Reúso de classes

Prof. Ítalo Assis

Ajude a melhorar este material =]

Encontrou um erro? Tem uma sugestão?

Envie e-mail para italo.assis@ufersa.edu.br

Agenda

- Introdução
- Delegação
- Herança
  - A palavra-chave *super*
  - Modificador de acesso *protected*

Introdução

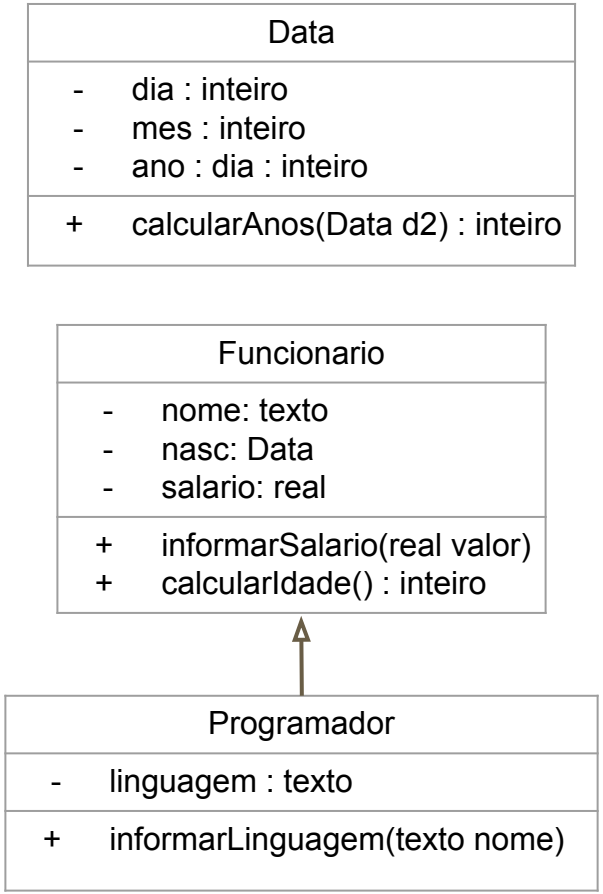
- Uma das características mais interessantes de linguagens de programação orientadas a objetos é a capacidade de facilitar a reutilização de código
  - Diminui a necessidade de escrever novos métodos e classes
- Linguagens de programação orientadas a objetos permitem a criação de classes baseadas em outras
  - Classes criadas com esta técnica poderão conter os métodos das classes originais, além de poder adicionar comportamento específico da nova classe

Introdução

- Dado que exista uma classe *RegistroAcademico* que modela um graduando, como poderíamos modelar um pós-graduando?
  - Suponha que um pós-graduando tem os mesmos atributos e métodos de um graduando, mas também atributos para o título da sua tese e o nome do seu orientador
  - A inclusão dos novos campos à classe *RegistroAcademico* provocaria desperdício de memória
  - Solução orientada a objetos: criar uma classe nova, *RegistroAcademicoPosGraduacao*, que contém os campos e métodos da classe base *RegistroAcademico* e os campos e métodos adicionais
    - O código não é copiado diretamente da classe base para a nova classe

Introdução

- Existem dois mecanismos básicos de reuso de classes em Java: delegação (ou composição) e herança
- Com **delegação**, usamos uma instância da classe base como campo na nova classe
- Com **herança**, criamos a classe nova como uma extensão direta da classe base



Delegação ou Composição

- Podemos criar novas classes que estendem uma outra classe base se incluirmos uma instância da classe base como um dos campos da nova classe, que será então **composta** de campos específicos e de uma instância de uma classe base
- Para que os métodos da classe base possam ser executados, escreveremos métodos correspondentes na classe nova que chamam os da classe base, desta forma **delegando** a execução dos métodos

Data
- dia : inteiro - mes : inteiro - ano : dia : inteiro
+ **calcularAnos**(Data d2) : inteiro

Funcionario
- nome: texto - nasc: **Data** - salario: real
+ informarSalario(real valor) + **calcularIdade**() : inteiro

Exemplo

DataHora.java

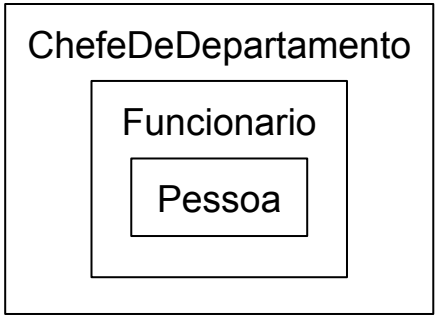
```
public class DataHora {
  private Data estaData;
  private Hora estaHora;

  DataHora(int hora, int minuto, int segundo, int dia, int mes, short ano) {
    estaData = new Data(dia, mes, ano);
    estaHora = new Hora(hora, minuto, segundo);
  }

  DataHora(int dia, int mes, short ano) {
    estaData = new Data(dia, mes, ano);
    estaHora = new Hora(0, 0, 0);
  }

  public String toString() {
    return estaData + " " + estaHora;
  }
}
```

