Fundamentos da Programação Orientada a Objetos

UNIDADE 05

Heranças entre classes

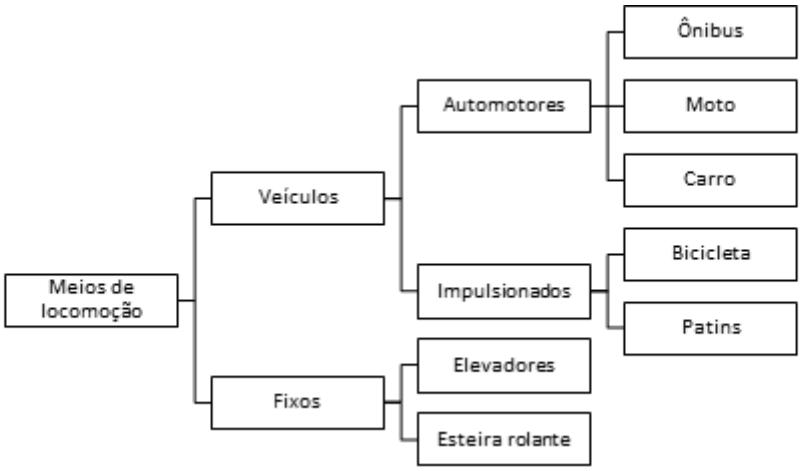
Olá! Nesta Unidade você irá descobrir que as classes não se associam somente por meio da relação de associação, mas também de acordo com sua estrutura, em uma relação conhecida como ***herança****.Desta forma, você poderia criar* ***hierarquias de classes****, permitindo reaproveitar código e subdividir a aplicação com mais facilidade.*

Hierarquias de classes

Você já aprendeu que objetos se associam entre si, por exemplo, na **composição** de um **carro** (todo), que se associa com o **motor**, **rodas**, **portas** etc. (suas partes); ou na **agregação** de um objeto, que pode ou não estar dentro de uma embalagem.

Entretanto, quando procuramos entender o mundo à nossa volta, também **agrupamos os objetos por suas características**. Um exemplo clássico está na biologia: quando estudamos os animais, os subdividimos em reinos, filos, raças etc. Assim, sabemos que cães e gatos são bichos completamente diferentes, mas guardam similaridades por serem ambos mamíferos. Embora a biologia tenha essa hierarquia definida formalmente, nós fazemos o tempo todo uma organização mais informal, como no exemplo ilustrado na Figura 1, nesse caso para organizar diferentes tipos de veículos:

Figura 1 – Hierarquia dos veículos



Fonte: Autores (2021).

Observe que:

- **Carros, ônibus e motos** possuem **características diferentes**: no número de rodas, capacidade de passageiros etc. Porém, **várias similares**: a impulsão usa um motor, o deslocamento é sobre rodas etc. e isso permite entendê-las como **veículos automotores.**
- Isso significa que um **carro** ou **moto** possuem as mesmas **características e operações** que um veículo automotor, pois eles **são um** tipo mais específico deste tipo de veículo (especialização).
- Da mesma forma, os **veículos automotores** e os **impulsionados** podem agrupados como veículos, uma vez que podem deslocar pessoas livremente de um ponto a outro (generalização).
- Classes mais gerais também são chamadas de "**superclasses**" ou "**classes pai**", analogamente, as mais específicas são chamadas de "**subclasses ou classes filhas**". Exemplo: meio de locomoção é uma **superclasse** de **patins**.

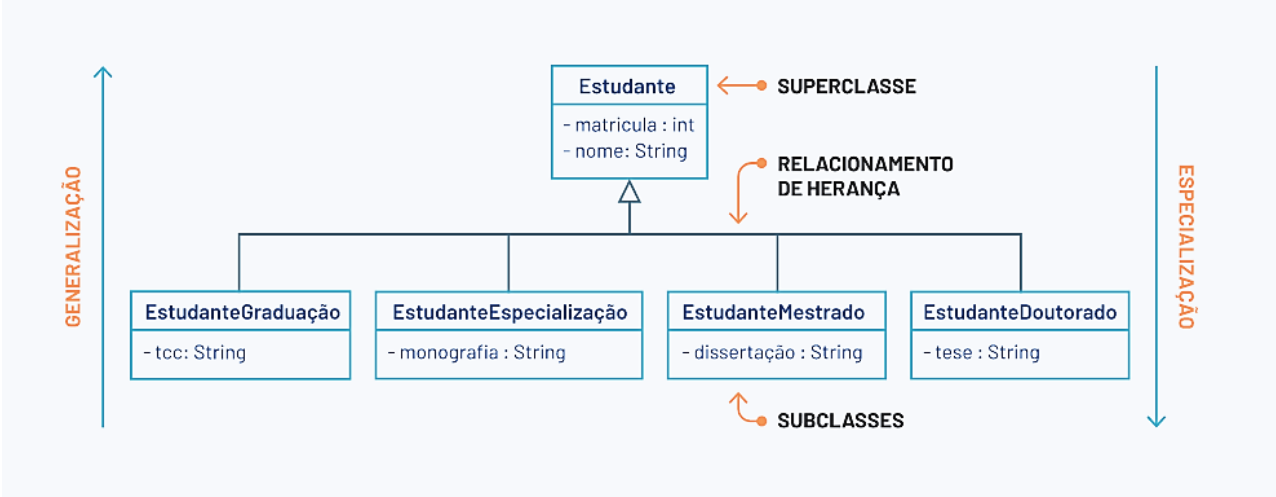
Especialização e generalização

Observe este exemplo de hierarquia, representado com UML: as relações de herança são indicadas pelo símbolo do triângulo vazado, que aponta para a classe pai, ou superclasse.

Nesse caso, a **leitura do diagrama** acompanha o **sentido da hierarquia**:

- Da **subclasse** para a **superclasse** é a **generalização**: vai para a classe mais geral, ou **genérica**.
- Da **superclasse** para a **subclasse** é a **especialização**: vai para a classe mais **específica**.

Figura 2 – Hierarquia dos veículos



Fonte: Autores (2021).

EXERCÍCIO

Buscando similaridades

Vamos trabalhar nos exemplos passados na Figura 1, para ver como esses conceitos são traduzidos em código Java.

1. Crie um projeto com as classes **carro**, **moto** e ***main*** no mesmo pacote default, como indicado.
2. Execute o projeto a partir da classe ***main***.
3. Após isso:

    a. Quais **métodos** e **atributos** idênticos as classes **moto** e **carro** possuem?
    b. Existe algum **método em comum**, cuja **implementação não seja idêntica**? Reflita: o significado do método é o mesmo em ambos os casos?
    c. Agora, observe a classe ***main***. Há trechos de códigos realizando praticamente as mesmas operações? Quais são eles?
    d. O que aconteceria na classe ***main*** se mais classes fossem inseridas?

