

## Exemplo de Herança, Polimorfismo, Interfaces e *ArrayList*.

1. Considere a classe *Cao* com um campo de dados *nome*, construtor que recebe o *nome* como parâmetro e 2 métodos: *ladra* e *cacaGatos*.

```
public class Cao {  
    private String nome;  
  
    public Cao(String nome) {  
        this.nome = nome;  
    }  
  
    public void ladra() {  
        System.out.println(this.nome + " a ladrar.");  
    }  
  
    public void cacaGatos() {  
        System.out.println(this.nome + " a caçar gatos.");  
    }  
}
```

Construa uma classe *Teste* apenas com método *main*, no qual cria 2 cães, Bart e Fido. Coloque-os num *array* de 10 elementos e invoque o método *ladra*:

```
public class Teste {  
  
    public static void main(String[] args){  
  
        Cao c1 = new Cao("Bart");  
        Cao c2 = new Cao("Fido");  
  
        Cao[] caes = new Cao[10];  
  
        caes[0] = c1;  
        caes[1] = c2;  
  
        for(int i=0; i<caes.length; i++)  
            if(caes[i] != null)  
                caes[i].ladra();  
    }  
}
```

2. Na classe *Cao* coloque um atributo *tamanho* do tipo *int*.

```
public class Cao {  
    ...  
    private int tamanho;  
    ...  
}
```

3. Modifique o método *ladra* da classe *Cao* para acrescentar:

```
public void ladra() {  
    String som;  
    if (this.tamanho > 60) som = "Uff! Uff!";  
    else if (this.tamanho > 20) som = "Ruff! Ruff!";  
    else som = "Ip! Ip!";  
    System.out.println(this.nome + " a ladrar: " + som);  
}
```

Copie este código, insira na classe *Cao*, marque o código inserido, e com o botão direito do rato seleccione: *Format (Alt+Shift+F)*.

4. Comece por executar o método *main* da classe *Teste*. Verifique que o campo de dados *tamanho* é usado, mesmo sem ser inicializado.

Em seguida coloque os tamanhos de Bart e Fido no método *main*.

```
public class Teste {  
    public static void main(String[] args){  
        ...  
        caes[0] = new Cao("Bart");  
        c1.tamanho=70;  
        caes[1] = new Cao("Fido");  
        c2.tamanho=8;  
        ...  
    }  
}
```

Verifique o problema do acesso e a mensagem de erro:

```
"tamanho has private access in Cao"
```

5. Coloque métodos para acesso ao campo de dados *tamanho* (obter e alterar) na classe *Cao* (métodos *getter* e *setter*).

```
public class Cao {  
    ...  
    public int getTamanho() {  
        return this.tamanho;  
    }  
    public void setTamanho(int tamanho) {  
        this.tamanho = tamanho;  
    }  
    ...  
}
```

```
public class Teste {  
    public static void main(String[] args) {  
        ...  
        caes[0] = new Cao("Bart");  
        c1.setTamanho(70);  
        caes[1] = new Cao("Fido");  
        c2.setTamanho(8);  
        ...  
    }  
}
```

Em seguida coloque os tamanhos de Bart e Fido no método *main*, usando o respectivo método *set*.

Execute.

6. Volte a apagar os métodos de acesso *getter* e *setter* e use o *Refactor* do NetBeans.

Com o botão direito do rato sobre o atributo *tamanho*, selecione:

- *Refactor > Encapsulate Fields ...*
- *Refactor*

7. Acrescente o método *ladra(int num)* na classe *Cao* (*overloading*, sobrecarga de métodos):

```
public class Cao {  
    ...  
    public void ladra() { ... }  
    public void ladra(int num) {  
        while( num > 0) {  
            this.ladra();  
            num = num -1;  
        }  
    }  
    ...  
}
```

No método *main* da classe *Teste*, invoque *ladra(2)*.

```
public class Teste {  
    public static void main(String[] args){  
        ...  
        for(int i=0; i<caes.length; i++)  
            if(caes[i] != null)  
                caes[i].ladra(2);  
    }  
}
```

