UNIRIO

Agregação e Herança

Aula 7
BSI — 2018.2

Jefferson Elbert Simões

CCET/DIA

11 de setembro de 2018

Previously on TP2...

- Fundamentos: classes, objetos, atributos, métodos
- Referências
- Construtores e destrutores
- Métodos e atributos estáticos
- Encapsulamento e visibilidade
- Enumerações e (intro a) exceções

Até agora, somente consideramos atributos simples, representáveis por tipos primitivos

- Classes apenas "organizam" dados/funções
- Programação baseada em objetos

A partir da aula de hoje, começamos a relacionar classes

Agregação

- Relacionamento de parte, posse
  - Tipo "has-a"
- Exemplos:
  - Carro + motor
  - Casa + proprietário
  - Pessoa + endereço
  - Metrô + estação
  - Aluno + orientador
- Relação entre dois objetos de classes distintas
- Objeto "parte" pode existir sem objeto "dono"

Agregação

Representação UML

Agregação

Implementação — objetos como atributos de classes:

```
class Motor {
    String modelo;
    int potencia; // HP
}
class Carro {
    int ano;
    String modelo;
    Motor motor;
}
```

Agregação

- Atenção: atributos não são automaticamente instanciados!

```
class Motor {
    int potencia; // HP
}
class Carro {
    Motor motor;



}
Carro fiat147 = new Carro();
fiat147.motor = new Motor();
fiat147.motor.potencia = 75;
System.out.println( fiat147.motor.potencia );
```

Agregação

- Atenção: atributos não são automaticamente instanciados!

```
class Motor {
    int potencia; // HP
}
class Carro {
    Motor motor;



}
Carro fiat147 = new Carro();

fiat147.motor.potencia = 75; // ?
System.out.println( fiat147.motor.potencia ); // ?
```

Agregação

- Atenção: atributos não são automaticamente instanciados!
  - Importante implementar um construtor para isso!

```
class Motor {
    int potencia; // HP
}
class Carro {
    Motor motor;
    Carro() {
        motor = new Motor();
    }
}
Carro fiat147 = new Carro();

fiat147.motor.potencia = 75;
System.out.println( fiat147.motor.potencia );
```

Agregação

- Principais vantagens:
  - Reutilização de código: uma mesma classe pode ser agregada a várias outras
    - Imagine quantos usos teria uma classe `Data`
  - Modularização: implementar métodos na classe agregada e chamá-los na classe principal
    - Separa as funcionalidades e responsabilidades
- Utilizar quando:
  - Nossas classes podem ser vistas como partes de um todo ("has-a")
  - Queremos utilizar funcionalidades de classes existentes (por exemplo, obtidas em pacotes de terceiros)

Herança

- Terceiro conceito fundamental de orientação a objetos
  - 1: abstração de dados; 2: encapsulamento
  - Muito ligado ao quarto conceito: polimorfismo
- Também conhecida como: especialização, derivação, extensão...

- Estabelece uma nova forma de relacionamento entre classes

Herança

Afinal, o que é herança?

- É a descrição de uma classe B a partir da descrição realizada por outra classe A
  - A é a superclasse (classe pai)
  - B é a subclasse (classe derivada, classe filha)
  - B fornece uma descrição "mais detalhada", logo descreve objetos mais específicos
- Relação entre duas classes distintas, encarnada em um único objeto
- Relacionamento do tipo "is-a"
  - Todo objeto da classe B é, também e ao mesmo tempo, um objeto da classe A

Herança

Exemplo (com UML):

Outros exemplos:

- Animal e Cachorro
- Forma Geométrica e Círculo
- Veículo e Carro

Herança

Exemplo (em Java):

```
class Pessoa {
    public String nome;
    public String endereco;
    Pessoa( String nome ) { this.nome = nome; }
    public String FalarNome() {
        return "Meu nome é " + nome + ".";
    }
}
class Aluno extends Pessoa {
    public int matricula;
    public String curso;
    Aluno( String nome, String curso ) {
        this.nome = nome; this.curso = curso;
    }
    public String FalarNomeCurso() {
        return "Meu nome é " + nome +
               " e sou aluno do curso de " + curso + ".";
    }
}
```

Herança

Exemplo (em Java):

```
Pessoa zezinho = new Pessoa("Zezinho das Couves");
zezinho.FalarNome();
zezinho.FalarNomeCurso(); // ?
```