Prática

- Escreva a classe *RegistroAcademico* que contém os atributos *nomeDoAluno*, *matricula* e *curso*.
  - A classe deve ter também seu construtor e um método *toString*
- Depois escreva a classe *RegistroAcademicoPosGraduacao* que reutiliza a classe *RegistroAcademico* via delegação acrestando os atributos *tituloTese* e *orientador*.
- Em uma classe executável, crie algumas instâncias de *RegistroAcademicoPosGraduacao* e as apresente na tela utilizando o método *toString*.

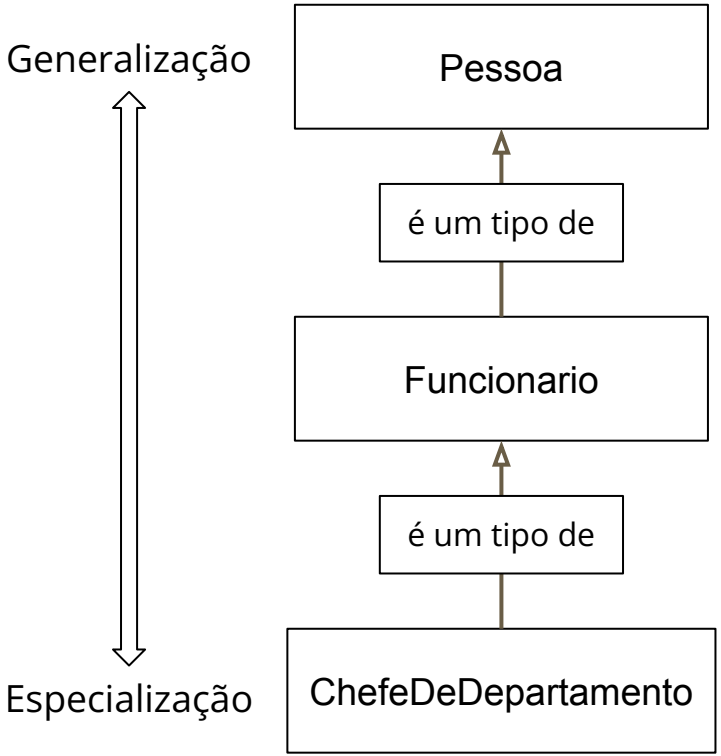
Herança

- Nem sempre o mecanismo de delegação é o mais natural para reutilização de classes já existentes
- Em especial, quando queremos usar uma classe para servir de base à criação de outra mais especializada
- Exemplo: Como poderíamos utilizar a **delegação** para reutilizar código na escrita das classes *Pessoa*, *Funcionario* e *ChefeDeDepartamento*?



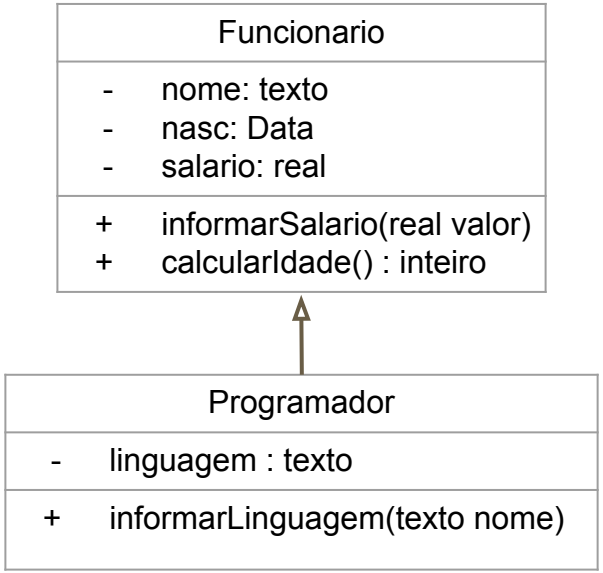
Herança

- Nem sempre o mecanismo de delegação é o mais natural para reutilização de classes já existentes
- Em especial, quando queremos usar uma classe para servir de base à criação de outra mais especializada
- Exemplo: Como poderíamos utilizar a **herança** para reutilizar código na escrita das classes *Pessoa*, *Funcionario* e *ChefeDeDepartamento*?