Figura 3 – Entendimento das similaridades entre as classes carro e moto

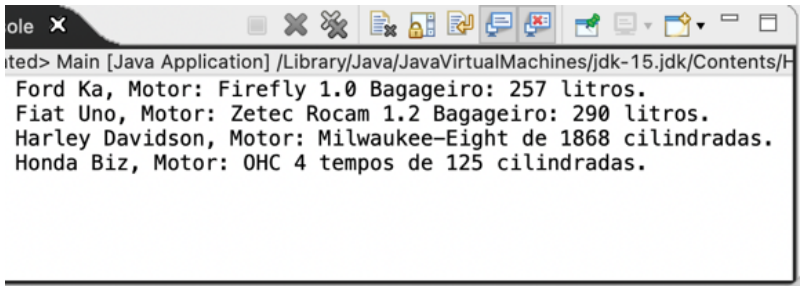
```
</>
1
2
3
4
```

```
5
6   public class Carro {
7       private String marca;
8       private String motor;
9       private int portaMalas;
10
11      public Carro(String marca, String
12  motor, int portaMalas) {
13          this.marca = marca;
14          this.motor = motor;
15          this.portaMalas = portaMalas;
16      }
17
18      public String getMarca() {
19          return marca;
20      }
21
22      public String getMotor() {
23          return motor;
24      }
25
26      public int getPortaMalas() {
27          return portaMalas;
28      }
29
30      public String imprimir() {
31          return "Marca: " + getMarca() + ",
32  Motor: " + getMotor() +
33                  " Porta malas: " +
34  getPortaMalas() + " litros.";
35      }
36  }
37
```

</>

```
public class Moto {
    private String
marca;
    private String
motor;
    private int
cilindradas;

    public Moto(String
marca, String motor,
int cilindradas) {
        this.marca =
marca;
        this.motor =
motor;

this.cilindradas =
cilindradas;
    }

    public String
getMarca() {
        return marca;
    }

    public String
getMotor() {
        return motor;
    }

    public int
getCilindradas() {
        return
cilindradas;
    }

    public String
imprimir() {
        return "Marca:
" + getMarca() + ",
Motor: " + getMotor() +
            " de " +
getCilindradas() + "
cilindradas.";
```

```
        }
    }
```


ole
ted> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/H
Ford Ka, Motor: Firefly 1.0 Bagageiro: 257 litros.
Fiat Uno, Motor: Zetec Rocam 1.2 Bagageiro: 290 litros.
Harley Davidson, Motor: Milwaukee-Eight de 1868 cilindradas.
Honda Biz, Motor: OHC 4 tempos de 125 cilindradas.

</>

```
31
32  import java.util.*;
33
34  public class Main {
35      public static void main(String[] args)
36  {
37          var carros = new ArrayList<Carro>
38  ();
39       var motos = new ArrayList<Moto>();
40
41          carros.add(new Carro("Ford Ka",
42  "Firefly 1.0", 257));
43          carros.add(new Carro("Fiat Uno",
44  "Zetec Rocam 1.2", 290));
45
46          motos.add(new Moto("Harley
47  Davidson", "Milwaukee-Eight", 1868));
48          motos.add(new Moto("Honda Biz",
49  "OHC 4 tempos", 125));
50
51          for (var carro : carros) {
52
53  System.out.println(carro.imprimir());
54          }
55
56          for (var moto : motos) {
57
58  System.out.println(moto.imprimir());
59          }
60      }
61  }
62
```

Fonte: Autores (2021).

Resolução do exercício

Classes moto e carro:

a. Atributos em comum: marca e motor.

Métodos em comum: getMarca(), getMotor().

b. O método imprimir() é similar nas duas classes. Possui o mesmo papel: descrever a classe. Porém, sua implementação, difere um pouco.

c. Classe main:

São utilizadas duas listas, uma de carros e outra de motos.

São feitos dois “for” praticamente idênticos, com a mesma função: imprimir os veículos.

d. Para cada classe incluída, o código seria repetido mais uma vez.

Observe: foi usada a palavra-chave **var** antes das declarações. Você pode utilizá-la em **variáveis locais** para que o Java determine o tipo de dado automaticamente, com base na atribuição. Assim, os comandos:

```
var carros = new ArrayList<Carro>();
var nome = "Orientação à Objetos";
```

São equivalentes à:

```
ArrayList<Carro> carros = new ArrayList<>();

String nome = "Orientação à objetos";
```

Porém, apesar de prático, o comando **var** possui **limitações**. **Não pode ser usado em atributos** e, no caso dele, a inicialização da variável na mesma linha é **obrigatória**.

EXERCÍCIO

Generalizando o código

Neste exercício, melhoraremos o código de exemplo da Figura 3, agrupando as classes similares em uma superclasse, chamada **veiculo**. Faremos isso por meio da palavra-chave ***extends***, que cria relações de herança.

1. Crie uma nova classe, chamada **veiculo**.
2. Em seguida, faça as seguintes alterações no código, como indicado:
    a. Copie os métodos e atributos que são idênticos na classe **carro** e **moto** para a classe **veiculo** e altere os construtores conforme o código a seguir.
    b. Altere a declaração das classes **moto** e **veiculo** para incluir a instrução ***extends* veiculo**.
    c. Exclua os métodos idênticos das classes **carro** e **moto**.
3. Crie uma terceira classe **onibus**.
    a. Adicione a ela os atributos do número de passageiros e se é ou não articulado.
    b. Não esqueça também do método imprimir.
    c. E de criar alguns ônibus na classe ***main***.

Figura 4 – Criando a superclasse comum entre as classes carro e moto

```
</>
```

```
1   public class Veiculo {
2       private String marca;
3       private String motor;
4
5       public Veiculo(String marca, String
6   motor) {
7           this.marca = marca;
8           this.motor = motor;
9       }
10
11      public String getMarca() {
12          return marca;
13      }
14
15      public String getMotor() {
16          return motor;
17      }
18  }
19
```

```
</>
```

```
1
2
3
```

```
public class Carro extends Veiculo {
    private int portaMalas;

    public Carro(String marca, String
motor, int portaMalas) {
        super(marca, motor);
        this.portaMalas = portaMalas;
    }

    public int getPortaMalas() {
        return portaMalas;
    }

    public String imprimir() {
        return "Marca: " + getMarca() + ",
Motor: " + getMotor() +
            " Porta malas: " +
getPortaMalas() + " litros.";
    }
}
```

</>

```

```

```
21
22  public class Moto extends Veiculo {
23      private int cilindradas;
24
25      public Moto(String marca, String
26  motor, int cilindradas) {
27          super(marca, motor);
28          this.cilindradas = cilindradas;
29      }
30
31      public int getCilindradas() {
32          return cilindradas;
33      }
34
35      public String imprimir() {
36          return "Marca: " + getMarca() + ",
37  Motor: " + getMotor() +
38          " de " + getCilindradas() + "
39  cilindradas.";
40      }
41  }
42
```

Fonte: Autores (2021).

