



Reusabilidade em Orientação a Objetos

Programação Orientada a Objetos

Tércio de Moraes

¹Ciência da Computação
Campus Arapiraca
Universidade Federal de Alagoas – UFAL

Arapiraca, 16 de agosto de 2024





Objetivos

- Entender as diferentes formas de reusabilidade em Orientação a Objetos
- Conhecer o conceito de **herança**, **polimorfismo** e **delegação** (agregação e composição), bem como suas implicações
- Entender a aplicação do **relacionamento de herança**, **agregação** e **composição** entre classes e **polimorfismo**
- Praticar o uso destes recursos



Agenda

- 1 Fatos Básicos
 - Reusabilidade em OO
- 2 Herança
 - Modificadores de Herança
 - Regras Básicas de Modelagem
 - Herança em Java
 - Exemplo prático
 - Exercícios
- 3 Abstração
 - Abstração na herança
 - Abstração em Java
 - Classe abstrata
- 4 Herança múltipla
 - Definição
 - Interface
 - Praticando
- 5 Polimorfismo
 - Introdução
 - Tipos de polimorfismo
 - Sobrecarga – *Overloading*
 - Substituição – *Overriding*
- 6 Delegação
 - Introdução
 - Composição
 - Agregação



Agenda

- 1 Fatos Básicos
 - Reusabilidade em OO
- 2 Herança
- 3 Abstração
- 4 Herança múltipla
- 5 Polimorfismo
- 6 Delegação



Fatos Básicos

Reusabilidade em Orientação a Objetos



Definição

É a habilidade de **reaproveitar código**, podendo usá-los em outras implementações, estender ou redefinir suas funcionalidades sem a necessidade de reescrevê-los por completo.



Fatos Básicos

Reusabilidade em Orientação a Objetos



Definição

É a habilidade de **reaproveitar código**, podendo usá-los em outras implementações, estender ou redefinir suas funcionalidades sem a necessidade de reescrevê-los por completo. A programação Orientada a Objetos tem como uma de suas metas **maximizar a reusabilidade do código**.



Fatos Básicos

Reusabilidade em Orientação a Objetos



Definição

É a habilidade de **reaproveitar código**, podendo usá-los em outras implementações, estender ou redefinir suas funcionalidades sem a necessidade de reescrevê-los por completo.

A programação Orientada a Objetos tem como uma de suas metas **maximizar a reusabilidade do código**.

Conceitos de **Herança**, **Polimorfismo** e **Delegação** (agregação e composição) tem como objetivo a reutilização.



Agenda

1 Fatos Básicos

2 Herança

- Modificadores de Herança
- Regras Básicas de Modelagem
- Herança em Java
- Exemplo prático
- Exercícios

3 Abstração

4 Herança múltipla

5 Polimorfismo

6 Delegação



Herança

O que é

Classes diferentes podem partilhar características em comum. Herança é o mecanismo que permite que duas ou mais classes distintas partilhem tais características.



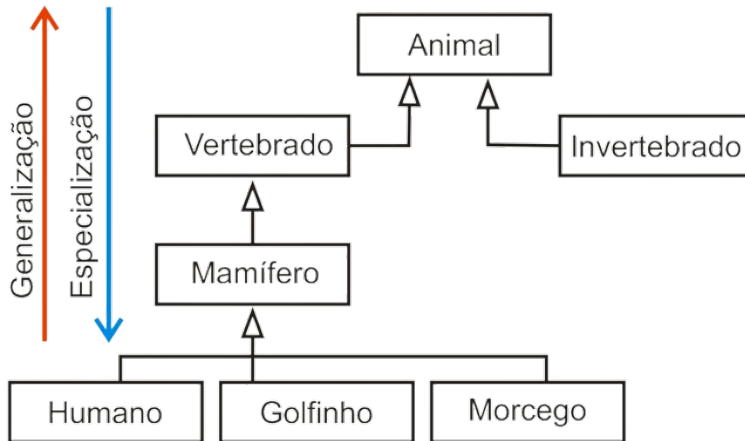
Herança

Como é

- Na herança, as características comuns devem ser agrupadas em uma **classe-base**. Ela será a referência para outras classes que desejam reusar o código;
- É possível criar outras classes a partir da classe-base;
- As classes que reutilizarão o código devem ser redefinidas como uma **extensão** da classe-base;
- A relação de herança envolve duas partes: a **classe-base** e a **classe-herdeira**;
 - **classe mãe** e **classe filha**
 - **superclasse** e **subclasse**
 - **classe base** e **classe derivada**
- Na classe herdeira, o desenvolvedor implementará apenas as características específicas de daquela classe.



