Programação Orientada a Objetos

Herança, polimorfismo, sobrecarga e sobrescrita

Ely – ely.miranda@ifpi.edu.br

1

Herança

- Na vida real:
  - Herança (direito):
    - Transmissão de bens, direitos e obrigações de uma pessoa falecida a seus sucessores legais;
  - Herança genética:
    - Célula ou organismo adquirir características semelhantes a de um que o gerou;
  - Herança em P.O.O:
    - Compartilhar atributos e métodos com o objetivo de reaproveitar código e comportamento;

2

Funcionários e Gerentes

- Uma empresa possui funcionários:

```
public class Funcionario {
    private String nome;
    private String cpf;
    private double salario;
    //construtores e métodos get e set necessários
}
```

- E além disso, funcionários que ocupam cargos de gerente...

3

Funcionários e Gerentes

- Gerentes possuem as mesmas características de funcionários e:
  - Possuem login e senha;
  - Possuem um comportamento de autenticação em um sistema interno;

4

Funcionários e Gerentes

```
public class Gerente {
    private String nome;
    private String cpf;
    private double salario;
    private String login;
    private String senha;
    public boolean autentica(String login, String senha) {
        return (this.login.equals(login) &&
                this.senha.equals(senha)) ;
    }
    //construtores e métodos get e set necessários
}
```


Problema

- Duplicação de código:
  - Duas classes com atributos repetidos;
  - Métodos get, set e construtores também se repetem;
  - Mudanças em um atributo ou método (nome, tipo ou validação) geram alterações em ambas as classes;
  - Havendo outro funcionário com outras características e comportamentos, o código seria triplicado;
  - ... e assim por diante;

Possível solução

- Deixar a classe funcionário mais genérica, com atributos e métodos da classe Gerente;
- Criar um campo lógico em funcionário indicando se o mesmo é um gerente;
- Caso não seja um gerente, deixar os campos em branco (opcionais) e jamais chamar o método autenticar.

Outro problema

- Havendo muitos atributos opcionais, o programador deveria lembrar quais são necessários;
- Ninguém garante que o método autenticar não seria chamado;
- Havendo outro funcionário com outras características e comportamentos:
  - A variável de tipo de funcionário não seria mais lógica e sim outro tipo;
  - O controle de atributos opcionais aumentaria;
  - O controle de métodos que não podem ser chamados aumentaria
  - ... e assim por diante;

Conta x Poupança

- Em um banco, além da conta comum:
  - Temos uma conta poupança;
  - Além de número e saldo, tem também uma taxa de juros;
  - Pode-se gerar um rendimento devido ao saldo acumulado;

ely.miranda@ifpi.edu.br

Conta x Poupança

- A classe Poupança, possui poucas diferenças para a classe Conta, apesar de ser um tipo distinto:

```
public class Poupanca {
    private String numero;
    private double saldo;
    private double taxaJuros;

    public void renderJuros() {
        creditar(saldo*(1 + taxa/100);
    }
    //a partir daqui, tudo igual à classe conta
    public void creditar(double valor) {
        this.saldo += valor;
    }
  //construtores e métodos get e set necessários
}
```

Únicas diferenças

Mesmos problemas que há com as classes Funcionario e Gerente

ely.miranda@ifpi.edu.br

Conta x Poupança: Piorando a situação

- Como ficaria a nossa aplicação do banco?
  - Uma possível solução é ter 2 arrays, um para cada tipo de conta;
  - Duplicar cada método (creditar, incluir, alterar...) para cada tipo de conta;
  - Duplicar a variável de índice dos arrays;

  Novamente: ... e caso apareçam mais N tipos de conta? Conta Imposto, Conta Salário Conta Especial...???

Problemática

- Classe banco com elementos duplicados:

```
public class Banco {
    private Conta[] contas;
    private Poupanca[] poupancas;
    private int indiceContas;
    private int indicePoupancas;

    public void inserirConta(Conta c) {
        contas[indiceContas] = c;
        indiceContas++;
    }

    public void inserirPoupanca(Poupanca p) {
        contas[indicePoupancas] = p;
        indicePoupancas++;
    }
    ...
}
```

Solução

- As classes citadas estão em um mesmo contexto:
  - Gerente é um funcionário;
  - Uma poupança é uma conta;
- Poupanca e Gerente são simples extensões das definições de Conta e Funcionário;
- A partir dessa semelhança podemos usar "herança" e simplificar as implementações;