Resolução do exercício

Resposta dos itens a e b:

```
</>
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
public class Onibus extends Veiculo {
    private int passageiros;
    private boolean articulado;

    public Onibus(String marca, String
motor, int passageiros, boolean
articulado) {
        super(marca, motor);
        this.passageiros = passageiros;
        this.articulado = articulado;
    }

    public int getPassageiros() {
        return passageiros;
    }

    public boolean isArticulado() {
        return articulado;
    }

    public String imprimir() {
        return "Marca: " + getMarca() + ",
Motor: " + getMotor() +
                " Passageiros: " +
getPassageiros() +
                " Articulado: " +
(isArticulado() ? "sim" : "não");
    }

```

Resposta do item c:

```
</>
```

```
12
13  import java.util.*;
14
15  public class Main {
16      public static void main(String[] args)
17  {
18          var carros = new ArrayList<Carro>
19  ();
20          var motos = new ArrayList<Moto>();
21          var onibuses = new
22  ArrayList<Onibus>();
23
24      //...
25
26          onibuses.add(new Onibus("Mercedes
27  Benz", "V8", 50, true));
28          onibuses.add(new Onibus("Scania
29  Escolar", "V8", 15, false));
30
31      //...
32          for (var onibus : onibuses) {
33
34  System.out.println(onibus.imprimir());
35          }
36
37      }
38  }
39
```

Diagrama de classes da herança

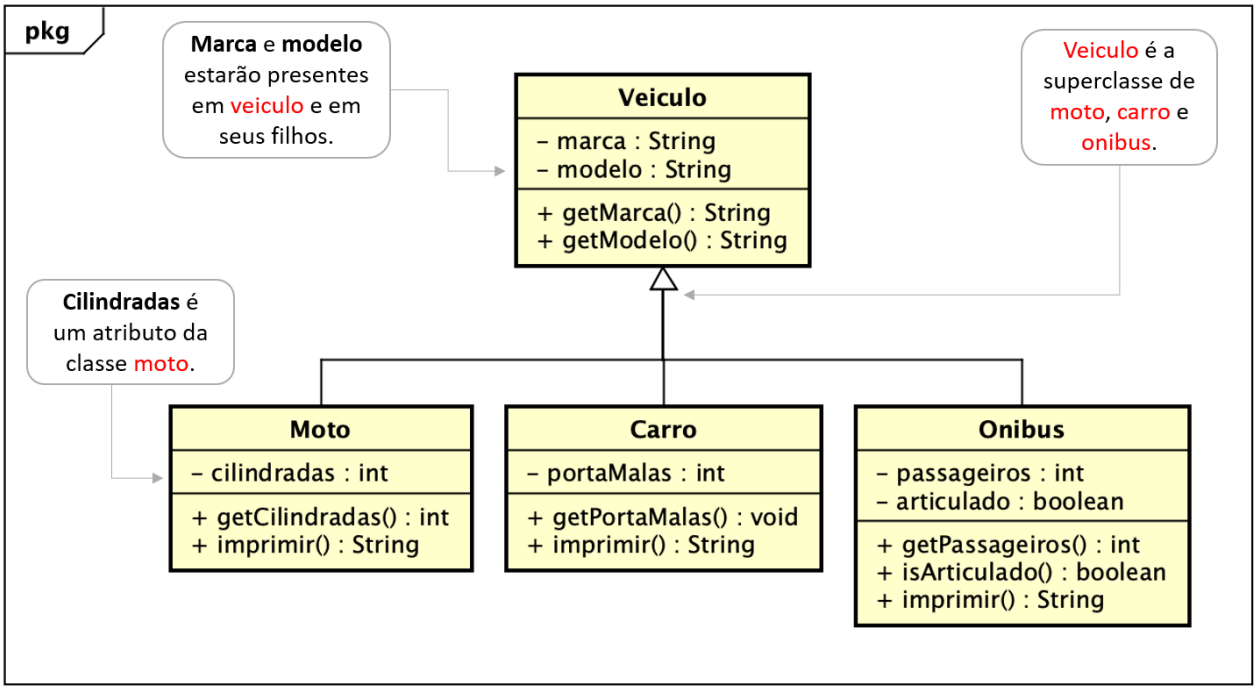
Exatamente como fizemos com a associação, na Unidade passada, o código de exemplo da Figura 4 também pode ser representado por seu diagrama de classes, conforme apresentado na Figura 5.

A especificação em UML das classes do Exercício 2, **veiculo**, **moto**, **carro** e **onibus**:

a. Todos os atributos são privados (encapsulamento, com modificador -).
b. Todos os métodos são públicos (com modificador +, são a interface das classes).

c. Observe o símbolo do triângulo no conector. A ponta do triangulo indica a classe pai.
d. Em Java, cada uma classe pode ter qualquer número de filhos, **mas apenas um único pai**. Outras linguagens, como o C++, permitem mais de um pai (herança múltipla).

Figura 5 – Diagrama das classes moto, carro, onibus e veiculo



Fonte: Autores (2021).

Observe na Figura 5 que, assim como ocorreu no código, não há necessidade de repetir atributos da classe pai que estejam presentes nas classes filhas. Isso acontece porque a relação de **herança** é uma relação de tipos, ou seja, dizemos que o **carro é um tipo** específico de **veículo**. Portanto, ele obrigatoriamente deve possuir os mesmos atributos e métodos que todos os veículos possuem.

> De forma prática, dizemos que a **herança** cria uma relação do tipo **"é um tipo"**, enquanto as **associações** criam relações do tipo **"tem um"**. Se ainda estiver em dúvida, **prefira associação à herança**.