8. Acrescente um construtor sem parâmetros na classe *Cao*.

```
public class Cao {  
    ...  
    public Cao() {  
    }  
    ...  
}
```

9. No método *main* crie um cão através do construtor sem parâmetros, coloque no *array* e execute.

```
public class Teste {  
  
    public static void main(String[] args){  
        ...  
        Cao c3 = new Cao();  
        ...  
        caes[2] = c3;  
        ...  
    }  
}
```

Verifique que os campos de dados são usados, mesmo sem atribuir valores. Coloque métodos de acesso para o campo de dados *nome*. Dê um nome e tamanho ao cão criado.

```
public class Cao {  
    ...  
    public String getNome() {  
        return this.nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    ...  
}
```

```
public class Teste {  
  
    public static void main(String[] args){  
        ...  
        Cao c3 = new Cao();  
        c3.setNome("Snoopy");  
        c3.setTamanho(30);  
        ...  
    }  
}
```

10. Substitua o ciclo *for* do método *main* pelo ciclo *for* designado "for each":

```
public class Teste {  
    public static void main(String[] args) {  
        ...  
        for(Cao c : caes)  
            if(c != null)  
                c.ladra(2);  
    }  
}
```

Significa: para cada elemento "c" do tipo *Cao* no *array* *caes*, se for diferente de *null*, execute a ação *ladra(2)*.

11. Os *arrays* têm tamanho fixo. Para colocar um objeto num *array*, esse objeto tem de ser atribuído a um índice específico. A remoção ou inserção de um elemento numa determinada posição obriga a deslocamento de outros elementos.

*ArrayList<E>* é uma classe da biblioteca de classes da linguagem Java, utilizada para servir de contentor de objetos. O parâmetro *E* permite definir o tipo de elementos do *ArrayList*. Os objetos são guardados sob o tipo de dados *E* e retribuídos também como *E*.

*ArrayList<E>* possui muitos métodos, sendo os principais:

<code>public boolean <b>add</b>(E e)</code>	adiciona o objeto <i>e</i> , recebido por parâmetro, no final do <i>ArrayList</i> .
<code>public int <b>size</b>()</code>	retorna o número de elementos do <i>ArrayList</i> .
<code>public E <b>get</b>(int index)</code>	retorna o objeto guardado no índice indicado como parâmetro (mas sob o tipo de dados <i>E</i> ).
<code>public E <b>remove</b>(int index)</code>	remove o objeto guardado no índice enviado como parâmetro.
<code>public boolean <b>remove</b>(Object o)</code>	remove o objeto <i>o</i> , recebido por parâmetro, se ele existir no <i>ArrayList</i> .
<code>public boolean <b>contains</b>(Object o)</code>	retorna <i>true</i> se o objeto <i>o</i> , recebido por parâmetro, existir no <i>ArrayList</i> .
<code>public int <b>indexOf</b>(Object o)</code>	retorna o índice do objeto <i>o</i> , recebido por parâmetro, se existir no <i>ArrayList</i> , ou -1.
<code>public boolean <b>isEmpty</b>()</code>	retorna <i>true</i> se o <i>ArrayList</i> não tiver elementos.

A classe `ArrayList<E>` está contida no *package* `java.util`.

Reescreva novamente o método *main* da classe *Teste* usando a classe `ArrayList<E>`.

```
import java.util.ArrayList;

public class Teste {

    public static void main(String[] args){
        Cao c1 = new Cao("Bart");
        c1.setTamanho(70);
        Cao c2 = new Cao("Fido");
        c2.setTamanho(8);

        ArrayList<Cao> caes = new ArrayList<>();

        caes.add(c1);
        caes.add(c2);

        for(int i=0; i<caes.size(); i++) {
            Cao c = caes.get(i);
            c.ladra(2);
        }
    }
}
```

