INSTITUTO SUPERIOR POLITECNICO DE TECNOLOGIAS E CIENCIAS

1º Semestre

2025/26

ISPTEC

Sumário

- Herança e Sobreposição
  - O que é?
  - Hierquia de classes
  - Subclasses e construtores
  - Overriding de métodos e variáveis
  - Método toString() e equals()
  - Exercícios de Fixação

Herança o que é ?

- Na vida real o que é **herança**?
  - É quando uma pessoa deixa os seus bens para outra
  - Geralmente ocorre entre membros de uma mesma família

- No **Java** também é a mesma a coisa
  - Herança, em Java, nada mais é do que criar classes usando outras classes já existentes.

Sumário

- Herança e Sobreposição
  - O que é?
  - Hierquia de classes
  - Subclasses e construtores
  - Overriding de métodos e variáveis
  - Método toString() e equals()
  - Exercícios de Fixação

Hierquia de classes

- A **herança**, é o processo de criação de uma nova **classe** com as características de uma **classe** existente, juntamente com algumas **características adicionais específicas à nova classe**.

- Quando utilizada correctamente permite a reutilização de código e facilita o desenho de software

- A **classe** que herda as propriedades de outra, chama-se **subclasse**, e a classe que disponibiliza as suas propriedades à subclasse chama-se **superclasse**.

- Poderá ser bastante útil, criar ***superclasses*** que têm como única função servir de modelo **abstracto** para as **subclasses** utilizadas posteriormente. Uma **classe abstracta** não pode ser instanciada, ou seja, não podem ser criados **objectos** a partir de uma classe deste tipo.

- Em Java, todas as **classes**, são ***subclasses*** da ***superclasse*** chamada **Object**.

- A forma de derivar classes de outras classes, em vez da *classe* ***Object***, é a utilização da *palavra chave* **extends**

Hierquia de classes (Cont…)
Exemplo

- Supondo que queremos modelar um stand de veículos onde estão a venda **carros** e **motas**

```
1  public class Carro {
2
3      String motor;
4      double preço;
5      string marca;
6      string nomeCliente;
7
8      boolean temArCondicionado;
9      int numeroDePortas;
10 }
```

```
13 public class Mota {
14
15     String motor;
16     double preço;
17     String marca;
18     String nomeCliente;
19
20     boolean temCapacete;
21 }
```

- Como se percebe tanto na classe **Carro** como na classe **Mota** existem atributos que são comuns ou seja estamos a repetir **atributos**.

- Se pensarmos um bocado chegaremos a conclusão de que tanto um Carro como uma Mota são Veículos certo?
  - Então nesse caso podemos aplicar o conceito de herança e assim reutilizarmos código que é um dos fundamentos da POO

Hierquia de classes (Cont…)
Exemplo

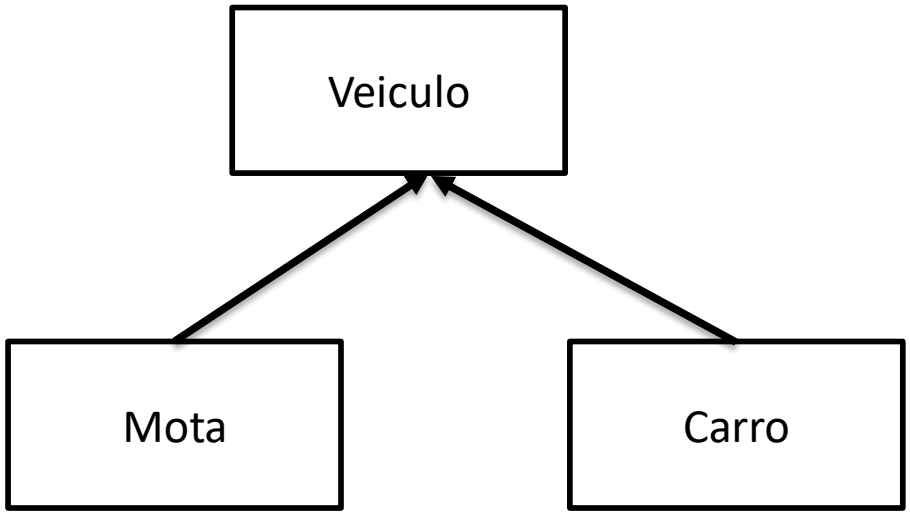


Diagrama de classes da herança

Hierquia de classes (Cont...)
Exemplo

- **Sintaxe de declaração da Herança**

```
public class Filha extends Pai {



}
```

Hierquia de classes (Cont...)
Exemplo

- Para resolver o problema iremos manter as duas classes **Mota** e **Carro**. Mas estas irão herdar de uma **Veículo** uma vez que tanto um carro como uma mota são ambos veículos

```
public class Veiculo{

    String motor;
    double preço;
    String marca;
    String nomeCliente;
}
```

```
public class Carro extends Veiculo {

    boolean temArCondicionado;
    int numeroDePortas;
}
```

```
public class Mota extends Veiculo {

    boolean temCapacete;
}
```

- Nesse caso dizemos que **Veiculo** é a **superclasse**, **Mota** e **Carro** são subclasses.