Encadeamento de construtores

Todo objeto, ao ser instanciado, precisará ser completamente construído antes de ser utilizado. Por isso, classes filhas devem chamar um dos construtores da sua classe pai, por meio da palavra-chave **super**. Isso foi observado no código da Figura 4:

```
...     ...
 4      public Carro(String marca, String motor, int portaMalas) {
 5          super(marca, motor);
 6          this.portaMalas = portaMalas;
 7      }
```

Na linha 5, a chamada à **super** fará com que o construtor da classe **veiculo**, descrito a seguir, seja acionado. Observe que a marca e o motor foram fornecidos, uma vez que são os parâmetros de entrada do construtor.

```
...     ...
 5      public Veiculo(String marca, String motor) {
 6          this.marca = marca;
 7          this.motor = motor;
 8      }
```

É importante que você saiba algumas regras sobre encadeamento de construtores:

1. Construtores **não são herdados**. Assim, você precisará criar um construtor na classe filha mesmo que a única coisa que ele faça seja chamar ***super***.
2. A chamada ao comando ***super*** deve estar na primeira linha do construtor.
3. Além do comando ***super***, há também o comando ***this***, que permite chamar um construtor da mesma classe.
4. Você só pode ter um único comando ***super*** ou ***this*** em cada construtor.

Por exemplo, o construtor a seguir poderia ser definido na classe **carro** para permitir a criação de carros com a capacidade padrão de 135 litros no porta-malas.

```
</>
```

```
1   public Carro(String marca, String motor) {
2           this(marca, motor, 135); //Chama o construtor com 3
3   parâmetros de carro
4       }
5
```

Ainda confuso? Então, veja o que o comando:

```
var gol = new Carro("Gol", "1.0");
```

Faria:

1. Ele chamaria o construtor `Carro(String marca, String motor)` com os valores “Gol” e “1.0”.
2. A primeira linha desse construtor, como vimos, possui a chamada a ***this***`(marca, motor, 135)`. Isso fará com que o construtor `Carro(String marca, String motor, int portaMalas)` seja chamado, com os valores “Gol”, “1.0” e 135.
3. Na primeira linha deste construtor, encontra-se a chamada à ***super***`(marca, motor)`. Isso faz com que o construtor `Veiculo(String marca, String motor)` seja chamado.
4. Este construtor inicializa os atributos **`marca`** e **`motor`** da classe **veiculo** com os valores passados (“Gol”, “1.0”).
5. Ao retornar, o construtor de carro inicializa o atributo do **`portaMalas`** com o valor passado (135).

Acesso aos membros herdados

Um objeto da classe filha possuirá todos os **atributos** e **métodos** definidos em sua classe, além dos definidos em todas as suas classes pai. Entretanto, ele poderá ou não os acessar, de acordo com o que foi definido em seus **modificadores de acesso.**

Atributos ***private*** estão acessíveis somente em sua própria classe, onde foram declarados. Por isso, os atributos **marca** e **modelo** não serão diretamente acessíveis nas classes filhas de **veiculo** (**carro**, **moto** e **onibus**). Outros modificadores de acesso – seja o ***public***, ***default*** ou ***protected*** – podem permitir o acesso ao atributo ou método nas classes filhas. Nesse caso, basta utilizá-los diretamente, prefixando ou não o membro com a palavra-chave ***this***, como você faria se ele fosse declarado na própria classe.

Por exemplo, o método **imprimir()** da classe **moto** não pode utilizar diretamente os atributos **marca** e **modelo**, uma vez que foram definidos como ***private***. Porém, a classe utilizou os seus ***getters***, públicos, para acessá-los:

```
  1 public class Moto extends Veiculo {
...     ...
 13     public String imprimir() {
 14         return "Marca: " + getMarca() + ", Motor: " + getMotor() +
 15         " de " + getCilindradas() + " cilindradas.";
 16     }
 17 }
```

Para fixar esse conceito e relembrar os demais modificadores de acesso, que tal utilizarmos o modificador ***protected*** em um pequeno exercício? Podemos recordar as definições para os modificadores de acesso, como está na Unidade 3, encapsulamento,

também apresentado na Figura 6.

Figura 6 – Recordando: modificadores de acesso

Visibilidade				
Modificador em Java	*Class*	*Package*	Subclasse*	Mundo
Sem modificador (default)	☑	☑	☒	☒
`public`	☑	☑	☑	☑
`protected` *	☑	☑	☑	☒
`private`	☑	☒	☒	☒

Fonte: Autores (2021).

Membros *protected*

EXERCÍCIO

Membros *protected*

Conforme já vimos na Unidade 3, um atributo *protected* é **visível e acessível por qualquer classe filha ou classes no mesmo pacote**.

1. Altere a visibilidade do atributo **marca** para ***protected*** e do atributo **motor** para ***default***, como no modelo.
2. Em seguida, altere o método imprimir da classe **moto** para utilizar os atributos diretamente, sem a palavra-chave ***this***.
3. Altere também o método da classe carro para utilizar os atributos, mas prefixe-os com a palavra-chave ***this***.
4. Crie um pacote motos, mova a classe **moto** para lá e reflita: Por que um problema ocorreu?
5. Uma das formas de eliminar esse problema seria trocar o atributo motor para ***public***. Mas essa seria uma solução correta?

Figura 7 – Exercício: membros *protected*

</>

```
public class Veiculo {
    protected String marca;
    String motor;

    public Veiculo(String marca, String
motor) {
        this.marca = marca;
        this.motor = motor;
    }

    public String getMarca() {
        return marca;
    }

    public String getMotor() {
        return motor;
    }
}
```

Fonte: Autores (2021).

Resolução do exercício

Resposta da questão 2:

```
public String imprimir() {
    return "Marca: " + marca + ", Motor: "
+ motor +
          " de " + cilindradas + "
cilindradas.";
}
```

Resposta da questão 3:

```
```

```
5
6   public String imprimir() {
7       return "Marca: " + this.marca + ",
8   Motor: " + this.motor +
9            "  Porta malas: " +
10  this.portaMalas + " litros.";
11  }
12
```

Resposta da questão 4:

Ao mover a classe moto para outro **pacote**, o atributo motor deixou de ser visível na classe moto. Para corrigir, ou teríamos que declará-lo como ***protected***, ou utilizar modificadores de acesso.

Resposta da questão 5:

O modificador ***public*** permite que o atributo sempre seja acessível. Mas essa solução é ruim, pois remove o **encapsulamento da classe**. Utilizar o ***protected*** seria mais vantajoso, pois permitiria o acesso direto somente às classes filhas.

Sobrescrita de métodos

Dê uma boa olhada no método imprimir das classes **carro** e **moto**:

```
1   public class Moto extends Veiculo {
... ...
13      public String imprimir() {
14          return "Marca: " + getMarca() + ", Motor: " + getMotor() +
15          " de " + getCilindradas() + " cilindradas.";
16      }
17  }
```

```
1   public class Carro extends Veiculo {
... ...
3       public String imprimir() {
4           return "Marca: " + getMarca() + ", Motor: " + getMotor() +
5               " Porta malas: " + getPortaMalas() + " litros.";
6
7       }
8   }
```

Notou a **similaridade**? Isso ocorreu, pois a marca e o motor são características de todos os veículos. Por isso, poderíamos imaginar que a classe **veiculo** poderia fornecer uma implementação do método imprimir igual a apresentada a seguir:

```
 1  public class Veiculo {
...     ...
17      public String imprimir() {
18          return "Marca: " + getMarca() + ", Motor: " + getMotor();
19      }
20  }
```

Isto é, obviamente, uma vantagem. Se incluíssemos agora mais dez tipos diferentes de veículos em nossa hierarquia, não precisaríamos implementar o método imprimir novamente. Todos possuiriam o método que, ao ser chamado, imprimiria os atributos **marca** e o **motor**.

