



Java

Herança e Polimorfismo

METODOLOGIA

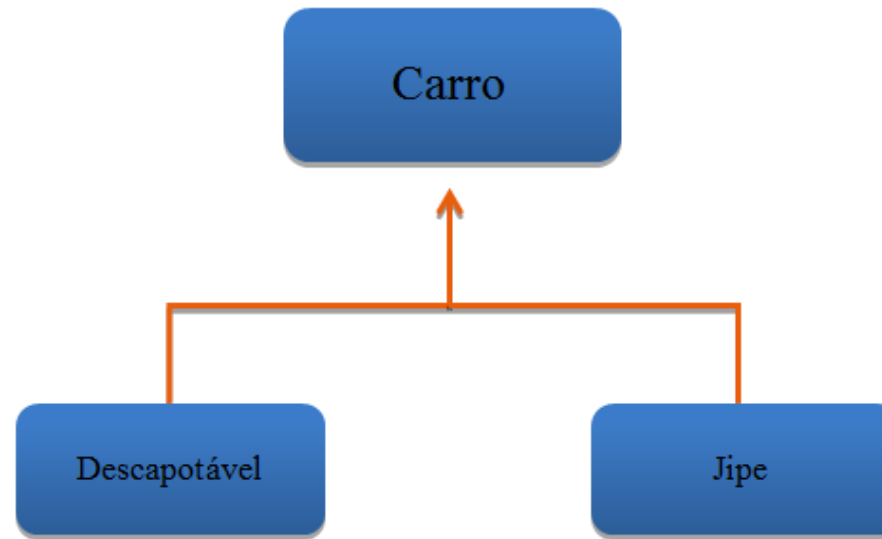
- › Interprete o documento calmamente e com atenção.
- › Acompanhe a execução do exercício no seu computador.
- › Não hesite em consultar o formador para o esclarecimento de qualquer questão.
- › Não prossiga para o ponto seguinte sem ter compreendido totalmente o ponto anterior.
- › Caso seja necessário, execute várias vezes o exercício até ter compreendido totalmente o processo.

CONTEÚDO PROGRAMÁTICO

1. [Herança](#)
2. [Polimorfismo](#)
3. [Classes Abstratas](#)

1. Herança

Até agora temos estado a usar objetos para representar todo o tipo de entidades com as suas ações e informações. No entanto, no mundo real a maior parte das entidades relacionam-se com outras. Em muitos casos partilham até algum tipo de informação. Voltando ao exemplo utilizado anteriormente, podemos assumir que existem especificações de carro, como um descapotável ou um jipe, que têm ações e informações semelhantes. Estes fazem parte de um grupo de objetos, o grupo "Carro". A herança numa linguagem de programação como o Java não é mais do que uma representação disso mesmo. Vamos ver um diagrama representativo deste conceito:



Neste diagrama percebe-se que, tanto a classe `Descapotável` como a classe `Jipe` descendem da classe `Carro`. Em Java utiliza-se o termo "**derivação**" para este tipo de relação entre duas classes, ou seja, diz-se que as classes `Descapotável` e `Jipe` derivam da classe `Carro`. A palavra reservada no Java para esta situação é **extends** e deve ser colocada na declaração da classe.

```
public class Classe1 extends Classe2 {
```

A linha anterior a `Classe1` deriva da `Classe2`. Quando uma classe deriva de outra, herda todos os atributos e métodos que a classe base tem, à exceção dos que estiverem definidos como **private**.

Através da herança podemos dizer que um `Descapotável` é um `Carro`, assim como um `Jipe` também o é, mas o contrário pode não ser verdade.

Vamos traduzir este conceito em código:

- › Crie uma **classe** no package `Carro` com o nome `Descapotavel.java` e **adicione** o código:

```
package carro;
```

```
public class Descapotavel extends Carro {
```

```
    public Descapotavel(String matricula_carro, String marca_carro, String cor_carro) {  
        super(matricula_carro, marca_carro, cor_carro);  
    }
```

```
    private boolean estadoCapota;
```

```
    /**  
     * Metodo que alterna o estado da capota  
     */
```

```
    public void abrirFecharCapota(){
```

```
        if (estadoCapota == false){  
            estadoCapota = true;  
        }
```

```
        else {  
            estadoCapota = false;  
        }
```

```
        /*A mesma logica feita num operador ternario seria:  
        estadoCapota = (estadoCapota == true? False : true);  
        */
```

```
    }
```

```
    /**  
     * Mostra o estado corrente da capota  
     */
```

```
    public void mostrarEstadoCapota() {
```

```
        if (estadoCapota == true){  
            System.out.println("Estado da capota : aberta");  
        }
```

```
        else {  
            System.out.println("Estado da capota : fechada");  
        }
```

```
    }
```

```
}
```

Acabámos de definir a classe `Descapotavel` que possui um elemento específico que é a capota, algo que a classe base não possui. Este código contém uma palavra reservada nova: `super`.

```
super(matricula_carro, marca_carro, cor_carro);
```

A instrução `super` neste caso invoca o construtor da classe base com os respetivos parâmetros. Isto significa que a construção de um descapotável é feita através da construção de um carro.