Herança - Exemplo

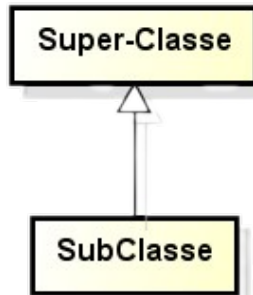




Herança

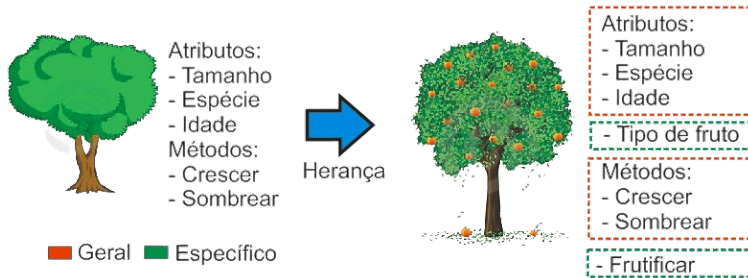
Representação gráfica

A relação de herança entre duas classes é representada por uma linha com um triângulo no lado da super-classe.





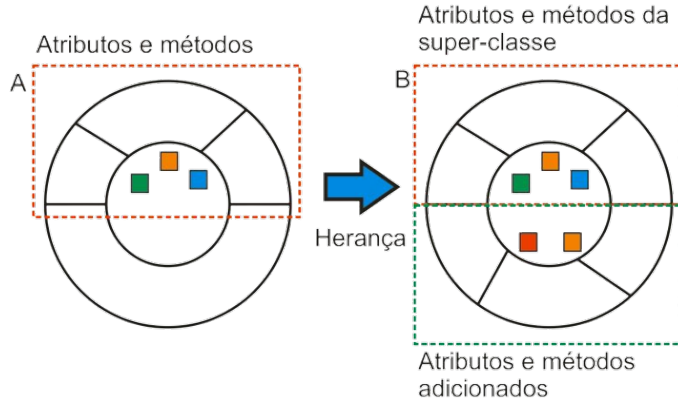
Analogia de herança





Herança

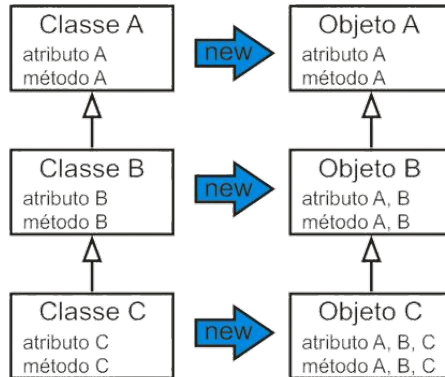
Olhando dentro dos objetos ...





Herança

Olhando os objetos da classe herdada, estes terão atributos e métodos pertencentes a sua classe e de todas as super-classes.





Modificadores da Herança

➤ Podemos usar modificadores para caracterizar alguns aspectos de herança em nosso código

Final Uma classe *final* não permite que seu código seja herdado

Abstrata Uma classe *abstrata* não permite que objetos sejam instanciados a partir dela.

Neste caso, seu código é exclusivo para herança

Protegida Quando usado em métodos e atributos, permite *acesso exclusivo* apenas de suas subclasses

Privado Elementos de uma classe com o modificador *private* não são herdados por subclasses



Regras Básicas de Modelagem

Características comuns

- ➡ *Procure entidades que possuam atributos e comportamento em comum.*
- ➡ *Projete uma classe que represente o estado e comportamentos comuns.*
- ➡ *Para tornar clara a relação de herança, a seguinte afirmação deve fazer sentido:*
Uma subclasse É UMA classe superclasse
- ➡ *Mas o contrário não poderá ser afirmado:*
Uma superclasse É UMA subclasse