12. Modifique o ciclo *for* por um ciclo "for each":

```
public class Teste {

    public static void main(String[] args){
        ...
        for(Cao c: caes) {
            c.ladra(2);
        }
    }
}
```

13. Coloque no *ArrayList* uma *String*.

```
public class Teste {  
    public static void main(String[] args){  
        ...  
        caes.add(c1);  
        caes.add(c2);  
  
        caes.add("String é um tipo diferente do tipo Cao");  
        ...  
    }  
}
```

Verifique que dá erro em tempo de compilação:

```
no suitable method found for add(String)  
  method Collection.add(Cao) is not applicable  
    (argument mismatch; String cannot be converted to Cao)  
  method List.add(Cao) is not applicable  
    (argument mismatch; String cannot be converted to Cao)  
  method AbstractCollection.add(Cao) is not applicable  
    (argument mismatch; String cannot be converted to Cao)  
  method AbstractList.add(Cao) is not applicable  
    (argument mismatch; String cannot be converted to Cao)  
  method ArrayList.add(Cao) is not applicable  
    (argument mismatch; String cannot be converted to Cao)  
-----  
(Alt-Enter shows hints)
```

14. Apague a colocação da *String* no *ArrayList* e altere o código do método *main* da classe *Teste*.

```
public class Teste {  
    public static void main(String[] args){  
        ...  
        ArrayList<Cao> caes = new ArrayList<>();  
  
        caes.add(c1);  
        caes.add(c2);  
  
        for(Cao c: caes) {  
            c.ladra(2);  
        }  
    }  
}
```



15. Suponhamos que precisávamos de representar vários animais entre os quais cães, gatos e leões, com as seguintes características:

Atributos:

- String nome – o nome do animal.

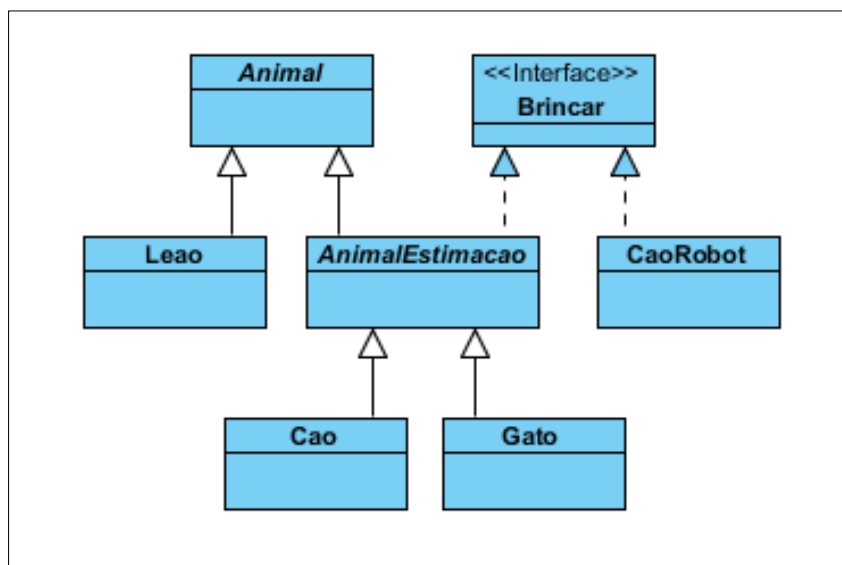
Comportamentos:

- come() - Comportamento quando o animal come.
- fazRuido() - Comportamento quando o animal faz ruído.

Os cães e os gatos por serem animais de estimação ainda apresentam um comportamento comum: *brinca()*.

Um *CaoRobot* também apresenta este comportamento *brinca()* mas não pertence à mesma árvore hierárquica dos cães, gatos e leões pois não come.