Mas o que acontece com esse método nas classes **moto**, **carro** e **onibus**? Eles passam a ser proibidos? Teremos que excluí-los?

A resposta é: **não**. Nós podemos criar versões mais específicas de um método por meio de um recurso chamado de **sobrescrita** (*override*). Se um método de mesma **assinatura** (nome, tipo de retorno e parâmetros) for definido na classe filha, ele será utilizado daquele ponto em diante na hierarquia.

> **Sobrescrita** (*override*) é um recurso que permite a uma **subclasse**, ou **classe filha**, **redefinir** a implementação de um **método herdado** de uma de suas **superclasses** ou **classes pai**.
>
> **Método sobrescrito** é um **método** em **uma classe filha** que tem a **mesma assinatura** (**mesmo nome**, **mesmos parâmetros** e o mesmo **tipo de retorno**) desse mesmo **método** definido em uma de **suas classes pai**.

Pratique o conceito de **sobrescrita** no exercício a seguir.

EXERCÍCIO

Métodos sobrescritos

Teste você mesmo esse comportamento:

1. Adicione o método **imprimir()** à classe **veiculo**, como descrevemos, e em seguida, rode o programa. Você verá que ele continua executando exatamente igual rodava antes.
2. Então, **exclua o método imprimir** de uma das classes filhas (ex.: **moto**) e teste novamente. O que acontece?

Resolução do exercício

Resposta da questão 1:

O programa executa da mesma forma, após a inclusão do método **imprimir()**, como no código a seguir:

```
</>
```

```
1   public class Veiculo {
2       protected String marca;
3       String motor;
4
5       public Veiculo(String marca, String
6   motor) {
7           this.marca = marca;
8           this.motor = motor;
9       }
10
11      public String getMarca() {
12          return marca;
13      }
14
15      public String getMotor() {
16          return motor;
17      }
18              public String imprimir() {  //
19  Método imprimir() acrescentado
20                  return "Marca: " +
21  getMarca() + ", Motor: " + getMotor();
22  }
23  }
24
```

Fonte: Autores (2021).

Resposta da questão 2:

O programa continua rodando, porém um pouco diferente: quando invocamos o método **imprimir()** de um objeto da classe filha que já não tem esse método imprimir() próprio, será invocado o método herdado da classe **veiculo**, apresentando na tela de console apenas os atributos comuns definidos na classe pai: **marca** e **motor**.

Chamando métodos sobrescritos da classe pai

Agora que criamos o método imprimir na classe **veiculo**, você pode estar se perguntando: haveria alguma forma de utilizá-lo nas classes **moto**, **carro** e **onibus** para evitar a repetição do código, uma vez que o sobrescrevemos nessas classes? Em outras palavras, se **moto**, **carro** e **onibus** também possuem o método **imprimir**, haveria alguma forma de utilizar a implementação já feita no mesmo método da classe **veiculo**?

Sim! E a resposta está na palavra-chave ***super***. Podemos usá-la em vez do ***this*** para indicar que queremos chamar o método da classe pai. Assim, seria possível agora **reaproveitar seu código** na hora de implementar o método **imprimir** da classe **moto**, da seguinte forma:

```
 1  public class Moto extends Veiculo {
...     ...
13      public String imprimir() {
14          return super.imprimir() + " de " + getCilindradas() + " cilindradas.";
15      }
16  }
```

Diferentemente do que ocorre nos construtores, você pode utilizar o ***super*** para invocar métodos do pai em qualquer linha – e até mesmo mais de uma vez. Este reaproveitamento torna os códigos mais sucintos e bem mais fáceis de manter. Embora não seja usual, você poderia até mesmo chamar **super.imprimir()** em outro método, que não o **imprimir()**.

Finalmente, lembre-se: todos os métodos da classe pai são herdados automaticamente, ou seja, você só precisa utilizar o super se a classe filha sobrescreveu o método, e você quer diferenciar a chamada entre a versão da classe filha e de seu pai,

tal como fizemos com **moto** e **veiculo**.

Anotação *@Override*

Nas primeiras versões do Java, um problema relativamente banal às vezes dava bastante dor de cabeça aos programadores. Digamos que você criasse a classe **trator** e, na hora de sobrescrever o método **imprimir**, confundisse seu nome e digitasse **imprime**. Entretanto, note que esse engano não geraria qualquer erro. A classe **trator** criaria o método **imprime** como um **novo método**, no momento que você chamasse **imprimir**, a versão de veículo seria chamada.

A partir do Java 5, podemos agora sinalizar a um método de que nossa intenção era sobrescrevê-lo, utilizando a anotação **@Override**:

```
</>
1   @Override
2       public String imprimir() {
3
```

Assim, o compilador poderá emitir um erro caso esse método não exista nas classes pai.

Sobrescrita: últimos detalhes

Por fim, esteja atento a mais alguns detalhes sobre métodos sobrescritos:

a. Um método **sobrescrito** pode aumentar a visibilidade de um método da classe pai, mas nunca a diminuir.
b. Um método ***private*** é exclusivo da classe em que foi declarado e, portanto, não pode ser sobrescrito.
c. Para impedir um método de ser sobrescrito de um ponto em diante da hierarquia de classes, basta declará-lo como **final**, por exemplo: `public final String imprimir()`.
d. Um método sobrescrito pode retornar um tipo mais específico do tipo retorno do mesmo método da classe pai. Isso é chamado de **tipo de retorno covariante**. Por exemplo, se a classe **veiculo** possuir um método chamado `getPneu()` que retorna um objeto do tipo **pneu**, a classe **moto** poderia ter o método `getPneu()` retornando o tipo `PneuDeMoto`, filho de **pneu**, e isso ainda seria uma sobrescrita.

Sobrecarga de métodos

Um conceito similar, mas não relacionado à herança, é o de **sobrecarga** (*overload*) de método. Ela ocorre quando criamos um método com mesmo nome, mas parâmetros diferentes. Isso fará com que o Java escolha qual versão do método deve ser usada de acordo com os parâmetros fornecidos.