Regras Básicas de Modelagem

Características comuns

- *Procure entidades que possuam atributos e comportamento em comum.*
- *Projete uma classe que represente o estado e comportamentos comuns.*
- *Para tornar clara a relação de herança, a seguinte afirmação deve fazer sentido:*
Uma subclasse É UMA classe superclasse
- *Mas o contrário não poderá ser afirmado:*
Uma superclasse É UMA subclasse



Regras Básicas de Modelagem





Características comuns

- *Procure entidades que possuam atributos e comportamento em comum.*
- *Projete uma classe que represente o estado e comportamentos comuns.*
- *Para tornar clara a relação de herança, a seguinte afirmação deve fazer sentido:*
Uma subclasse É UMA classe superclasse
- *Mas o contrário não poderá ser afirmado:*
Uma superclasse É UMA subclasse



Regras Básicas de Modelagem

Características comuns

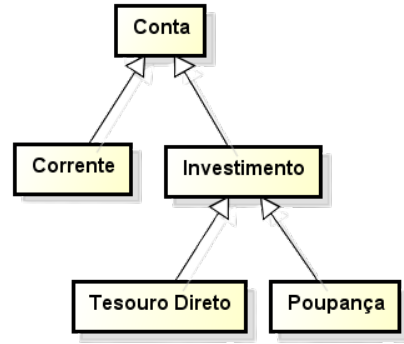
-  *Procure entidades que possuam atributos e comportamento em comum.*
-  *Projete uma classe que represente o estado e comportamentos comuns.*
-  *Para tornar clara a relação de herança, a seguinte afirmação deve fazer sentido:
Uma subclasse É UMA classe superclasse*
-  *Mas o contrário não poderá ser afirmado:
Uma superclasse É UMA subclasse*



Exemplo de Conta Corrente

Exemplo

Vamos dar uma olhada na árvore de herança ao lado:



powered by Astah

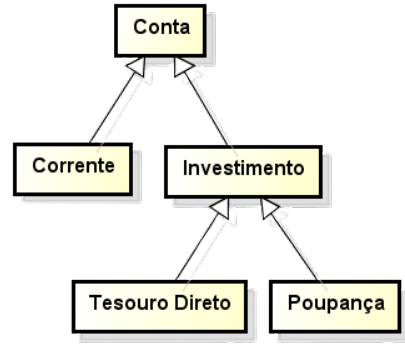


Exemplo de Conta Corrente

Exemplo

Vamos dar uma olhada na árvore de herança ao lado:

- **Corrente** é uma **Conta**?
- **Investimento** é uma **Conta**?
- **Poupança** é uma **Conta**?
- **Conta** é uma **Corrente**?



powered by Astah



Herança em Java

Sintaxe

```
[modificador] class SubClassName extends SuperClassname{  
    ...  
}
```

Exemplos da conta bancária

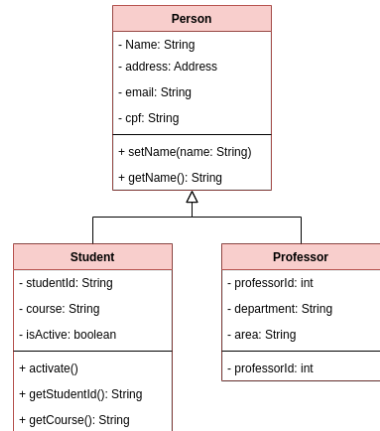
```
public class Corrente extends Conta  
  
public class Investimento extends Conta  
  
public class TesouroDireto extends Investimento  
  
public class Poupanca extends Investimento
```



Exemplo na Prática

Pessoas em um sistema acadêmico

- Observe a **relação de herança** entre as classes do diagrama ao lado
- Observe que boa parte da implementação das classes filha foi **reutilizada através da herança**
- Agora, vejamos isso no código





Exemplo na Prática

Implementação da Herança em Java



```
1 public class Person {  
2     private String name;  
3     private String address;  
4     private String email;  
5     private String cpf;  
6     ...  
7     public String getEmail() {  
8         return email;  
9     }  
10    public void setEmail(String email  
    ) {  
11        if(email.contains("@"))  
12            this.email = email;  
13    }  
14 }
```