› Na classe `TesteCarro`, **acrescente** o código para testar:

```
Descapotavel desc1 = new Descapotavel("74-83-KY", "Mazda", "Azul");  
desc1.mostrar();  
  
desc1.mostrarEstadoCapota();
```

› **Teste** a classe

📌 Repare como não foi especificado o método `mostrar` na classe `Descapotavel`, mas foi usado sem qualquer problema. Isto acontece porque, uma vez que o descapotável deriva de carro, tem acesso a todos os seus métodos e atributos. O método `mostrar` executado foi o da classe base.

› **Adicione** mais duas instruções no final do método `main` da classe `TesteCarro`:

```
desc1.abrirFecharCapota();  
desc1.mostrarEstadoCapota();
```

› **Teste** de novo o programa

📄 Agora sim estamos a usar métodos e atributos exclusivos da classe `Descapotavel`.

› Vamos **definir** outra extensão da classe `Carro`, a classe `Jipe`:

```
package carro;

public class Jipe extends Carro {
    private boolean tracaoligada;
    private boolean luzestejadilho;

    public Jipe(String matricula_carro, String marca_carro, String cor_carro, boolean luzestejadilho_carro) {
        super(matricula_carro, marca_carro, cor_carro);
        tracaoligada = false;
        luzestejadilho = luzestejadilho_carro;
    }

    /**
     * Alterna o estado da tracao
     */
    public void ligarDesligarTracao(){
        if (tracaoligada == true){
            tracaoligada = false;
        }
        else {
            tracaoligada = true;
        }
    }

    /**
     * Mostra o estado corrente da tracao do jipe
     */
    public void mostrarEstadoTracao(){
        System.out.println("Estado da tracao: " + (tracaoligada == true? "Ligada": "Desligada"));
    }
}
```

› **Acrescente** as instruções de teste no final do método `main` da classe `TesteCarro`:

```
Jipe jipe1 = new Jipe("59-12-TK", "Land Rover", "Castanho", true);
jipe1.mostrar();

jipe1.mostrarEstadoTracao();
jipe1.ligarDesligarTracao();

jipe1.mostrarEstadoTracao();
```