- Herança e Sobreposição
  - O que é?
  - Hierquia de classes
  - Subclasses e construtores
  - Overriding de métodos e variáveis
  - Método toString() e equals()
  - Exercícios de Fixação

- Herança: o construtor da **Subclasse** invoca sempre o construtor da **Superclasse** em primeiro lugar.

```
public class Heranca {

    public static void main(String[] args) {
        new Filha();
    }
}

public class Pai {

    public Pai(){
        System.out.println("Construtor da classe Pai");
    }
}

public class Filha extends Pai {

    public Filha(){
        System.out.println("Construtor da classe Filha");
    }
}
```

- Qual o resultado?

Construtor da classe Pai
Construtor da classe Filha

Subclasses e Construtores


```java
public class Veiculo{

    private String marca;

    public Veiculo(String marca){
        this.setMarca(marca);
    }

    public String getMarca(){
        return marca;
    }

    public void setMarca(String marca){
        this.marca = marca
    }

}
```

Superclasse

Construtor da Superclasse

```java
public class Carro extends Veiculo {

    boolean temArCondicionado;
    int numeroDePortas;

    public Carro(String marca, boolean temArCondicionado, int numeroDePortas){
                super(marca)

                this.temArCondicionado = temArCondicionado;
                this.numeroDePortas = numeroDePortas;
    }
}
```

Subclasse

Construtor da Superclasse

- Para invocarmos o constructor de uma **superclasse** numa **subclassse** utilizamos a palavra reservada **super()**

- Herança e Sobreposição
  - O que é?
  - Hierquia de classes
  - Subclasses e construtores
  - **Overriding de métodos e variáveis**
  - Método toString() e equals()
  - Exercícios de Fixação

- Quando uma classe herda comportamento de uma classe mãe, esta pode fazer override a esse comportamento usando a mesma **assinatura do método** identificando explicitamente esse desejo de override com uma anotação @Override.

- Herança e Sobreposição
  - O que é?
  - Hierquia de classes
  - Subclasses e construtores
  - Overriding de métodos e variáveis
  - Método toString() e equals()
  - Exercícios de Fixação

- No Java toda classe tem um ancestral por "padrão": a classe **Object**.
  - Deste modo, qualquer **objeto** de qualquer **classe** que criarmos em Java também será uma instância de **Object**.
  - A classe **Object** já expõe por padrão pelo menos dois métodos:
    - toString()
    - equals()

- O objetivo do método toString() é devolver uma **representação textual** de uma instância de um objeto.

- Essa representação textual de um objeto é muito útil principalmente em situações de **debugging** e de **logging**.
  - Isso ocorre porque os métodos de saída para o streamming padrão (os famosos **System.out.print[ln]()** ou **Console.Write[Line]()),** assim como os principais métodos de praticamente todas as APIs de log (métodos como o **debug**() e **info**()) invocam por padrão o método toString().

O método toString()


```
int numero = 10;
System.out.println(numero);
```

- Qual o resultado?

10

- Quando uma variável de um tipo primitivo é passada como parâmetro para o método **println()** o valor primitivo é impresso na ecrã (saída padrão) porque esse método invoca por padrão o método **toString()** que têm como objectivo devolver uma representação textual.

O método toString()


```
3   public class Conta {
4
5       private double saldo;
6
7       public Conta(double saldo) {
8           this.saldo = saldo;
9       }
10
11      public double getSaldo() {
12          return saldo;
13      }
14
15      public void setSaldo(double saldo) {
16          this.saldo = saldo;
17      }
18  }
19
```

```
Conta conta1 = new Conta(25.000);

System.out.println(conta1.toString());
System.out.println(conta1);
```

- Qual o resultado?

entidades.Conta@15db9742

entidades.Conta@15db9742

- A implementação padrão do **toString()**, na classe **object**, devolve o nome da classe mais específica do objeto.

- Para obtermos uma **representação textual** mais palpável dos objectos de uma classe podemos **reescrever**(override) o **método toString()** padrão da classe **object**

```
3   public class Conta {
4
5       private double saldo;
6
7       public Conta(double saldo) {...3 lines }
10
11      public double getSaldo() {...3 lines }
14
15      public void setSaldo(double saldo) {...3 lines }
18
19      @Override
        public String toString() {
21          return "Saldo: " + this.saldo;
22      }
23  }
```

```
Conta conta1 = new Conta(25.000);

System.out.println(conta1.toString());
System.out.println(conta1);
```

- Qual o resultado?