Herança com Java

- Uma classe pode ser *derivada* de outra e herdar:
  - seu estado (atributos)
  - seu comportamento (métodos);
- Caso um objeto faz o mesmo que outro objeto e "mais alguma coisa":
  - ***reutilizamos*** o código, definindo uma subclasse apenas com diferenças.
- A ***extends*** é utilizada para indicar a herança

```
public class Gerente extends Funcionario { ... }
public class Poupanca extends Conta { ... }
```

Herança

- Classe Gerente reescrita:

```
public class Gerente extends Funcionario {
    private String login, senha;
    //métodos get e set para login e senha
    public boolean autentica(String login,
                             String senha) {
        return (this.login.equals(login) &&
                this.senha.equals(senha)) ;
    }
}
```

Os demais atributos e métodos não precisam ser reescritos, pois são herdados

15

15

Herança

- Classe Poupanca reescrita:

```
public class Poupanca extends Conta {
    private double taxaDeJuros;

    public double getTaxaDeJuros() {
        return taxaDeJuros;
    }
    public void setTaxaDeJuros(double taxaDeJuros) {
        this.taxaDeJuros = taxaDeJuros;
    }

    public void renderJuros() {
        creditar(getSaldo()*(1 + taxaDeJuros/100));
    }

}
```

Os demais atributos e métodos não precisam ser reescritos, pois são herdados

16

16

Herança

- Hierarquia de classes:
  - A mais acima na hierarquia é chamada de super classe ou classe mãe;
  - As que herdam são chamadas de subclasses ou classes filhas;
  - Uma subclasse não tem acesso direto aos membros privados de sua superclasse;
  - **Todo objeto de uma subclasse também é um objeto de sua super classe;**

17

Restrições

- Atributos e métodos privados são herdados, mas não podem ser acessados diretamente;
- Modificador protected: visibilidade restrita a classe e subclasses;
- Construtores não são herdados;
- Construtor padrão só é disponível se também for disponível na superclasse;

18

Usando Poupanca

- Apesar de não definidos, os métodos crédito e débito são herdados visíveis por serem públicos:

```
...
Poupanca  p;
p = new Poupanca();
p.creditar(200);
p.debitar(50);
System.out.println(p.getSaldo());
…
```


Polimorfismo

- É a capacidade de um objeto poder ser referenciado de várias formas;
- Ex:

```
Conta  conta;
conta = new Poupanca();
conta.credito(200);
```

// ou no caso funcionário/gerente:

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
```

Polimorfismo

- Todo objeto de uma subclasse também é um objeto de sua super classe
- Dizemos que uma Poupanca cabe em uma Conta, mas não o contrário

```
...
Conta  conta;
conta = new Poupanca();
conta.credito(200);
conta.debito(50);
System.out.println(conta.getSaldo());
...
```


Casts

- Declarando como uma super classe, deve-se usar casts para acessar elementos específicos da subclasse:

```
...
Conta  conta;
conta = new Poupanca();
...
((Poupanca) conta).renderJuros();
System.out.println(conta.getSaldo());
...
```

Aplicação Banco

- Como Poupanca é do mesmo tipo de Conta, a classe Banco pouco deve ser alterada

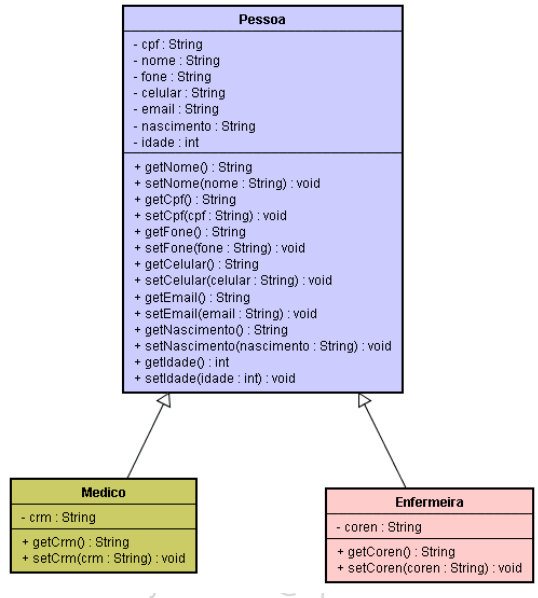
```
...
Banco banco = new Banco();
banco.inserir(new Conta("2134-3"));
banco.inserir(new Poupanca("32990-5"));
banco.credito("2134-3",200);
banco.transfere("2134-3","32990-5",50);
System.out.print(banco.consultarSaldo("32990-5"));
...
```


Verificação de tipos

- Usa-se o operador instanceof para saber se um objeto é de determinado tipo
- Uso indicado para evitar casts que gerem erros

