Programação I

Herança

Samuel da Silva Feitosa

Aula 9

Introdução

- Nesta aula estudaremos mais um pilar da Orientação a Objetos.
  - Como modelar um sistema para reaproveitar código utilizando Herança.
  - Uso dos modificadores de visibilidade.
  - Utilização de Construtores de forma implícita e explícita.

Herança

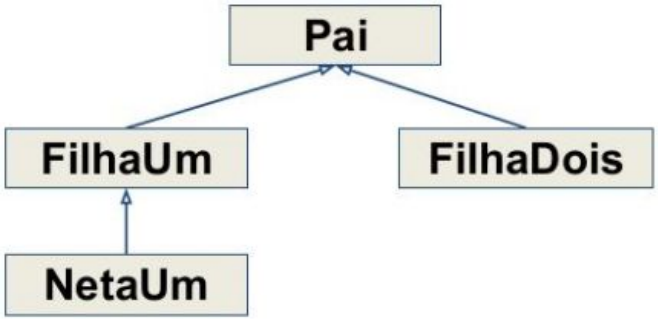
- Terceiro pilar da Orientação a Objetos.
- É um mecanismo da Orientação a Objetos que possibilita a uma classe usar campos e métodos definidos em outra classe.
  - Compartilhamento de membros entre classes.
  - Relação hierárquica, onde uma classe **pai/mãe** empresta suas definições para as classes definidas como **filhas**.
- O Java suporta apenas **herança simples**.
  - Uma única classe pode ser usada como base.

Exemplo básico

- Emprego básico de Herança

```
// superclasse, classe base ou classe pai
public class Pai { }
// subclasses, classes derivadas ou classes-filha
public class FilhaUm extends Pai { }
public class FilhaDois extends Pai { }
public class NetaUm extends FilhaUm { }
```

- Diagrama UML



Modificadores de Visibilidade

- Por meio de Herança, a **subclasse** conta com:
  - Os membros públicos e protegidos da **superclasse**.
  - Pode adicionar membros ou substituir existentes.

	Acessibilidade de membros			
Especificador	**Implementação Superclasse**	**Instâncias Superclasse**	**Implementação Subclasse**	**Instâncias Subclasse**
`private`	sim	não	não	não
`protected`	sim	não	sim	não
`public`	sim	sim	sim	sim

Reutilização de Código (1)

- Uma das principais vantagens de se utilizar Herança é o **reaproveitamento de código**.
- Por exemplo:
  - Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes.
  - Quando um serviço é contratado, o sistema do banco deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data de contratação.

Reutilização de Código (2)

- Com o intuito de ser produtivo, a modelagem dos serviços do banco deve **diminuir a repetição de código**.
  - A ideia é reaproveitar o máximo do código já criado. Essa ideia está diretamente relacionada ao conceito **Don’t Repeat Yourself**.
  - Em outras palavras, devemos minimizar ao máximo a utilização do “copiar e colar”.
- A seguir vamos discutir **algumas modelagens possíveis** para os serviços do banco.

Serviços em Classe única (1)

- Podemos usar uma única classe para modelar todos os tipos de serviço que o banco oferece.
  - Empréstimo: quando um cliente contrata esse serviço, são definidos o **valor** e a **taxa de juros**, gerando dois novos atributos para a Classe Serviço.
  - Seguro de veículo: para este serviço são definidos o **veículo**, o **valor do seguro** e a **franquia**, gerando três novos atributos para a Classe Serviço.

Serviços em Classe única (2)

- Classe Serviço contendo todos os serviços disponíveis.

```
public class Servico {
    // Informações básicas
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataContratacao;

    // Empréstimo
    private double valor;
    private double taxa;

    // Seguro de Veículo
    private String veiculo;
    private double valorSeguro;
    private double franquia;
}
```

- Ao criar um objeto dessa classe, algumas informações ficarão “em branco”, porém ocupando memória.

Uma Classe para cada serviço

- Podemos separar os serviços, cada um com sua própria Classe.

```java
public class Emprestimo {
    // Informações básicas
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataContratacao;

    // Empréstimo
    private double valor;
    private double taxa;
}
```

```java
public class SeguroVeiculo {
    // Informações básicas
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataContratacao;

    // Seguro de Veículo
    private String veiculo;
    private double valorSeguro;
    private double franquia;
}
```

