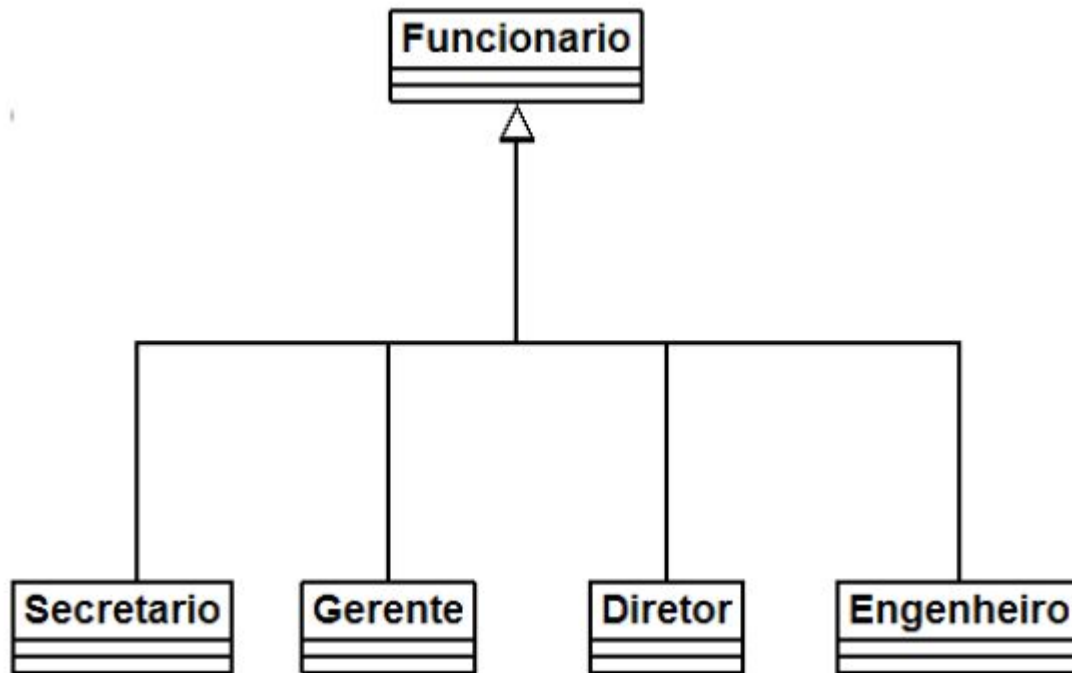
- E se precisamos acessar os atributos que herdamos?
 - Atributos de Funcionario como Public?
- Modificador Protected
 - Fica entre o public e private
 - Só pode ser acessado pela classe e subclasses

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
    // métodos devem vir aqui  
}
```


Herança

49

- Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java.



Reescrita de Método

50

- O comportamento de um método que foi herdado pode ser alterado através da reescrita (override) desse método na subclasse.

“Todo fim de ano, os funcionários do banco recebem uma bonificação. Os funcionários comuns recebem 10% e os gerentes 15%”

Reescrita de Método

51

➤ Classe Funcionario

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.10;  
    }  
}
```



Reescrita de Método

52

- Se a Classe Gerente manter como está, ela vai herdar o método getBonificacao()

```
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```

Reescrita de Método

53

- Se a Classe Gerente manter como está, ela vai herdar o método `getBonificacao()`

```
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```

- Resultado: 500 pois a classe Gerente herda o método `getBonificacao()` da superclasse Funcionario.

Reescrita de Método

54

- No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos reescrever (reescrever, sobrescrever, override)

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
}
```



Reescrita de Método

55

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
    // ...  
}  
  
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```



Reescrita de Método

56

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
    // ...  
}  
  
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```

Resultado: 750 pois o método `getBonificacao()` da classe `Gerente` foi reescrito.

Reescrita de Método

57

- E se eu sobrescrever um método, mas precisar fazer o que a superclasse fazia, vou reescrever o código...
- **super** é usado para invocar explicitamente o método da superclasse imediatamente superior.
- Usamos `super.Metodo(...)`
 - Onde colocarmos essa linha o sistema chamará o método da superclasse e depois continuará na subclasse

super

58

- Super é também usado para acessar membros (atributos e métodos) da superclasse imediatamente superior
- `super.nomeDoAtributo` ou `super.nomeDoMétodo()`

Exemplo – Invocando o método Reescrito

59

- “A bonificação de um gerente é igual a bonificação de um funcionário comum mais a adição de R\$ 1000”.

Exemplo – Invocando o método Reescrito

60

- “A bonificação de um gerente é igual a bonificação de um funcionário comum mais a adição de R\$ 1000”.

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.10 + 1000;  
    }  
    // ...  
}
```