> **Sobrecarga** (***overload***) é um recurso que permite que **métodos diferentes, na mesma classe**, tenham o **mesmo nome**, mas **assinaturas diferentes *(*parâmetros ou tipo de retorno diferentes).**

IMPORTANTE

Não confunda a **sobrescrita** (***override***) com a **sobrecarga** (***overload***). Lembre-se:

- Na **sobrescrita**, criamos um método em **classe herdeira** com a **mesma assinatura** do método na **classe pai** (**mesmo nome**, **parâmetros** e **tipo de retorno**).
- Na **sobrecarga**, criamos um método, na **mesma classe**, com **mesmo nome**, mas **parâmetros diferentes**, podendo também ter **tipo de retorno diferente**.

Utilizando a sobrecarga

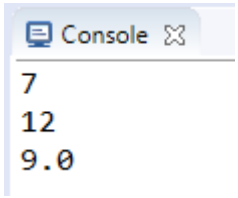
Veja alguns exemplos da utilização da sobrecarga.

1. Implemente o programa a seguir, conforme indicado.
2. Observe a **sobrecarga** (*overload*) do método **somar**, na classe **soma**: existem três implementações diferentes.
3. O **compilador** decide qual método **somar** será invocado, dependendo da forma com que são chamados. Veja essas diferenças nas invocações ao método **somar**, realizadas no método ***main***.

Figura 9 – Verificando a implementação da sobrecarga

</>

```
1   public class Soma {
2       // Overloaded (sobrecarga) de
3   somar(). Esta soma tem 2 parâmetros
4   int;
5       // retorno int
6       public int somar(int x, int y)
7       {
8           return (x + y);
9       }
10
11      // Overloaded (sobrecarga) de
12  somar(). Esta soma tem 3 parâmetros
13  int;
14      // retorno int
15      public int somar(int x, int y,
16  int z)
17      {
18          return (x + y + z);
19      }
20
21      // Overloaded (sobrecarga) de
22  somar(). Esta soma tem 2 parâmetros
23  double;
24      // retorno double
25      public double somar(double x,
26  double y)
27      {
28          return (x + y);
29      }
30
31      public static void main(String[]
32  args) {
33          Soma s = new Soma();
34
35  System.out.println(s.somar(3, 4));
36  // invoca somar
37
38  System.out.println(s.somar(3, 4,
39  5));  // invoca somar
40
41  System.out.println(s.somar(3.4,
42  5.6)); // invoca somar
43
44
```



```
45          }
46      }
47
```

Fonte: Autores (2021).

Veja mais um exemplo: poderíamos ter um método estático na classe **carro** para criar um conjunto de carros baseado em seus modelos. Os modelos poderiam ser passados como parâmetro em um ***array*** de *strings*:

```
</>
1   public static List<Carro> criar(String ... modelos) {
2       var carros = new ArrayList<Carro>();
3       for (var modelo : modelos) {
4           carros.add(new Carro(modelo, "1.0", 250));
5       }
6       return carros;
7   }
8
```

Mas, e se quiséssemos que o método também funcionasse com uma lista de modelos passada por parâmetro? Nesse caso, poderíamos sobrecarregá-lo:

```
</>
1   public static List<Carro> criar(List<String> modelos) {
2       //o método toArray() copia a lista em um vetor
3       return criar(modelos.toArray(new String[0]));
4   }
5
```

Há algumas regras, entretanto, que devem ser observadas ao utilizar métodos **sobrecarregados**:

1. O tipo da lista não pode ser utilizado como parâmetro para fazer sobrecarga. Isso ocorre porque o Java transforma a lista em um **List<Object>** durante a compilação, uma propriedade chamada de *type erasure*. Ou seja, dois métodos com assinatura **metodo(List<String>)** e metodo(**List<Carro>)** após a compilação, ficam com a mesma assinatura.

2. O Java determina qual versão do método utilizar durante a **compilação**. Ou seja, em uma sobrecarga em que um dos tipos é filho de outro, será levado em conta o tipo da **referência**, e não do objeto instanciado. Em seguida, o Java procurará a versão mais específica para fazer a chamada. Por exemplo, suponha a existência das classes avo, pai e filho:

```
</>
```

```
1   public static void main(String args[]){
2       Avo obj = new Filho();
3       teste(obj);  // Chama teste(Avo avo), já que a o
4   variável obj é do tipo Avo
5       teste(new Filho()); // Chama teste(Pai avo) já que a
6   sobrecarga mais específica
7                           //para o tipo Filho é Pai
8   }
9
```

3. O tipo do retorno, **sozinho**, não pode ser usado para gerar uma sobrecarga.

Polimorfismo

Muitas vezes, entendemos que uma **mesma operação** está presente em **objetos diferentes**, e por isso ela pode ser realizada de **maneiras diferentes**. Isto é, seu significado (semântica) não muda, embora talvez sua **forma** seja muito diferente. Na da orientação a objetos, esse processo é conhecido como **polimorfismo**, que é possível em relacionamentos de **herança** entre classes.

> **Herança**: mecanismo que permite que uma **subclasse estenda** uma **classe pai**, ou **superclasse**, provendo **reuso de código** (atributos e métodos), que pode ser incrementado e/ou alterado.
>
> **Polimorfismo** (**polis** = muitas, **morphos** = formas): na POO, o polimorfismo permite enviar uma mesma mensagem (invocação de método) para **diferentes objetos** de **classes herdeiras** da mesma **superclasse (herança)**, e fazê-los **responder de maneira diferente**, ou seja, da forma mais apropriada para cada objeto.

Vamos trabalhar esse conceito no exemplo da figura apresentada a seguir.