Herança

- O mecanismo de herança permite que criemos uma classe usando outra como base
- Deve-se descrever na nova classe as diferenças da classe base
- Reutiliza os campos e métodos não-privados da classe base
  - Instâncias da subclasse podem chamar métodos e acessar atributos (herdados) diretamente como se fossem delas mesmas
- É o mais apropriado para criar relações *é-um-tipo-de* entre classes
- Podemos declarar a classe *Programador* como sendo um tipo de *Funcionario*
  - ***class Programador extends Funcionario { ... }***
  - *Programador* herdará todos os campos e métodos da classe *Funcionario*, não sendo necessária a sua redeclaração



A palavra-chave *super*

- Útil para aumentar a reutilização de código
- Refere-se à classe ancestral imediata
  - Similar a palavra-chave *this*
- Permite que classes derivadas tenham acesso a métodos das superclasses
- Construtores são chamados simplesmente pela palavra-chave *super*

```
public class Ponto2D {
    private double x;
    private double y;

    public Ponto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Ponto2D() {
        this.x = 0.0;
        this.y = 0.0;
    }

    public String toString() {
        return "x=" + x + ", y=" + y;
    }
}
```

```
public class Ponto3D extends Ponto2D {
    private double z;

    public Ponto3D(double x, double y, double z) {
        super(x, y);
        this.z = z;
    }

    public Ponto3D() {
        this.z = 0.0;
    }

    public String toString() {
        return super.toString() + ", z=" + z;
    }
}
```

A palavra-chave *super*

- Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses
  - A chamada precisa estar na primeira linha
- O construtor de uma subclasse sempre chama o construtor de sua superclasse, mesmo que a chamada não seja explícita

```
public class Ponto2D {
    private double x;
    private double y;

    public Ponto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Ponto2D() {
        this.x = 0.0;
        this.y = 0.0;
    }

    public String toString() {
        return "x=" + x + ", y=" + y;
    }
}
```

```
public class Ponto3D extends Ponto2D {
    private double z;

    public Ponto3D(double x, double y, double z) {
        super(x, y);
        this.z = z;
    }

    public Ponto3D() {
        this.z = 0.0;
    }

    public String toString() {
        return super.toString() + ", z=" + z;
    }
}
```

A palavra-chave *super*

- Métodos herdados da superclasse podem ser chamados da seguinte maneira:
  - *super.nomeDoMetodo(par1, ..., parN)*
- A declaração de métodos com a mesma assinatura que métodos de classes ancestrais chama-se **sobreposição** ou superposição
- A razão de sobrepormos métodos é que métodos de classes herdeiras geralmente executam tarefas adicionais que os mesmos métodos das classes ancestrais não executam

Prática

- Escreva um projeto com as classes *Pessoa*, *Funcionario* e *ChefeDeDapartamento*
  - *Pessoa* deve ter os atributos *nome* e *identidade* e os métodos *getNome*, *getIdentidade* e *toString*
  - *Funcionario* deve herdar (atributos e métodos) de *Pessoa*, além de ter o atributo *salario* e o método *getSalario*
  - *ChefeDeDapartamento* deve herdar de *Funcionario*, além de ter o atributo *departamento* e o método *getDepartamento*
  - Em uma classe executável, crie uma instância de cada uma das classes criadas e as apresente na tela utilizando o método *toString*

Classe *Object*

- Todas as classes em Java herdam direta ou indiretamente da classe Object (*java.lang.Object*)
- Seus 11 métodos (alguns sobrecarregados) são herdados por todas as outras classes
  - *equals*, *hashCode*, *toString*, *wait*, *notify*, *notifyAll*, *getClass*, *finalize* e *clone*

Modificador de acesso *protected*

- O modificador *protected* funciona como o modificador *private* exceto que classes herdeiras também terão acesso ao campo ou método marcado com este modificador

Prática

- Escreva um projeto com as classes *Automovel*, *AutomovelBasico* e *AutomovelDeLuxo*
  - *Automovel* deve ter os atributos *ano, modelo e cor* e os métodos *quantoCusta* (calcula preço baseado no ano)
  - *AutomovelBasico* deve herdar de *Automovel*, além de ter os atributos airbag e radio. Seu método *quantoCusta* deve somar o valor do seu custo como *Automovel* com o custo dos acessórios básicos
  - *AutomovelDeLuxo* deve herdar de *AutomovelBasico*, além de ter os atributos *arCondicionado* e *direcaoHidraulica*. Seu método *quantoCusta* deve somar o valor do seu custo como *AutomovelBasico* com o custo dos acessórios de luxo. Também deve conter um método *toString* para imprimir todos os seus atributos (inclusive os herdados) e o custo
  - Em uma classe executável, crie uma instância de *AutomovelDeLuxo* e a apresente na tela utilizando os métodos *toString* e *quantoCusta*
  - Use a criatividade para definir as fórmulas empregadas em *quantoCusta*

Os códigos relacionados a esta aula estão disponíveis em

https://github.com/italoaug/Programacao-Orientada-a-Objetos/blob/main/codigos/reuso

Referências

SANTOS, R. **Introdução à programação orientada a objetos usando JAVA**. 2. ed. Rio de Janeiro: Campus, 2013. 336p.