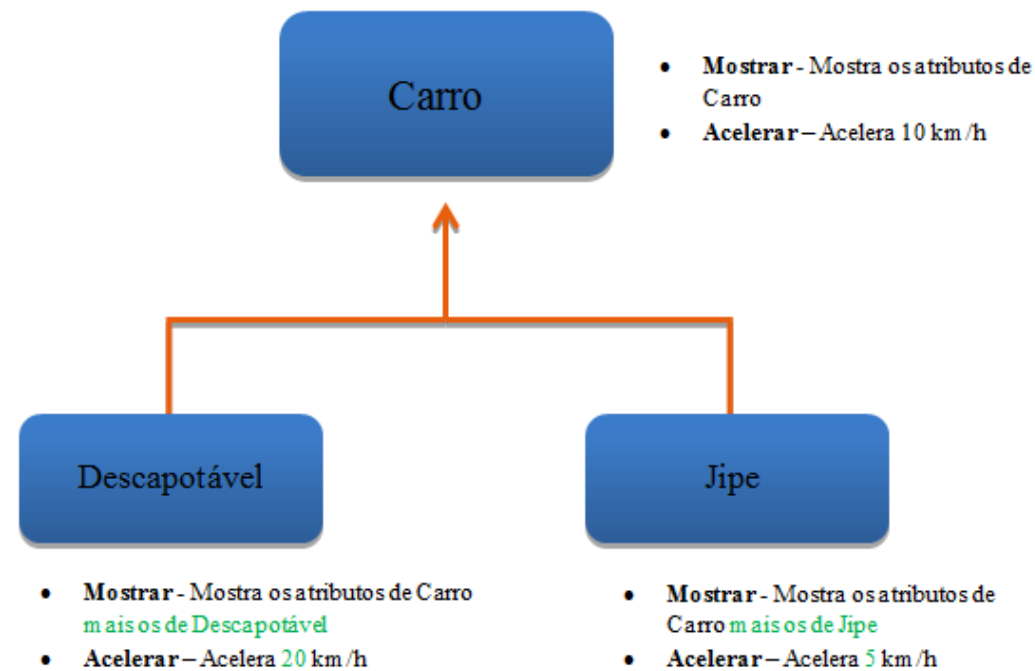
› **Teste** agora a classe `TesteCarro`

Esta classe `Jipe` tem duas características (atributos) que diferem da classe `Carro`, assim como métodos para as manipular. Foi também definido um construtor diferente que recebe mais parâmetros que os da classe base.

O objetivo da herança é reaproveitar métodos e atributos de uma classe base e definir uma ligação lógica e hierárquica entre várias entidades.

2. Polimorfismo

O polimorfismo é um conceito muito importante em linguagens de programação orientada a objetos, que descreve a atribuição de diferentes comportamentos à mesma ação ou método. Seguindo este conceito, a classe base define por omissão um comportamento para uma determinada ação, e as suas classes derivadas redefinem este comportamento de forma a adaptar-se ao seu contexto. Vamos observar este conceito diagrama:



Pelo diagrama é mais fácil perceber que, quando os métodos são executados em objetos diferentes (Carro, Descapotável, Jipe), o seu efeito é diferente. Diz-se que os métodos das classes derivadas sobrepõem-se aos da classe base.

Vamos então redefinir o método `mostrar` da classe `Jipe` de forma a ter um comportamento mais apropriado:

› **Insira** na classe `Jipe` o seguinte código:

```
/**
 * Mostra os atributos do Jipe invocando o bloco de atributos comuns da classe base (Carro)
 */
public void mostrar(){
    System.out.println();
    System.out.println("Jipe:");
    super.mostrar();
    mostrarEstadoTraccão();
}
```

```
if(luzestejadilho == true){  
    System.out.println("Luzes de Tejadilho : Tem");  
}  
else {  
    System.out.println("Luzes de Tejadilho :Não tem");  
}  
}
```

› **Apague** as seguintes linhas do método `main` da classe `TesteCarro`:

```
jipe1.mostrarEstadoTracao();  
jipe1.ligarDesligarTracao();  
jipe1.mostrarEstadoTracao();
```

› **Teste** essa mesma classe e **observe** a diferença

De notar que este método `mostrar` do objeto `jipe1` usa o método da classe `Carro` na seguinte linha:

```
super.mostrar();
```

📄 Significa que são apresentadas as informações específicas da classe `Jipe`, para além das informações normais da classe base `Carro`.

O Eclipse **indica-nos que existe sobreposição de métodos** **1**, comportamento que em inglês se traduz por *Override*:


```
/**
 * Mostra os atributos do Jipe invocando o bloco de atributos comuns da classe base (Carro)
 */
@Override
public void mostrar() {
    System.out.println();
    System.out.println("Jipe:");
    super.mostrar();
}
```

Passemos a aplicar o mesmo conceito à classe `Descapotavel`:

› **Adicione** na classe `Descapotavel` mais um método:

```
/**
 * Mostra os atributos do Descapotavel invocando o bloco
 * de atributos comuns da classe base (Carro)
 */
public void mostrar() {
    System.out.println();
    System.out.println("Descapotavel:");
    super.mostrar();
    mostrarEstadoCapota();
}
```

› **Apague** na classe `TesteCarro` as linhas destacadas:

```
Descapotavel desc1 = new Descapotavel("74-83-KY", "Mazda", "Azul");
desc1.mostrar();

desc1.mostrarEstadoCapota();
```

```
desc1.abrirFecharCapota();  
desc1.mostrarEstadoCapota();
```

› **Teste** a classe e **observe** a diferença

📘 Uma vez que "Jipe" e "Descapotavel" são carros, os objetos destes tipos podem ser guardados em variáveis do tipo "Carro".

› **Altere** na classe `TesteCarro` o código destacado:

```
carro desc1 = new Descapotavel("74-83-KY", "Mazda", "Azul");  
desc1.mostrar();  
  
carro jipe1 = new Jipe("59-12-TK", "Land Rover", "Castanho", true);  
jipe1.mostrar();
```

› **Compile** e execute a classe

📄 Não viu nenhuma diferença, correto? Vamos perceber o que aconteceu.

Na primeira linha estamos a guardar um objeto do tipo "Descapotavel" numa variável do tipo "Carro".