Exemplo – Invocando o método Reescrito

61

- “A bonificação de um gerente é igual a bonificação de um funcionário comum mais a adição de R\$ 1000”.

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return super.getBonificacao() + 1000;  
    }  
    // ...  
}
```

Exemplo – Invocando o método Reescrito

62

- A invocação do método `super()`, se presente, deve estar na primeira linha.
- Se o método `super()` não for usado, implicitamente o compilador faz a invocação do construtor `super()` default (sem argumentos) para cada construtor definido.
 - É sempre interessante ter o construtor default definido.
 - A invocação direta pode ser interessante quando se deseja invocar algum construtor que não o default , como no exemplo seguinte.

Mais sobre Herança

63

- Princípio da Substituição: “Podemos substituir um objeto de subclasse em que um objeto de superclasse é esperado”
- Podemos usar um objeto de uma subclasse onde um objeto da superclasse é esperado, ou seja, em substituição a ele. `Carro meuCarro = new Carro();`

```
Veiculo v1 = new Veiculo();  
Veiculo v2 = new Carro();  
Veiculo v3 = new Bicicleta();
```



Mais sobre Herança

64

- Princípio da Substituição: “Podemos substituir um objeto de subclasse em que um objeto de superclasse é esperado” o sentido inverso não é permitido.

```
Carro meuCarro = new Carro();
```

```
Veiculo v1 = new Veiculo();  
Veiculo v2 = new Carro();  
Veiculo v3 = new Bicicleta();
```

```
Carro car1 = new Veiculo(); // isso é um erro!  
// não sabemos se o o veículo é um carro, ele pode ou não ser um  
carro.!!!
```

```
Carro car2 = new Bicicleta(); // erro também!
```


Mais sobre Herança

65

- A subclasse possui todos os métodos e atributos da superclasse. Subclasse É-UM superclasse.

```
SuperClasse sp;  
SubClasse sb = new SubClasse();  
sp = sb; // correto  
SuperClasse sp2 = new SubClasse(); // correto  
sb = sp2; // incorreto
```

- Todos os métodos na SuperClasse são herdados sem modificação na SubClasse.
- Todos os atributos que formam a SuperClasse formam também a SubClasse.

Exemplo

66

```
Gerente g = new Gerente(...);  
Funcionario f;  
f = g; // ok, funciona !!!
```

- As variáveis f e g referem-se ao mesmo objeto. Mas a variável f é considerada pelo compilador apenas como uma referência a **objeto Funcionario**.
- Suponha que exista um método:
- ```
public void setBonus(double b) // definido somente na classe Gerente
g.setBonus(5000); // ok, funciona !!!
f.setBonus(5000); // ??
```

# Exemplo

67

```
Gerente g = new Gerente(...);
Funcionario f;
f = g; // ok, funciona !!!
```

```
public void setBonus(double b) // definido somente na classe Gerente
g.setBonus(5000); // ok, funciona !!!
f.setBonus(5000); // Erro !!! Pois f é considerada como uma referência a
um Funcionario, e o método setBonus não é um
método da classe Funcionario.
```

```
Funcionario f2 = new Funcionario(...);
Gerente g2 = f2; // ???
```

# Exemplo

68

```
Gerente g = new Gerente(...);
Funcionario f;
f = g; // ok, funciona !!!
```

```
public void setBonus(double b) // definido somente na classe Gerente
g.setBonus(5000); // ok, funciona !!!
f.setBonus(5000); // Erro !!! Pois f é considerada como uma referência a
um Funcionario, e o método setBonus não é um
método da classe Funcionario.
```