Implemente as classes e interfaces seguintes:



```
public abstract class Animal {
    private String nome;

    public Animal(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

    public abstract void fazRuido();
}
```

```
public class Leao extends Animal {  
    public Leao(String nome) {  
        super(nome);  
    }  
    public void fazRuido() {  
        System.out.println( this.getNome() + " a rugir... " );  
    }  
}
```

```
public interface Brincar {  
    void brinca();  
}
```

```
public abstract class AnimalEstimacao extends Animal  
                                implements Brincar {  
    public AnimalEstimacao(String nome) {  
        super(nome);  
    }  
    public void brinca() {  
        System.out.println( this.getNome() + " a brincar." );  
    }  
}
```

```
public class Cao extends AnimalEstimacao {  
    public Cao(String nome) {  
        super(nome);  
    }  
    public void fazRuido() {  
        System.out.println( this.getNome() + " a ladrar." );  
    }  
    public void cacaGatos() {  
        System.out.println( this.getNome() + " a caçar gatos." );  
    }  
}
```

```
public class Gato extends AnimalEstimacao {  
    public Gato(String nome) {  
        super(nome);  
    }  
    public void fazRuido() {  
        System.out.println( this.getNome() + " a miar... " );  
    }  
    public void cacaRatos() {  
        System.out.println( this.getNome() + " a caçar ratos." );  
    }  
}
```

```
public class CaoRobot implements Brincar {  
    private String nome;  
    public CaoRobot(String nome) {  
        this.nome = nome;  
    }  
    public void brinca() {  
        System.out.println( this.nome + " a brincar." );  
    }  
}
```

Se o *ArrayList* só permitir o armazenamento de objetos *Brincar*:

```
import java.util.ArrayList;

public class Teste {

    public static void main(String[] args) {

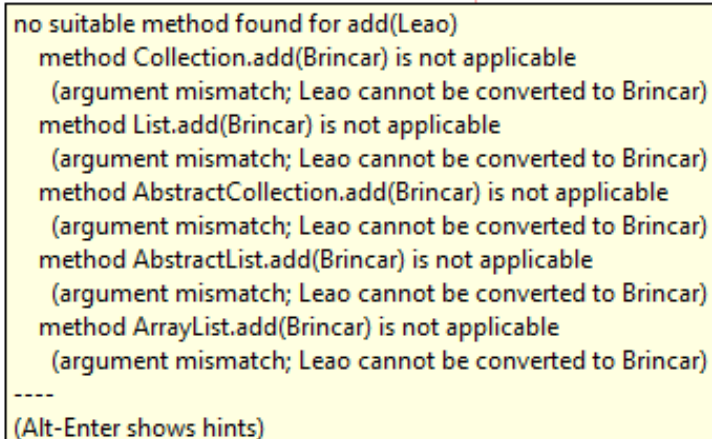
        Cao c1 = new Cao("Bart");
        Cao c2 = new Cao("Fido");
        Gato g1 = new Gato("Bolinhas");
        CaoRobot r1 = new CaoRobot("CaoRobot");
        Leao l1 = new Leao("Simba");

        ArrayList<Brincar> animais = new ArrayList<>();

        animais.add(c1);
        animais.add(c2);
        animais.add(g1);
        animais.add(r1);
        animais.add(l1); // erro de compilação

        for (Brincar b : animais) {
            b.brinca();
        }
    }
}
```

A verificação de compatibilidade de tipos é feita em tempo de compilação e dá erro de compilação ao adicionar ao contentor um objeto que não é do tipo *Brincar*.



```
no suitable method found for add(Leao)
  method Collection.add(Brincar) is not applicable
    (argument mismatch; Leao cannot be converted to Brincar)
  method List.add(Brincar) is not applicable
    (argument mismatch; Leao cannot be converted to Brincar)
  method AbstractCollection.add(Brincar) is not applicable
    (argument mismatch; Leao cannot be converted to Brincar)
  method AbstractList.add(Brincar) is not applicable
    (argument mismatch; Leao cannot be converted to Brincar)
  method ArrayList.add(Brincar) is not applicable
    (argument mismatch; Leao cannot be converted to Brincar)
  ----
  (Alt-Enter shows hints)
```

Assim são evitados erros de execução no ciclo “for each”.