Exemplo na Prática

Implementação da Herança em Java



```
1 public class Person {
2     private String name;
3     private String address;
4     private String email;
5     private String cpf;
6     ...
7     public String getEmail() {
8         return email;
9     }
10    public void setEmail(String email
11    ) {
12        if(email.contains("@"))
13            this.email = email;
14    }
```

```
1 public class Student extends Person
2 {
3     private String studentId;
4     private String course;
5     private boolean isActive;
6     ...
7     public boolean isActive() {
8         return isActive;
9     }
10    public void setActive(boolean
11    isActive) {
12        this.isActive = isActive;
13    }
14 }
```



Exemplo na Prática

Programa Principal: utilizando a herança



```
1 public class Academico {
2     public static void main(String[] args) {
3         Student     estudante     = new Student();
4         estudante.setName("Joao Silva");
5         estudante.setAddress("Av. Perpendicular");
6         estudante.setActive(true);
7         estudante.setCourse("Computacao");
8         System.out.println("Dados academicos registrados");
9         System.out.println("    nome: " + estudante.getName());
10        System.out.println("    endereco: " + estudante.getAddress());
11        System.out.println("    curso: " + estudante.getCourse());
12        String status = estudante.isActive() ? "Aluno matriculado" : "Aluno
    sem matricula";
13        System.out.println("    observacao: " + status);
14    }
15 }
```



Exercícios de Herança

Objetos geométricos

- Considere que todos objetos geométricos **são** uma **formas**.
- Toda **forma geométrica** tem uma **área** e uma **cor**
- Crie três classes que estão relacionada entre si pela herança:
 - Forma
 - Retângulo
 - Círculo



Agenda

1 Fatos Básicos

2 Herança

- 3 Abstração
- Abstração na herança
 - Abstração em Java
 - Classe abstrata

4 Herança múltipla

5 Polimorfismo

6 Delegação



Classes Abstratas

Relembrando **abstração**

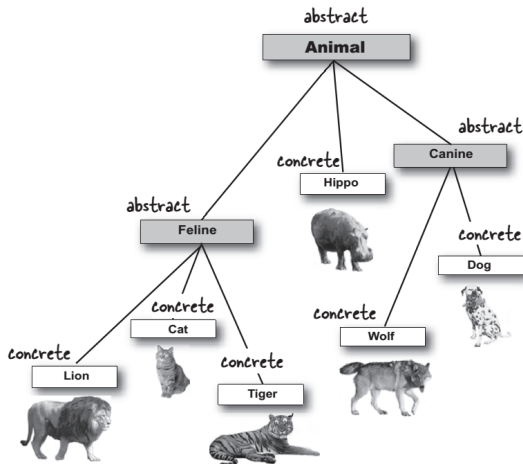
- ✎ Exercício intelectual de **seleção** de alguns aspectos de **domínio do problema**, desconsiderando outros aspectos não interessantes para o problema em questão
- ✎ Habilidade de concentrar-se nos **aspectos essenciais** de um contexto qualquer, ignorando características menos importantes ou acidentais

Voltando ao exemplo do sistema académico ...

Quais instâncias das classes implementadas anteriormente existirão no académico?



Classes Abstratas

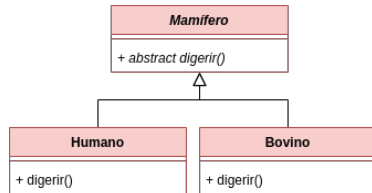




Classes Abstratas na Herança

Quando as “coisas” ainda não estão definidas

- ➡ A **abstração** permite **declarar** comportamentos que ainda **ainda não estão definidos**
- ➡ Por exemplo
 - ➡ Todo mamífero **digere o alimento**
 - ➡ Porém, a **forma de digestão** é diferente
 - ➡ Humanos são **monogástricos**
 - ➡ Hervíboros são **ruminantes**
- ➡ **Não é possível “implementar”** a função **digerir** para o mamífero
- ➡ **PORÉM**, é necessário **“obrigar a implementação”** em todos aqueles que **são mamíferos**