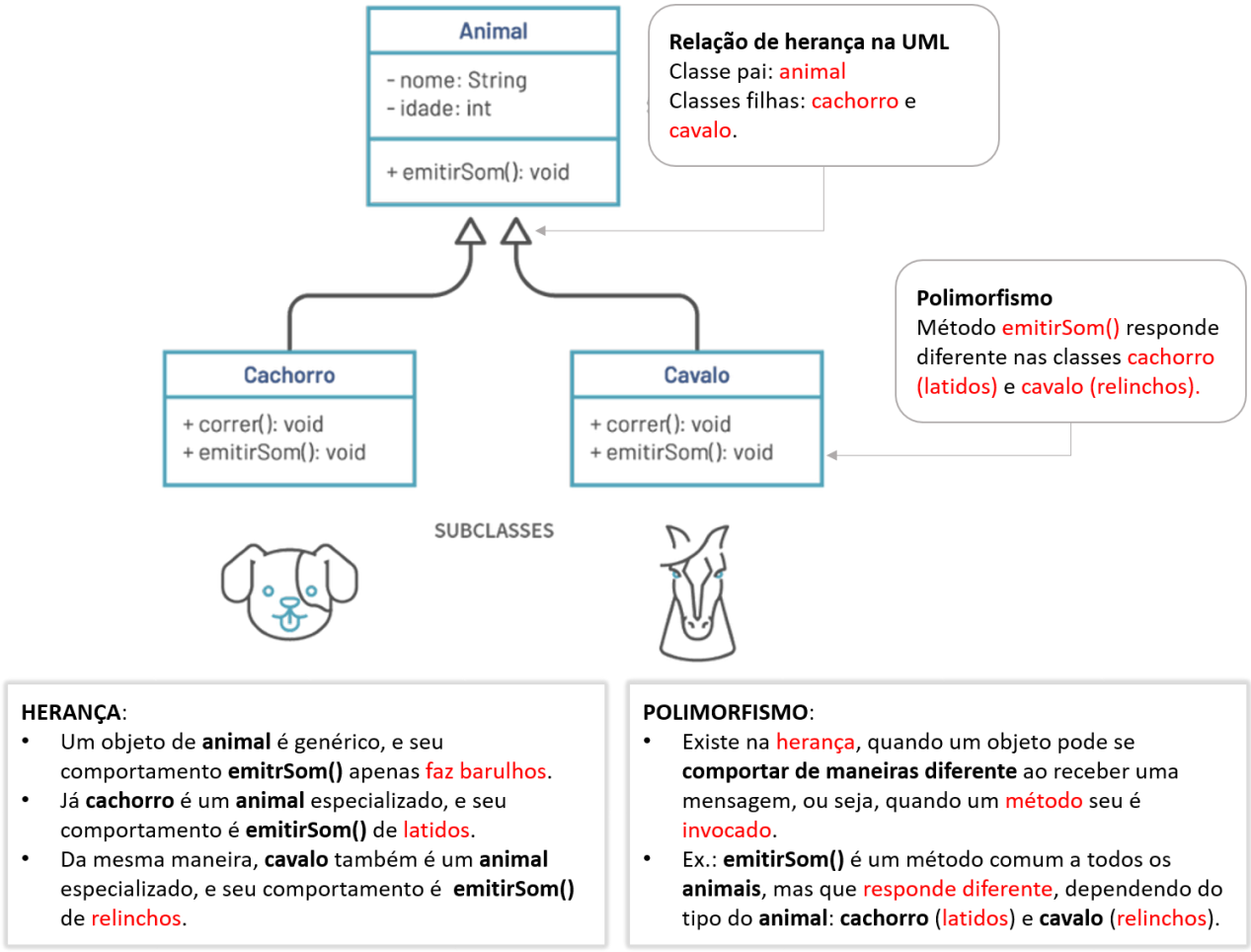
Observe o exemplo de **herança** entre as classes:

- Animal: **superclasse** ou **classe pai**.
- Cachorro e cavalo: **subclasses** ou **classes filhas**.

Logo: cachorro **é um tipo** de animal, e cavalo **é um tipo de** animal.

Observe a sobrescrita (*override*) no método **emitirSom():** ele é **genérico** na classe **animal**, e é **especializado** nas classes **cachorro** e **cavalo**.

Figura 10 – Conceito de polimorfismo



Fonte: Autores (2021).

Vamos continuar com o tema de **polimorfismo** com as classes de **veículo**, **moto** e **carro**: nesse caso, temos mais um exemplo, que pode ser observado no método **imprimir**. Obviamente, os dados impressos por uma **moto** ou **carro** mudam, mas não muda o fato de que a operação em si, de listar esses dados, é a mesma.

Isso ocorre em várias situações, por exemplo, você pode **abrir** uma **caixa**, um **pirulito** e sua **boca**. A ação em si é a mesma: **abrir**. Mas a forma que você abre as três coisas é radicalmente diferente.

Quando falamos em **herança**, lembre-se que dissemos que ela cria uma relação de tipos. Ou seja, quando dizemos que **moto** é uma classe filha de **veiculo**, estamos também dizendo que a uma **moto é um tipo veiculo**. Isso também significa dizer que uma variável do tipo **veiculo** poderá apontar para objetos de qualquer classe filha, como no exemplo a seguir:

```
Veiculo v = new Moto("Harley Davidson", "Milwaukee-Eight", 1868);
```

Essa linha pode ser lida como: "**o veículo v é uma moto**". O Java só permitirá que chamemos em **v** os métodos que seriam permitidos à classe **veiculo**, não os específicos da classe **moto**. Isso inclui até mesmo o método imprimir:

```
v.imprimir();
```

Agora, qual versão do método seria chamada? A da classe **moto**, ou a da classe **veiculo**? Como o tipo da variável é **veículo**, você poderia imaginar que a versão geral do método será chamada. Mas não é isso que ocorre. Pense um pouco: Se te dissermos "o animal A é um gato. Que som faz A?". Você obviamente responderá "miau". E, com o Java, não é diferente.

Esta característica, de o método imprimir de veículo "mudar" seu comportamento de acordo com o tipo específico do objeto sendo referenciado, é chamada de **polimorfismo** (**polis** = muitas, **morphos** = formas).

> Graças ao **polimorfismo**, o Java considerará o tipo **mais específico** da **instância** (objeto), independentemente de qual seja o tipo da variável que o referência.

Isso não só torna a sobrescrita muito mais poderosa, como simplifica drasticamente trechos de nosso código. Podemos imaginar que áreas do código tratam de **camadas de abstração** diferentes. Ficou confuso? Então, que tal relermos o método ***main***?

</>

```
1
2
```

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        var carros = new ArrayList<Carro>();
        var motos = new ArrayList<Moto>();
        var onibuses = new ArrayList<Onibus>();

        carros.add(new Carro("Ford Ka", "Firefly 1.0",
257));
        carros.add(new Carro("Fiat Uno", "Zetec Rocam 1.2",
290));

        motos.add(new Moto("Harley Davidson", "Milwaukee-
Eight", 1868));
        motos.add(new Moto("Honda Biz", "OHC 4 tempos",
125));

        onibuses.add(new Onibus("Mercedes Benz", "V8", 50,
true));
        onibuses.add(new Onibus("Scania Escolar", "V8", 15,
false));

        for (var carro : carros) {
            System.out.println(carro.imprimir());
        }

        for (var moto : motos) {
            System.out.println(moto.imprimir());
        }

        for (var onibus : onibuses) {
            System.out.println(onibus.imprimir());
        }
    }
}

```

Por que há três repetições de código? Três listas? Três *fors*? Se prestamos atenção, o método ***main*** está criando um conjunto de veículos e imprimindo suas características. Se incluíssemos outra classe (trator, por exemplo), provavelmente teríamos que criar outra lista e outro for, para realizar **a mesma tarefa**.

Mas o método ***main*** quer lidar simplesmente com objetos do **veiculo**, não um veículo específico. Ou seja, ele está em uma camada de abstração mais alta, pois foi feito para lidar com uma classe acima da hierarquia. E, para ele, pouco importa qual veículo específico seja esse ou como ele imprima seus dados.

Lembre-se: **moto**, **carro** e **onibus são** veículos. Por isso, nada nos impediria de os adicionarmos a um **List<Veiculo>**. E lembre-se também que o Java saberá qual versão do método **imprimir()** ele deve chamar, graças ao polimorfismo. Assim, veja como poderíamos reescrever esse código:

```
</>
```

```
1   import java.util.*;
2
3   public class Main {
4       public static void main(String[] args) {
5           var veiculos = new ArrayList<Veiculo>();
6
7           veiculos.add(new Carro("Ford Ka", "Firefly 1.0",
8   257));
9           veiculos.add(new Carro("Fiat Uno", "Zetec Rocam
10  1.2", 290));
11          veiculos.add(new Moto("Harley Davidson", "Milwaukee-
12  Eight", 1868));
13          veiculos.add(new Moto("Honda Biz", "OHC 4 tempos",
14  125));
15          veiculos.add(new Onibus("Mercedes Benz", "V8", 50,
16  true));
17          veiculos.add(new Onibus("Scania Escolar", "V8", 15,
18  false));
19
20          for (var veiculo : veiculos) {
21              System.out.println(veiculo.imprimir());
22          }
23      }
24  }
25
```

Viu como ficou mais simples? Isso porque agora o método ***main*** pode tratar todos os veículos de maneira uniforme!

Conversões de tipos

Como vimos, uma referência pode conter tipos de sua classe filha sem qualquer problema. Esse comando sempre será válido:

```
Veiculo v = new Carro(); //O carro é um Veículo
```

Entretanto, a operação contrária não é permitida. Afinal, qualquer carro certamente será um veículo, mas nem todo veículo será um carro (ele pode ser uma moto, onibus etc.).

Em Java, podemos testar o tipo do objeto apontado por uma referência por meio do operador ***instanceof***. Ele retornará verdadeiro caso a variável indicada do lado esquerdo seja do tipo da classe indicado do lado direito, ou sua filha, direta ou indiretamente (ou seja, se é a mesma classe, ou uma classe acima na hierarquia, independentemente de sua altura).

A partir do momento em que temos essa certeza, podemos utilizar a operação de ***type casting*** para converter a referência em um tipo inferior da hierarquia. Por exemplo:

```
//A variável v é um Carro?
        if (v instanceof Carro) {
            //Em caso positivo, convertemos
            Carro carro = (Carro)v; //Down casting – cast "para
baixo"
            //E agora podemos utiliza-la
            System.out.println("Capacidade do porta malas:" +
                    carro.getPortaMalas());
        }
```

Você já usou *type casting* antes, lembra? Ele também é necessário quando você converte um número de ponto flutuante (***float, double***) em um número inteiro. Ele é o operador para conversões perigosas (no caso dos números, pode ser que haja casas decimais que seriam descartadas).

E o que ocorre se a conversão do objeto v não for possível? O Java disparará um erro do tipo ***ClassCastException***.

Por fim, é importante ressaltar que esse tipo a operação de ***instanceof*** é relativamente raro. Normalmente, o polimorfismo e a hierarquia de tipos tornarão esse tipo de conversão desnecessária, resolvendo problemas com muita elegância. Desconfie do seu código caso você esteja fazendo esse tipo de testes o tempo todo.

Que tal um último exercício para reforçar esse conceito?

EXERCÍCIO

Polimorfismo e sobrescrita

O que será impresso pelo código a seguir? Explique.

Figura 11 – Exercício, sobrescrita e polimorfismo

```
</>
```

```
1   class Foo {
2       public void a() {
3           System.out.println("a");
4       }
5
6       public void b() {
7           System.out.println("b");
8       }
9
10      void print() {
11          a();
12          b();
13      }
14  }
15
16  class Bar extends Foo {
17      public void a() {
18          System.out.println("A");
19      }
20
21      public static void main(String[] args)
22  {
23          Foo f = new Bar();
24          f.print();
25      }
26  }
27
```

Fonte: Autores (2021).

Resolução do exercício

O tipo da variável de referência é *foo*, porém, ela aponta para um objeto bar. Graças ao polimorfismo, devemos levar em consideração esse objeto ao avaliar os métodos.

Na linha 23, *print()* é chamado. O objeto bar não possui esse método, por isso, chama o método de *foo*.

Na linha 11, o método *print()* chamará o método *a()*. Como o objeto é do tipo bar, e este possui uma sobrescrita do método (linha 17), sua versão será chamada imprimindo “A”.

Em seguida, na linha 12, o método *b()* será chamado. Não há sobrescrita, por isso, o valor “b” será impresso.

Resultado: “Ab”.

EXPERIMENTE

Mundo dos bichos

Exercício para uso da **herança.**

Que tal melhorarmos o programa dos cães da Unidade 4 para permitirmos outros pets?

1. Ajuste o nome dos atributos para ficarem mais genéricos. Por exemplo, em vez de “nomeCao”, use simplesmente “nome”. Na verdade, não repetir o nome da classe nos atributos é considerado uma boa prática.
2. Crie as classes **gato** e **peixe**, similares à classe **cao**. todos os pets terão um nome e raça. Porém, para os peixes, não é relevante saber seu gênero.
3. Os pets também se alimentam e podem agradar seu dono. Porém, os gatos só aceitam receber festa dos seus donos caso estejam alimentados. Se não estiverem, simplesmente imprima “o gato te ignorou”. Após receberem festa, os gatos voltam a sentir fome.
4. Altere a classes **dono** de modo que ele possa ter qualquer tipo de ***pet***.

5. Renomeie a classe **cauda** para **caudapelo**. Adicione também a classe **caudaescamas**. Todos os pets terão uma cauda, mas o peixe possui escamas.
6. Faça o diagrama de classes do resultado. **Dica**: pode ser mais fácil resolver esses exercícios montando esse diagrama **antes** de implementar os códigos.

Quer saber se acertou? consulte o protocolo de correção aqui.

Referências

GODOY, V. **Programação orientada a objetos I**. Curitiba: IESDE, 2019.

HORSTMANN, C. S.; CORNELL, G. **Core Java – Volume I**. 8. ed. São Paulo: Pearson, 2010.

SCHILDT, H. **Java para iniciantes**. Porto Alegre: Bookman, 2015.