3

2

```
Carro desc1 = new Descapotavel ("74-83-KY", "Mazda", "Azul");
```

Graças ao polimorfismo, o método executado depende do **objeto efetivamente lá guardado** 2 e não do **tipo da variável que o contém** 3 .

Aplicando o mesmo conceito para o método `acelerar`:

› **Adicione** à classe `Jipe` o seguinte método:

```
/**
 * acelera o carro no valor de 10km/h
 */
public void acelerar() {
    velocidade += 10 / CONVERSAO_METROS_SEGUNDO;
}
```

📌 Repare como neste momento lhe aparece um erro de compilação. Isto acontece porque o atributo `velocidade` é privado na classe `Carro`. Para que as classes derivadas tenham acesso a estes atributos e os possam modificar é necessário alterar a sua visibilidade.

› **Altere** na classe `Carro` o seguinte código:

```
package carro;
public class Carro {
    protected String marca;
    protected String matricula;
    protected double velocidade;
    protected String cor;
    protected static final double CONVERSAO_METROS_SEGUNDO = 3.6;
```

› **Confirme** que neste momento não existem erros na classe `Jipe`

› Introduza mais um método na classe `Descapotavel`:

```
/**
 * acelera o carro no valor de 20km/h
 */
public void acelerar() {
    velocidade += 20 / CONVERSAO_METROS_SEGUNDO;
}
```

› Falta apenas **adicionar** o código de **teste** na classe `TesteCarro`:

```
Carro[] carros = new Carro[6];

carros[0] = carro1;
carros[1] = carro2;
carros[2] = carro3;
carros[3] = carro4;
carros[4] = desc1;
carros[5] = jipe1;

for (int i = 0 ; i < 6; ++i){
    System.out.println(carros[i].obterVelocidade());
}

System.out.println("Carros depois de serem acelerados:");

for (int i = 0 ; i < 6; ++i){
    carros[i].acelerar();
    System.out.println(carros[i].obterVelocidade());
}
```

› Compile e teste o código

Foi construído um array de "Carros" e atribuídos os objetos "Carro" e derivados criados anteriormente a cada posição do array. De seguida percorreram-se todos os carros e mostrou-se a sua velocidade corrente. O segundo ciclo acelera cada carro e mostra a sua velocidade final.

Repare como nem todos os carros aceleraram com o mesmo valor, apesar de todos serem "Carros". Isto acontece porque o método acelerar é polimórfico, logo o método executado depende do tipo de objeto que esta variável tem guardado.

Como mencionado previamente, descapotáveis e jipes são carros e, por isso, é possível guardar objetos destes tipos num objeto do tipo "Carro" como foi feito no código anterior:

```
Carro desc1 = new Descapotavel("74-83-KY", "Mazda", "Azul");
```

Já o inverso não é verdade, pelo que a instrução contrária irá resultar num erro de compilação.

› Adicione a seguinte instrução na classe TesteCarro:

```
Jipe jipe2 = new Carro("28-76-QJ", "Ford");
```

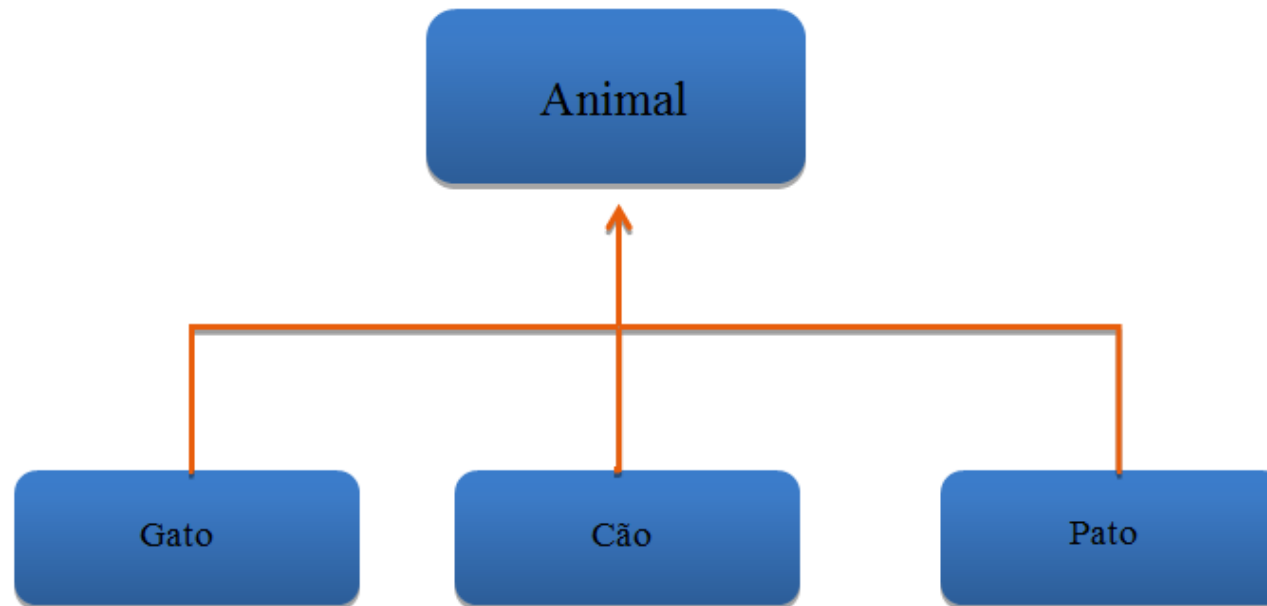
📄 Repare como é mostrado um erro de compilação. Isto porque tendo um objeto do tipo "Carro", não implica que este seja um jipe, logo faltariam informações como a "tracaoligada" entre outras. É por este motivo que esta conversão não é possível.