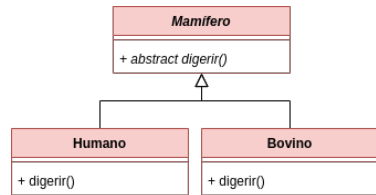




Classes Abstratas na Herança

Quando as “coisas” ainda não estão definidas

- A **abstração** permite **declarar** comportamentos que ainda **ainda não estão definidos**
- Por exemplo
 - Todo mamífero **digere o alimento**
 - Porém, a **forma de digestão** é diferente
 - Humanos são **monogástricos**
 - Hervíboros são **ruminantes**
- **Não é possível “implementar”** a função **digerir** para o mamífero
- **PORÉM**, é necessário **“obrigar a implementação”** em todos aqueles que **são mamíferos**



Atenção

Classes abstratas **não**
constroem objetos



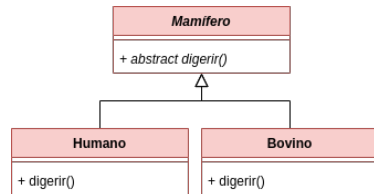
ATENÇÃO



Classes Abstratas na Herança

Quando as “coisas” ainda não estão definidas

- A **abstração** permite **declarar** comportamentos que ainda **ainda não estão definidos**
- Por exemplo
 - Todo mamífero **digere o alimento**
 - Porém, a **forma de digestão** é diferente
 - Humanos são **monogástricos**
 - Hervíboros são **ruminantes**
- **Não é possível “implementar”** a função **digerir** para o mamífero
- **PORÉM**, é necessário **“obrigar a implementação”** em todos aqueles que **são mamíferos**



Atenção

Classes abstratas **não constroem** objetos



*Em nosso sistema acadêmico,
qual classe seria abstrata?*



Abstração em Java

Formas de abstração

- ✎ **Parcial** – alguns **métodos** da classe podem ser **declarados** sem precisar de **implementação** – ***abstract class***
- ✎ **Total** – a classe é 100% **abstrata** – ***interface***

A abstração dentro da classe

A **abstração de dados** também pode ser definida como o processo de identificar **apenas as características necessárias** de um objeto, **ignorando os detalhes irrelevantes**

Implementação

- ✎ Classes abstratas
- ✎ Interfaces



Classe Abstrata

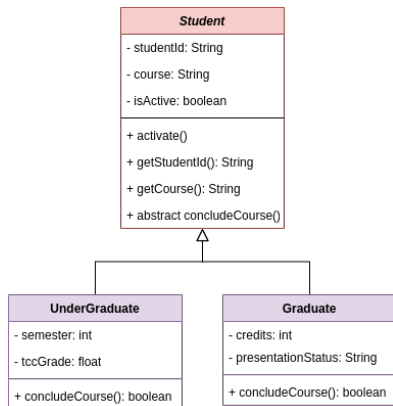
Características de uma classe abstrata

- Classe declarada como **abstract class**
- Composta de, **pelo menos um**, método abstrato
- Pode ser composta por **métodos concretos** (com implementação)
- **Não há objetos** de uma classe abstrata
- **Métodos abstratos**, obrigatoriamente, são **implementados** em **classes filhas**
- Se o método **não for implemetado**, a subclasse também **deverá ser uma classe abstrata**
- **Construtores podem ser implementados** em classes abstratas para serem **reusados** pelas classes filhas



Classe Abstrata

Exemplo de Implementação



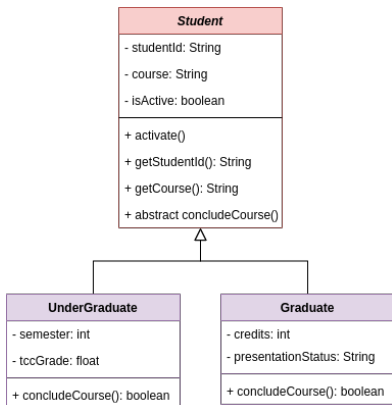
Classe Student

```
1 public abstract class Student extends Person{
2     private String studentId;
3     private String course;
4     private boolean isActive;
5     ...
6     public abstract boolean concludeCourse();
7 }
```