```
...
Conta c = banco.consultar("32990-5");
if (c instanceof Poupanca) {
 ((Poupanca) c).renderJuros();
} else {
  System.out.print(¨Poupança inexistente!¨)
}
...
```

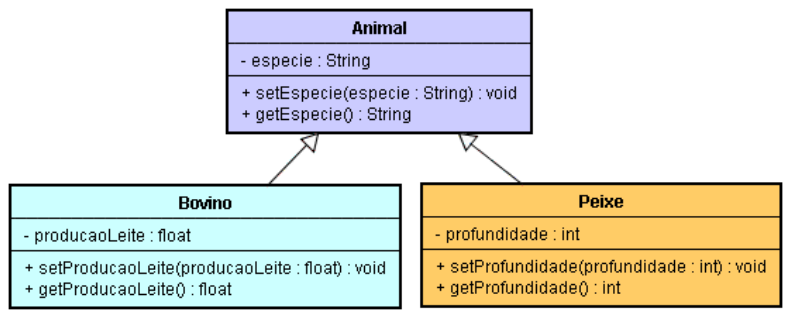
Exemplos de Herança



25

25

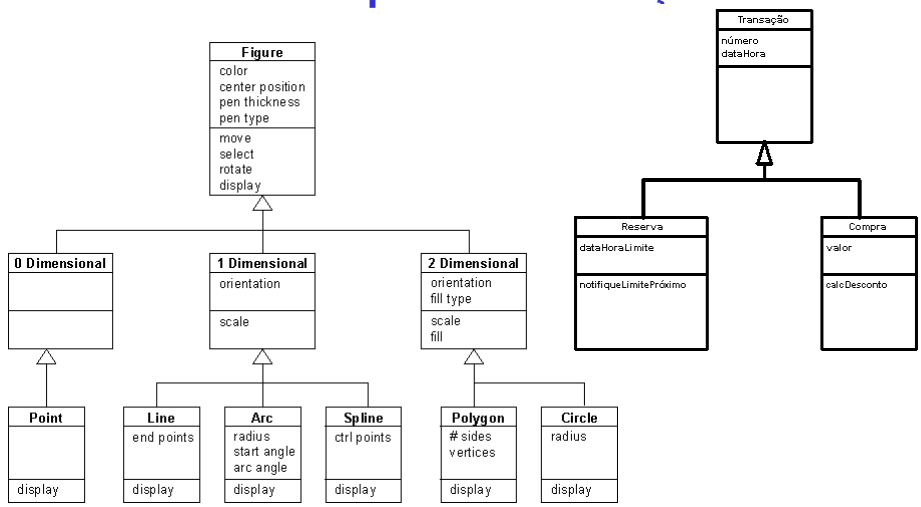
Exemplos de Herança



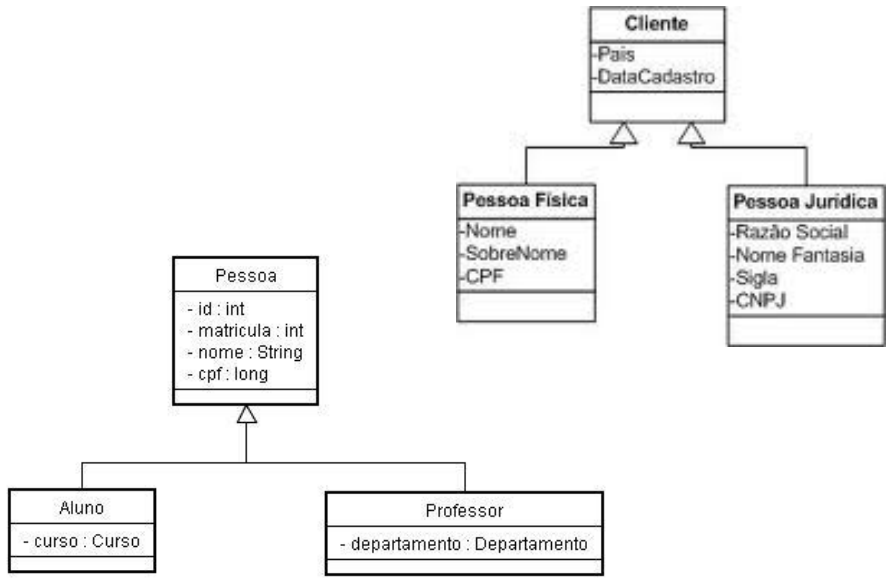
26

26

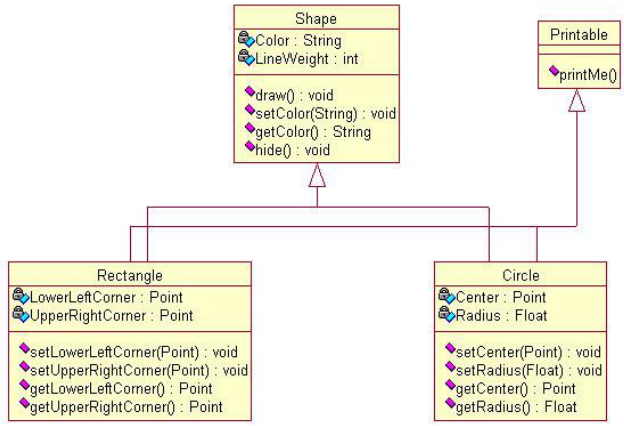
Exemplos de Herança



Exemplos de Herança



Exemplos de Herança



Sobrescrita (override)

- É a redefinição de métodos de uma superclasse em uma subclasse;
- A reescrita de um método sobrepõe a implementação original, mas:
  - O método deve possuir o mesmo nome, tipo de retorno e lista de parâmetros;
- Utilizada quando o comportamento do método da superclasse não corresponde ao desejado para o método da subclasse;
- Pode-se chamar ainda a implementação original se necessário;
- Pode-se evitar que um método sendo sobrescrito usando a palavra **final**;

Sobrescrita

- Supondo uma ContaImposto:
  - Herda de conta e possui um atributo que representa uma % descontada a cada debito
  - O débito da Conta apenas subtrai um valor do saldo
    - Precisa ser sobrescrito na classe ContaImposto para ter o comportamento desejado;

31

Sobrescrita

- Sobrescrevendo/redefinindo o método debitar:

```
public class ContaImposto extends Conta {
    private double taxaDesconto;

    @override
    public void debitar(double valor) {
        saldo -= valor;
        saldo -= valor*getTaxaDesconto()/100;
    }

     // get e set de taxaDesconto
}
```

Mesmo método, só que reescrito

32

Sobrescrita e super

- É possível acessar a definição dos métodos da superclasse imediata
- Usa-se a palavra reservada **super**
- Melhorando o exemplo anterior:

```
public class ContaImposto extends Conta {
    private double taxaDesconto;

    @override
    public void debitar(double valor) {
        super.debitar(valor);
        super.debitar(valor*getTaxaDesconto()/100);
    }
    // get e set de taxaDesconto
}
```

Sobrescrita e acesso à implementação anterior

Sobrecarga (overload)

- Escrever um método ou construtor com o mesmo nome, mas:
  - argumentos diferentes;
  - tipo do retorno pode ser igual ou diferente;
- Usada quando se tem a necessidade de diferentes formas de se chamar um método.

Sobrecarga

- Sobrecarga de construtores:

```
public class Conta {
    //...
    public Conta() {
    }

    public Conta(String numero, String titular) {
        this.numero = numero;
        this.titular = titular;
    }

    public Conta(String numero, String titular,
                 double saldoInicial) {
        this.numero = numero;
        this.titular = titular;
        saldo = saldoInicial;
    }
    //...
}
```


35

35

Sobrecarga

- Sobrecarga de métodos:

```
public class Calculadora {
    public int soma(int op1, int op2) {
        return op1 + op2;
    }

    public double soma(double op1, double op2) {
        return op1 + op2;
    }

    public String soma(String op1, String op2) {
        int op1Int = Integer.parseInt(op1);
        int op2Int = Integer.parseInt(op2);
        return String.valueOf(soma(op1Int, op2Int));
    }
}
```


36

36

Programação Orientada a Objetos

Herança, polimorfismo, sobrecarga e sobrescrita

Ely – ely.miranda@ifpi.edu.br

37