› Retire esta última instrução

3. Classes Abstratas

As classes abstratas são entidades das quais não é possível criar objetos. Estas servem então apenas para definir atributos e métodos que as classes derivadas vão ter ou podem redefinir. Como regra, uma classe base abstrata apenas define os métodos e não o código para estes. Analisando os animais e uma representação programática para estes, podemos considerar todos os tipos de animais como classes derivadas da classe `Animal`. Seguindo esta linha de raciocínio, podemos considerar que, não faz sentido criar objetos da classe `Animal` diretamente, mas sim dos seus subtipos (classes derivadas) como `Pato`, `Gato`, etc.

Passemos a analisar uma possível estrutura de classes que representa esta ideia:



- › **Comece** por criar um novo package **Animais**
- › Dentro desse package **crie** a classe `Animal.java`
- › **Insira** o seguinte código:

```
package animais;

public abstract class Animal {
    public abstract void manifestarse();
}
```

Esta classe define apenas o método `manifestarse`. Como este método é abstrato necessita de ser implementado nas classes derivadas.

› **Crie** a classe `Gato.java` e **introduza** o seguinte código:

```
package animais;

public class Gato extends Animal{

    public void manifestarse() {
        System.out.println("Miau!");
    }
}
```

› **Crie** outra classe de nome `Cao.java` e **introduza** o código:

```
package animais;

public class Cao extends Animal{

    public void manifestarse() {
        System.out.println("Ão!");
    }
}
```

Vamos agora criar a classe de testes destas classes.

› **Insira** outra classe no package **Animais** com o nome `TesteAnimais.java`

› **Introduza** o código:

```
package animais;

public class TesteAnimais {
    public static void main(String[] args) {

        Cao c = new Cao();
        Gato g = new Gato();

        c.manifestarse();
        g.manifestarse();
    }
}
```

› **Teste** este código

O facto da classe `Animal` ser abstrata impede-nos de criar um objeto diretamente.

› **Adicione** a seguinte instrução à classe `TesteAnimais`:

```
Animal a = new Animal();
```


📄 A compilação do programa irá gerar um erro.

› **Retire** esta última instrução

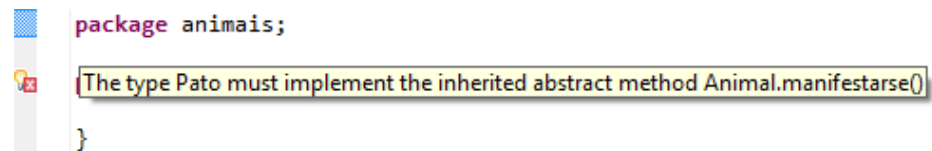
📘 As classes que derivam de uma classe abstrata têm obrigatoriamente que implementar todos os métodos abstratos da classe base.

› **Crie** uma nova classe `Pato.java`

› **Defina** o código através do seguinte:

```
package animais;  
  
public class Pato extends Animal{  
  
}
```

📄 Repare como é indicado um erro de compilação:



```
package animais;  
  
The type Pato must implement the inherited abstract method Animal.manifestarse()  
  
}
```

Como a classe `Pato` deriva de uma classe com métodos abstratos, tem que conter todos esses métodos.

› **Adicione** o seguinte método à classe `Pato`:

```
public void manifestarse() {  
    System.out.println("Quack!");  
}
```