Classe Abstrata

Exemplo de Implementação



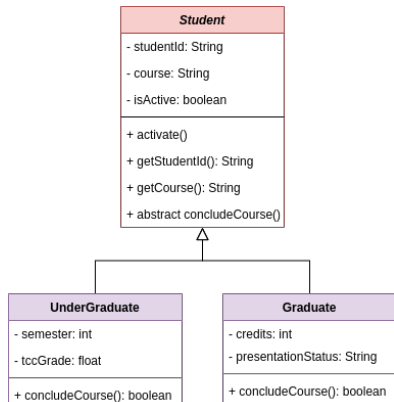
Classe UnderGraduate

```
1 public class UnderGraduate extends Student {
2     static final int TOTAL_SEMESTERS = 8;
3     private int semester;
4     private float tccGrade;
5     ...
6     public boolean concludeCourse() {
7         if (this.getSemester() == TOTAL_SEMESTERS
8             && this.getTccGrade() > 7.0){
9             return true;
10        } else{
11            return false;
12        }
13    }
14 }
```



Classe Abstrata

Exemplo de Implementação



Classe Graduate

```
1 public class Graduate extends Student{
2     static final int MIN_CREDITS = 24;
3     private int credits;
4     private String defenseStatus;
5     ...
6     public boolean concludeCourse() {
7         if(this.credits >= MIN_CREDITS
8             && this.getDefenseStatus().equals("
APROVED")){
9             return true;
10        } else{
11            return false;
12        }
13    }
14 }
```



Agenda

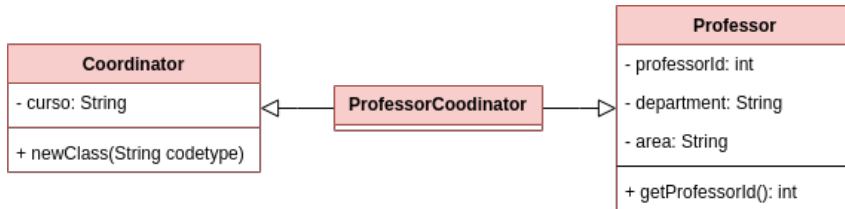
- 1 Fatos Básicos
- 2 Herança
- 3 Abstração
- 4 Herança múltipla
 - Definição
 - Interface
 - Praticando
- 5 Polimorfismo
- 6 Delegação



Herança Múltipla

Definição

- Uma **subclasse** pode **herdar características** de **várias classes**
- Uma pessoa pode ser **professor** e **servidor** na mesma instituição





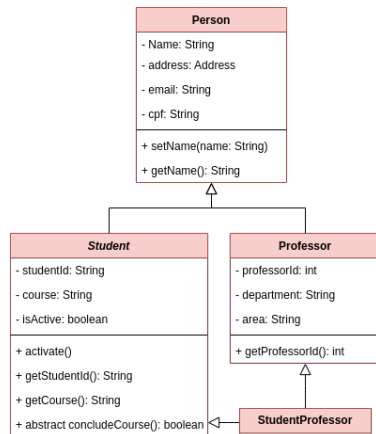
Herança Múltipla

Problemas

Problemas

- **Problema do diamante**
 - **Classes mãe com herança em comum**
 - **Construtores** baseados na superclasse em comum são **chamados 2x**
- **Assinaturas indênticas – Ambiguidade**
 - Classes mãe com **métodos iguais**, qual será executado?

Java NÃO implementa herança múltipla!





Interface em Java

Definição

- Especificação de **comportamento** de uma classe: **projeto de comportamento**
- Mecanismo para **obter 100% de abstração**

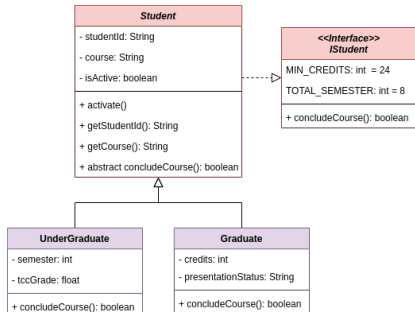
Característica de uma interface

- Especifica o que uma classe **deve fazer e não como**
- Uma classe **implementa** uma interface
- Se a classe **não implementa**, ela deve ser declarada **abstrata**
- Interfaces podem ter relação de **herança** entre elas
- **Não é** exatamente uma **relação de herança**
- Interface é sobre **capacidades** que uma classe deve ter