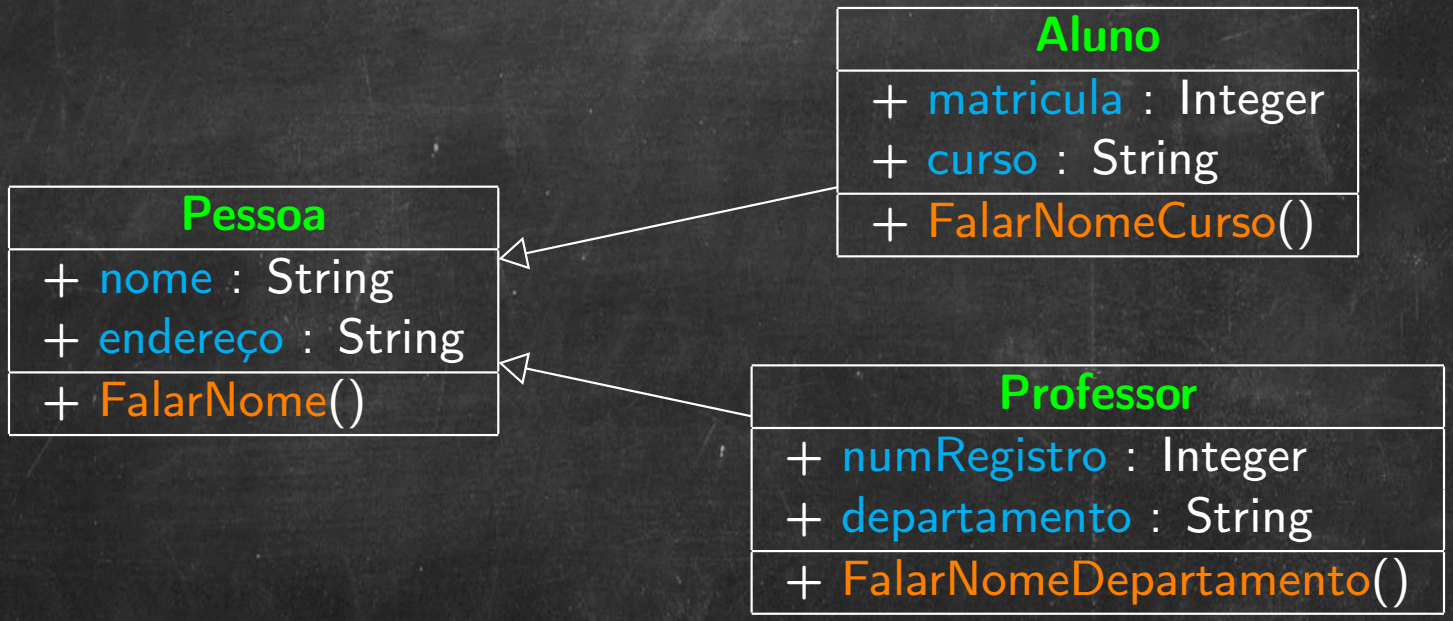
Herança

Exemplo (em Java):

```
Aluno zezinho = new Aluno("Zezinho das Couves","BSI");
zezinho.FalarNome();
zezinho.FalarNomeCurso();
```

Herança

Não estamos restritos a apenas uma subclasse:



- Animal: Cachorro, Gato, Cavalo, ...
- Forma geométrica: Círculo, Quadrado, Triângulo, ...
- Veículo: Carro, Caminhão, Bicicleta, ...

Herança

- Principal vantagem: reutilização de código
  - O código da superclasse não precisa ser reimplementado nas subclasses
  - Mais fácil de manter e garantir consistência
- Diversas outras vantagens (com polimorfismo)

- Utilizar quando:
  - Possuímos várias classes com características comuns e semântica semelhante ("is-a")
  - Queremos estender funcionalidades de classes existentes (por exemplo, obtidas em pacotes de terceiros)

Herança em Java

- Novo tipo de visibilidade: `protected`
  - Atributos e métodos são visíveis à própria classe e suas subclasses, mas não a outras classes
  - Meio-termo entre `public` e `private`

```
class Pessoa {
    public String nome;
    public String endereco;
    public String FalarNome() {
        return "Meu nome é " + nome + ".";
    }
}
class Aluno extends Pessoa {
    public int matricula;
    public String curso;
    public String FalarNomeCurso() {
        return "Meu nome é " + nome +
               " e sou aluno do curso de " + curso + ".";
    }
}
```