Saldo: 25.0

Saldo: 25.0

- Podemos sempre claro implementar um método que devolva uma representação textual dos objectos das nossas classes mas podemos evitar isso uma vez que o propósito do método toString() é esse mesmo, temos é apenas de fazer o **override** ao método **toString()** padrão.

Método equals()

- Para verificar se os valores armazenados em duas variáveis de algum tipo primitivo são iguais, deve ser utilizado o operador **"=="** .
  - Esse operador também pode ser aplicado em variáveis de tipos não primitivos.

```
Conta conta1 = new Conta(25.000);
Conta conta2 = new Conta(25.000);

System.out.println(conta1 == conta2);
```

- Qual o resultado?

  false

  - O operador **"=="**, aplicado em referências, verifica se elas apontam para o mesmo objeto na memória.

Método equals()

- Quando for necessário comparar o conteúdo de dois objetos e não as suas localizações na memória, devemos utilizar o método **equals()**

- Esse método está presente em todos os objetos pois ele foi definido na classe **object**.

- A implementação padrão do método **equals()** na classe **object** verifica se os objetos estão localizados no mesmo lugar da memória. Esse método deve ser reescrito caso seja necessário comparar o conteúdo dos objetos.

Método equals()


```
3   public class Conta {
4
5       private double saldo;
6
7       public Conta(double saldo) {...3 lines }
10
11      public double getSaldo() {...3 lines }
14
15      public void setSaldo(double saldo) {...3 lines }
18
19      @Override
        public boolean equals(Object o) {
21          if (o instanceof Conta) {
22              Conta conta = (Conta) o;
23              return this.saldo == conta.saldo;
24          } else {
25              return false;
26          }
27      }
28  }
```

```
Conta conta1 = new Conta(25.000);
Conta conta2 = new Conta(25.000);

System.out.println(conta1.equals(conta2));
```

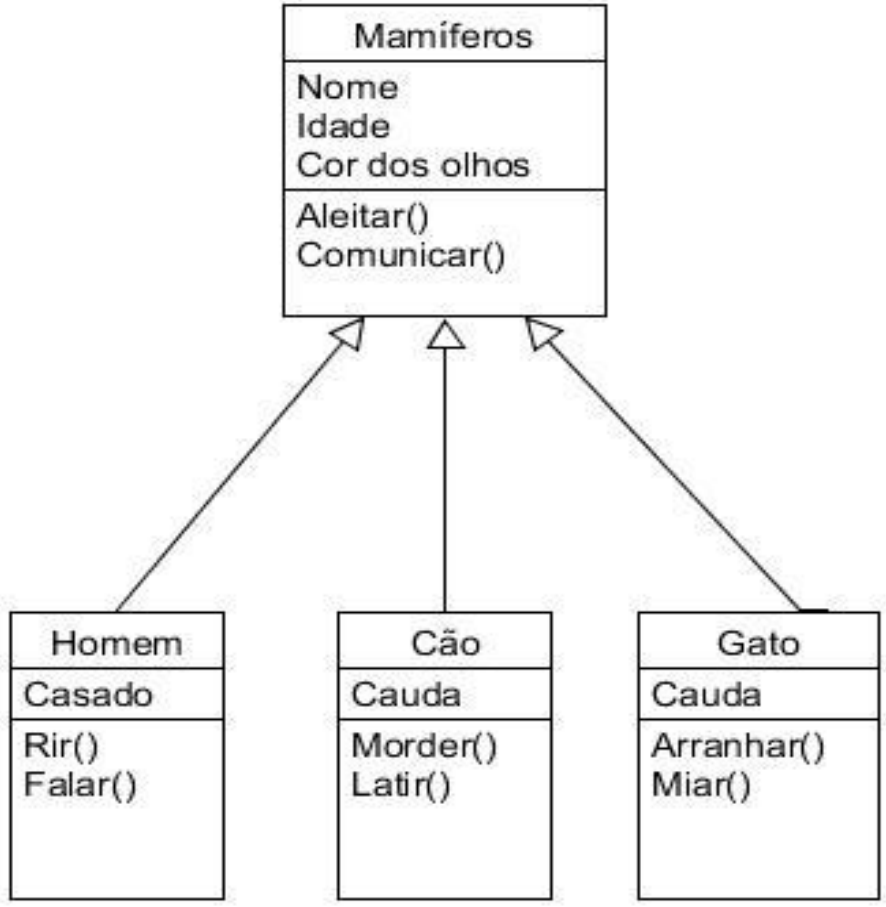
- Qual o resultado?

true

Método equals()

- Para compararmos objectos devemos utilizar o método **equals()** uma vez que o seu propósito é esse mesmo

- Dado o seguinte diagrama de classes faça a sua representação em Java



Dúvidas