Interface em Java

Exemplo de Implementação



```
1 public interface IStudent {
2     final int MIN_CREDITS      = 24;
3     final int TOTAL_SEMESTERS = 8;
4
5     public String getStudentId();
6     public void setStudentId(String studentId
7         ) ;
8     public String getCourse();
9     public void setCourse(String course);
10    public boolean isActive();
11    public abstract boolean concludeCourse();
12 }
```



Mãos na massa

Abstrações geométricas

- Considere o problema dos **objetos geométricos** visto anteriormente
- Use **classe abstrata** para implementar a classe **Forma**

Usando interface

- Resolva o mesmo problema usando **interface**
- Observe que ambas as classes (Retângulo e Círculo) implementarão a interface **Forma**





Agenda

1 Fatos Básicos

2 Herança

3 Abstração

4 Herança múltipla

5 Polimorfismo

- Introdução
- Tipos de polimorfismo
- Sobrecarga – *Overloading*
- Substituição – *Overriding*

6 Delegação



Polimorfismo

Introdução

O teste da herança

➤ Em nosso exemplo do sistema Acadêmico podemos fazer o teste:

➤ Graduate **É-UM** Graduate?

➤ Graduate **É-UM** Student?

➤ Graduate **É-UM** Person?

➤ Um **objeto** Graduate pode assumir **diferentes formas**



Polimorfismo

Definição

Polimorfismo

- Habilidade de um objeto tomar **várias formas**
- **Poli** = **muitas**, **morfi** = **forma**

Polimorfismo em métodos

- O polimorfismo nos permite realizar uma **única ação de maneiras diferentes**
 - Uma **interface** e **múltiplas** implementações

Benefícios do polimorfismo

- Sistemas facilmente **extensíveis**
- Novas classe podem ser adicionadas com **modificações de ajuste** no código
- Por exemplo, **Peixe**, **Cavalo** e **Gavião** são **Animais** que **se movem** de formas diferentes



Tipos de Polimorfismo

Uso mais comum do polimorfismo

- Referência de **superclasse** é usada para se referir a um objeto de **subclasse**
 - `UnderGraduate ug = new UnderGraduate();`
 - `Student s = new UnderGraduate();`
 - `Person p = new UnderGraduate();`
- Lembrando que, em Java, as **variáveis objeto** apontam para a mesma referência:
`ug, s, p → new UnderGraduate();`

Tipos de polimorfismo


- Em tempo de **compilação** – **sobrecarga** (*overloading*)
- Em tempo de **execução** – **substituição** (*overriding*)



Tipo de Polimorfismo

Sobrecarga

Sobrecarga

 **Múltiplas funções** (métodos) com o mesmo nome, porém com **parâmetros diferentes**

```
1 public class Person {  
2     private String name;  
3     private String cpf;  
4     ...  
5     public Person(){}  
6     public Person(String name, String  
7         cpf){  
8         this.setName(name);  
9         this.setCpf(cpf);  
10 }
```

Person
- Name: String - cpf: String
+ Person() + Person(name:String, cpf:String) + setName(name: String) + getName(): String



Tipos de Polimorfismo

Exemplo Prático de Sobrecarga

➡ Considere a classe **Multiplier** abaixo

```
1 class Multiplier {  
2     static int Multiply(int a, int b)  
3     {  
4         return a * b;  
5     }  
6     static double Multiply(double a,  
7     double b) {  
8         return a * b;  
9     }  
10 }
```

➡ Execute o código abaixo e observe quais **implementações** são **executadas**

```
1 class Main {  
2     public static void main(String[]  
3     args) {  
4         Sysout(Helper.Multiply(2, 4));  
5         Sysout(Helper.Multiply(5.5,  
6         6.3));  
7     }  
8 }
```



Tipos de Polimorfismo

Substituição

Substituição

- Ocorre quando uma **subclasse** tem a **implementação alterada** de um **comportamento herdado**
- Os **métodos herdados e alterados** são ditos **substituídas** ou **sobrepostas** – ***overriden***