```
Funcionario f2 = new Funcionario(...);
Gerente g2 = f2; // Erro !!! Não se pode atribuir uma referência de
superclasse a uma variável subclasse.
“ Nem todo funcionário é um gerente”.
```



# Mãos a obra

69

## Veiculo

---

nome  
marca  
anoFabricacao  
peso  
nroRodas  
valorDeVenda

---

calcularImposto  
trocarPneu

## Carro

### *Herda de Veiculo*

---

temFreioABS  
capacidadePortaMalas

---

acelerar  
freiar

## Bicicleta

### *Herda de Veiculo*

---

nroMarchas

---

pedalar



# Mãos a obra

70

Observações importantes considerando um exemplo Veículo e Carro:

- Na classe carro definimos apenas os atributos e métodos que são específicos de um carro.
  - Ou seja, não estão na classe Veículo.
- Ao criar um Carro os métodos públicos do Veículo e do Carro estão disponíveis.
- Dentro da classe Carro os atributos e métodos públicos de Veículo estão disponíveis.
  - E nesse caso não é necessário usar “objeto.”, pois um carro é um veículo.

# Mãos a obra

71

- Como podemos tratar o método calcularImposto?
- A classe veículo possui esse método, mas ele tem funcionamento diferente para um Carro ou uma Bicicleta.
- Fazemos isso através da sobrescrita do método.
- Para sobrescrever um método basta reimplementá-lo na subclasse.
- Ou seja, implementar um método com a mesma assinatura mas com um novo código.

# Classe Object

72

- Object é uma classe em Java da qual todas as outras derivam.
- Quando uma classe é criada e não há nenhuma referência a sua superclasse, implicitamente a classe criada é derivada diretamente da classe Object.
- Todos os objetos podem invocar os métodos da classe Object.



# Classe Object

73

- Alguns métodos da classe Object:
  - boolean equals (Object obj)
    - Testa se os objetos são iguais (apontam para o mesmo local).
  - Class getClass ()
    - Retorna a classe do objeto.
  - Object clone ()
    - copia o conteúdo de um objeto para outro

# Construção de Objetos Derivados

74

- Processo de execução de métodos construtores envolvendo herança:
- 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.

# final

75

- final é uma palavra chave em Java que indica que um atributo, um método ou uma classe não podem ser modificados ao longo do restante da hierarquia de descendentes
- Atributos final são usados para definir constantes.
  - Apenas valores dos tipos primitivos podem ser usados para definir constantes
  - O valor é definido na declaração ou nos construtores da classe, e não pode ser alterado
  - Exemplo: final double PI = 3.14

# final

76

- Para objetos e arranjos, apenas a referência é constante (o conteúdo do objeto ou arranjo pode ser modificado).
- Na lista de parâmetros de um método, final indica que o parâmetro não pode ser modificado.
- `public void exemplo (final int par1, double par2) { ... }`  
**par1** não pode ser modificado dentro do método.

# final

77

- Para método, final indica que o método não pode ser redefinido em classes derivadas.

- final public void exemplo () { ... }

método exemplo **não** pode ser redefinido em classes herdeiras.

- Para classe, final indica que a classe não pode ser derivada

- final public class Exemplo { ... }

public class subExemplo extends Exemplo { ... } **//ERRO**

- classe Exemplo não pode ser derivada.



# Exercícios

78

1. Verdadeiro ou Falso. Se for falso, explique porquê.
2. Os construtores de superclasse não são herdados por subclasses.
3. Um relacionamento “tem um” é implementado via herança.
4. Uma classe Carro tem um relacionamento “é um” com as classes Volantes e Freio.
5. A herança estimula a reutilização de software de alta qualidade comprovada.

# Exercícios

79

1. Explique a diferença entre composição e herança no paradigma da programação orientada a objetos. Utilize exemplos na sua explicação.
2. Discuta a função de atributos estáticos (static) no paradigma da programação orientada a objetos.
3. Discuta de que maneira a herança promove a reutilização de software, economiza tempo durante o desenvolvimento de programa e ajuda a evitar erros.

# Paradigmas de Linguagens de Programação

## Aula 9

## Relacionamento de Objetos

*Claudiane Maria Oliveira*  
*claudiane@gmail.com*