Herança em Java

- Novo tipo de visibilidade: `protected`
  - Atributos e métodos são visíveis à própria classe e suas subclasses, mas não a outras classes
  - Meio-termo entre `public` e `private`

```
class Pessoa {
    protected String nome;
    protected String endereco;
    public String FalarNome() {
        return "Meu nome é " + nome + ".";
    }
}
class Aluno extends Pessoa {
    protected int matricula;
    protected String curso;
    public String FalarNomeCurso() {
        return "Meu nome é " + nome +
               " e sou aluno do curso de " + curso + ".";
    }
}
```

Herança e referências

Atribuição:

- Referências à superclasse podem "apontar" para objetos da subclasse
  - Afinal, eles também são da superclasse

```
Pessoa algumDeVoces = new Aluno(); // OK
 Aluno outroDeVoces = new Pessoa(); // erro!
```

- No entanto, temos que tratá-lo como objetos da superclasse
  - Acesso apenas a atributos/métodos da superclasse

```
algumDeVoces.nome  = "Zezinho";
algumDeVoces.curso = "BSI"; // erro!
```

Herança em Java

- Construtores da subclasse precisam "chamar" um construtor da superclasse
  - Construção "em etapas"
  - Se nada for dito, construtor padrão é chamado
  - Pode-se escolher o construtor utilizando a chamada `super(...)` no início do construtor da subclasse

```
class Pessoa {
    Pessoa() {
        System.out.println( "@ Pessoa - 0 args"); }
    Pessoa( String nome ) {
        System.out.println( "@ Pessoa - 1 arg"); }
}
class Aluno extends Pessoa {
    Aluno() {

        System.out.println( "@ Aluno - 0 args"); }
}
Aluno voce = new Aluno();
```

Herança em Java

- Construtores da subclasse precisam "chamar" um construtor da superclasse
  - Construção "em etapas"
  - Se nada for dito, construtor padrão é chamado
  - Pode-se escolher o construtor utilizando a chamada `super(...)` no início do construtor da subclasse

```
class Pessoa {
    Pessoa() {
        System.out.println( "@ Pessoa - 0 args"); }
    Pessoa( String nome ) {
        System.out.println( "@ Pessoa - 1 arg"); }
}
class Aluno extends Pessoa {
    Aluno() {
        super();
        System.out.println( "@ Aluno - 0 args"); }
}
Aluno voce = new Aluno();
```

Herança em Java

- Construtores da subclasse precisam "chamar" um construtor da superclasse
  - Construção "em etapas"
  - Se nada for dito, construtor padrão é chamado
  - Pode-se escolher o construtor utilizando a chamada `super(...)` no início do construtor da subclasse

```
class Pessoa {
    Pessoa() {
        System.out.println( "@ Pessoa - 0 args"); }
    Pessoa( String nome ) {
        System.out.println( "@ Pessoa - 1 arg"); }
}
class Aluno extends Pessoa {
    Aluno() {
        super("Aleatório da Silva");
        System.out.println( "@ Aluno - 0 args"); }
}
Aluno voce = new Aluno();
```

Exercícios

Construa classes para representar os seguintes conceitos. Implemente métodos e atributos, e utilize relacionamentos na forma de agregação e herança, conforme achar adequado.

1. País, Estado, Cidade
2. Veículo, Carro, Bicicleta, Caminhão
3. Cliente, Cliente Pessoa Física, Cliente Empresa
4. Atleta, Time, Partida, Campeonato

Exercícios

Neste exercício, crie classes auxiliares quando for importante/conveniente. Isto é parte da modelagem!

5. Implemente um sistema acadêmico![1] Represente os seguintes conceitos e suas características básicas:

Aluno	nome, data de nascimento, login, matrícula (formato UNIRIO), curso, período de entrada
Professor	nome, data de nascimento, login, matrícula SIAPE (7 dígitos), departamento
Disciplina	nome, código, ementa, carga horária semanal

(continua...)

[1] v0.1

Exercícios

5. . . Requisitos adicionais do sistema:
   - Cada aluno pode, opcionalmente, possuir um professor orientador
   - Todos possuem um login no formato `<login>@uniriotec.br`
   - Toda disciplina possui um professor responsável

   Funcionalidades do sistema (funções na classe principal)
   - Cadastrar um novo aluno, professor ou disciplina
   - Imprimir lista de alunos e seus e-mails
   - Imprimir lista de professores e suas matrículas SIAPE
   - Imprimir lista de aniversários de todas as pessoas, em ordem cronológica
   - Imprimir lista de disciplinas e seus responsáveis

Créditos

- Baseado em slides de Gleison Santos e exercícios de Pedro Moura, Geiza Hamazaki e Edirlei Lima