Algumas regras

- A **lista de argumentos** e **tipo do retorno** devem ser os mesmos
- Atenção ao **nível de acesso** do método: o método substituído **não pode ter menos acesso** que o o método da superclasse
- Métodos **final** ou **static** não podem ser substituídos
- Este tipo de polimorfismo **não se aplica** a **Construtores**

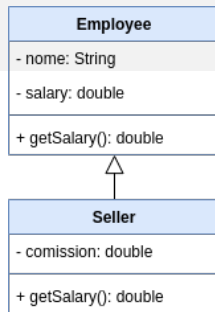


Tipos de Polimorfismo

Exemplo Prático de Substituição

Cálculo de salário

- Considere duas categorias de funcionários:
 - Técnico administrativo
 - Vendedor
- Ambos têm **maneiras distintas** de calcular o salário



```
1 class Employee{
2     private String name;
3     private double baseSalary;
4     public double getSalary(){
5         reuturn this.baseSalary;
6     }
7 }
```

```
1 class Seller extends Employee{
2     private String comission;
3     public double getSalary(){
4         return getBaseSalary
5             + comission;
6     }
7 }
```



Exercício

Objetos geométricos

- Considere que todos objetos geométricos **são** uma **formas** com cálculos de área específicos.
- Crie três classes que estão relacionada entre si pela herança, aplicando polimorfismo aos métodos das classes derivadas:
 - Forma (*o método `double getArea()` deve retornar 0*)
 - Retângulo
 - Círculo





Agenda

1 Fatos Básicos

2 Herança

3 Abstração

4 Herança múltipla

5 Polimorfismo

6 Delegação

- Introdução
- Composição
- Agregação



Delegação

O que é

- A grosso modo, **transferir responsabilidades**
- **Conferir poder e representatividade** a outro

Na Orientação a Objetos

- Dividir responsabilidades de acordo com o papel de cada entidade dentro do sistema
- Orientação a Objetos, por si só, trata-se de **delegação**
- Tipos de delegação
 - **Composição**
 - **Delegação**
- Ambos os tipos são usados para criar **relações entre classes**
- As diferenças são sutis em termos de **propriedades** e **ciclo de vida** dos objetos



Composição

Relação

- Na composição, um objeto é **composto** por outros objetos
- Assim, o **ciclo de vida** dos **objetos componentes** está **vinculado** ao ciclo de vida do **objeto que os contém**
- Quando o “**objeto maior**” é destruído, os **objetos componentes** também são
- Um objeto é **parte exclusiva** de outro objeto

Exemplo simples

- Um **carro** é composto de **vários componentes**
- Assim, o **motor** é um **objeto componente** do carro



Composição

Implementação em Java

Regra de implementação

✏ Para que um objeto seja parte de outro maior, ele deve ser **construído dentro da classe**

```
1 class Engine{  
2     //codigo  
3 }
```

```
1 class Car{  
2     private Engine engine;  
3     public Car(){  
4         this.engine = new Engine();  
5     }  
6 }
```

Observe que o objeto **engine** é construído dentro da classe **Car**



Agregação

Relação

- Na agregação, um **objeto agregado compõe** um objeto maior, mas a **sua existência não depende** do objeto maior
- Os dois objetos relacionados **podem existir independentes** um do outro
- Um **objeto agregado não é destruído** quando o objeto maior é destruído

Exemplo simples


- Turma de alunos é **composta** por alunos, mas pode estar vazia
- Por outro lado, um **aluno** existe independente da **existência de qualquer**



Agregação

Implementação

Regra de implementação

 O **objeto agregado** é **criado fora** do objeto maior e **passado por referência**

```
1 class Student {  
2     private String name;  
3     public Student(String name) {  
4         this.name = name;  
5     }  
6 }
```

*Observe que objetos **Student** são passado como argumento no método **addStudent***

```
1 class Course {  
2     private String title;  
3     private List<Student> students;  
4     public Course(String title) {  
5         students = new ArrayList<>();  
6     }  
7     public void addStudent(Student  
8         student) {  
9         students.add(student);  
10    }  
11 }
```



Agregação

Exercício



Sistema Acadêmico

- Considere nosso exemplo do Sistema Acadêmico
- Na classe **Person**, adicione o atributo **address** que deve ser do tipo classe **Address**
- Decida se esta relação é uma composição ou agregação