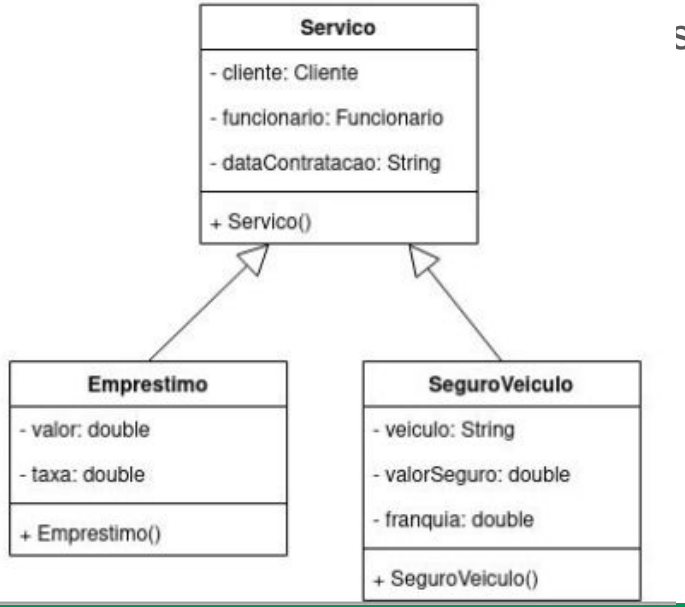
- Repetição de código.
- Qualquer alteração nas informações básicas tem que ser feita nas duas classes.

Usando Herança (1)

- Na modelagem dos serviços do banco, podemos aplicar o conceito de **Herança**.
  - A ideia é reutilizar o código de uma determinada Classe em outras Classes.
- Aplicando Herança, teríamos:
  - Uma classe **base** ou **superclasse**, para representar as informações básicas dos Serviços.
  - Algumas classes **filhas** ou **subclasses**, para representar cada Serviço individualmente.

Usando Herança (2)

- As classes específicas seriam “ligadas” de alguma forma à classe Serviço para reaproveitar o código nela definido.
  - Diagrama UML que ste relacionamento.



Usando Herança (3)

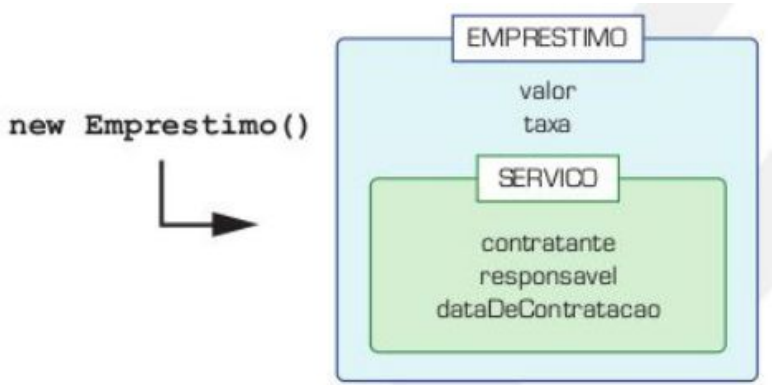
- As três classes definidas usando Herança.

```java
public class Servico {
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataContratacao;
}

public class Emprestimo extends Servico {
    private double valor;
    private double taxa;
}

public class SeguroVeiculo extends Servico {
    private String veiculo;
    private double valorSeguro;
    private double franquia;
}
```

- Sem repetição de código.
- Palavra-chave **extends** para representar a relação.

Usando Herança (4)

- Relembrando:
  - A classe genérica é denominada **superclasse**.
  - As classes específicas são chamadas **subclasses**.
- Quando o operador **new** é aplicado:
  - O objeto construído terá os atributos e métodos definidos na subclasse e na superclasse.



Construtores e Herança (1)

- Quando temos uma hierarquia de Classes, as chamadas dos construtores são mais complexas que o normal.
  - Pelo menos um construtor de cada classe deve ser chamado ao instanciar o objeto.
  - Quando instanciamos a classe Empréstimo, são realizadas chamadas para o construtor de Empréstimo e de Serviço.

Construtores e Herança (2)

- Podemos utilizar a chamada de construtores de forma explícita.
  - Para isso, usamos a palavra-chave **super**.

```
public class Servico {
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataContratacao;

    public Servico(String dataContratacao) {
        this.dataContratacao = dataContratacao;
    }
}

public class Emprestimo extends Servico {
    private double valor;
    private double taxa;

    public Emprestimo(String data, double valor, double taxa) {
        super(data);
        this.valor = valor;
        this.taxa = taxa;
    }
}
```

Considerações Finais

- Herança é um mecanismo que permite a reutilização de código.
  - Evita repetições e fornece maior produtividade a equipe de desenvolvimento.
  - A palavra-chave **extends** é utilizada para representar o relacionamento de Herança.
  - A palavra-chave **super** permite chamar construtores das superclasses de forma explícita.

Exercício Rápido

- Imagine um domínio de aplicação no qual pode ser interessante aplicar Herança.
  - Defina a classe base e as suas subclasses.
  - Desenhe o diagrama de classes que represente esta relação e os atributos e métodos que são reutilizados com o uso de Herança.