
# Programação I

## Herança

Jorge Roberto Trento
Bacharel em Ciências da Computação - UNOESC
Especialização em Ciências da Computação - UFSC
Formação Pedagógica - Formadores de Educação Profissional – UNISUL
Especialização em Ensino Superior – FIE
Especialização em Gestão de Tecnologia da Informação – FIE

# Introdução

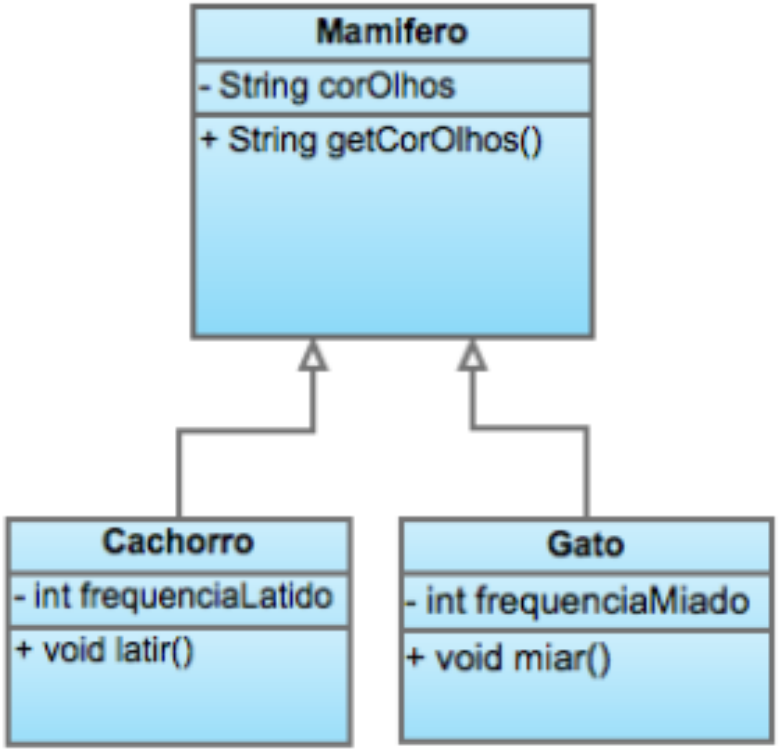
- A possibilidade da reutilização de código é uma das características mais poderosas da programação orientada a objetos.
- O conceito de herança está intimamente ligado à operação da abstração de generalização e especialização, na qual um conjunto de classes pode compartilhar características em comum, em vez de cada uma delas implementar repetidamente essas características.
- O princípio básico é reutilizar o que for comum e especializar o que for específico.

# Introdução


- A herança permite que uma classe herde atributos e métodos de outra classe. Isso permite que novas classes sejam criadas com base em outra classe preexistente e, deste modo, já possuam de antemão todos os atributos e métodos desta.
- As subclasses podem redefinir métodos herdados (sobreposição ou override), além de conter seus próprios atributos e métodos.

# Exemplo

- A classe Mamifero é chamada de superclasse (ou classe pai) e nela colocaremos o atributo corOlhos e quaisquer outros atributos/métodos comuns às classes Cachorro e Gato.
- Cachorro e Gato tornam-se então subclasses (ou filhas) de Mamífero.
- A notação UML para herança é uma seta aberta apontando para a superclasse, como mostra a figura abaixo.



# Explorando o exemplo anterior


- A superclasse contém todos os atributos e métodos que são comuns às classes que herdam dela. A superclasse Mamífero contém todas as características que definem um mamífero, então não é necessário que essas características sejam reimplementadas nas subclasses.
- Sem a utilização de herança, ambas as classes Cachorro e Gato deveriam conter todos os atributos e métodos, mesmo os que são comuns, levando a uma replicação de código.
- Ambas as classes Cachorro e Gato herdam de Mamifero . Isso quer dizer que os atributos e métodos da classe Mamífero também estão presentes nas classes Cachorro e Gato .

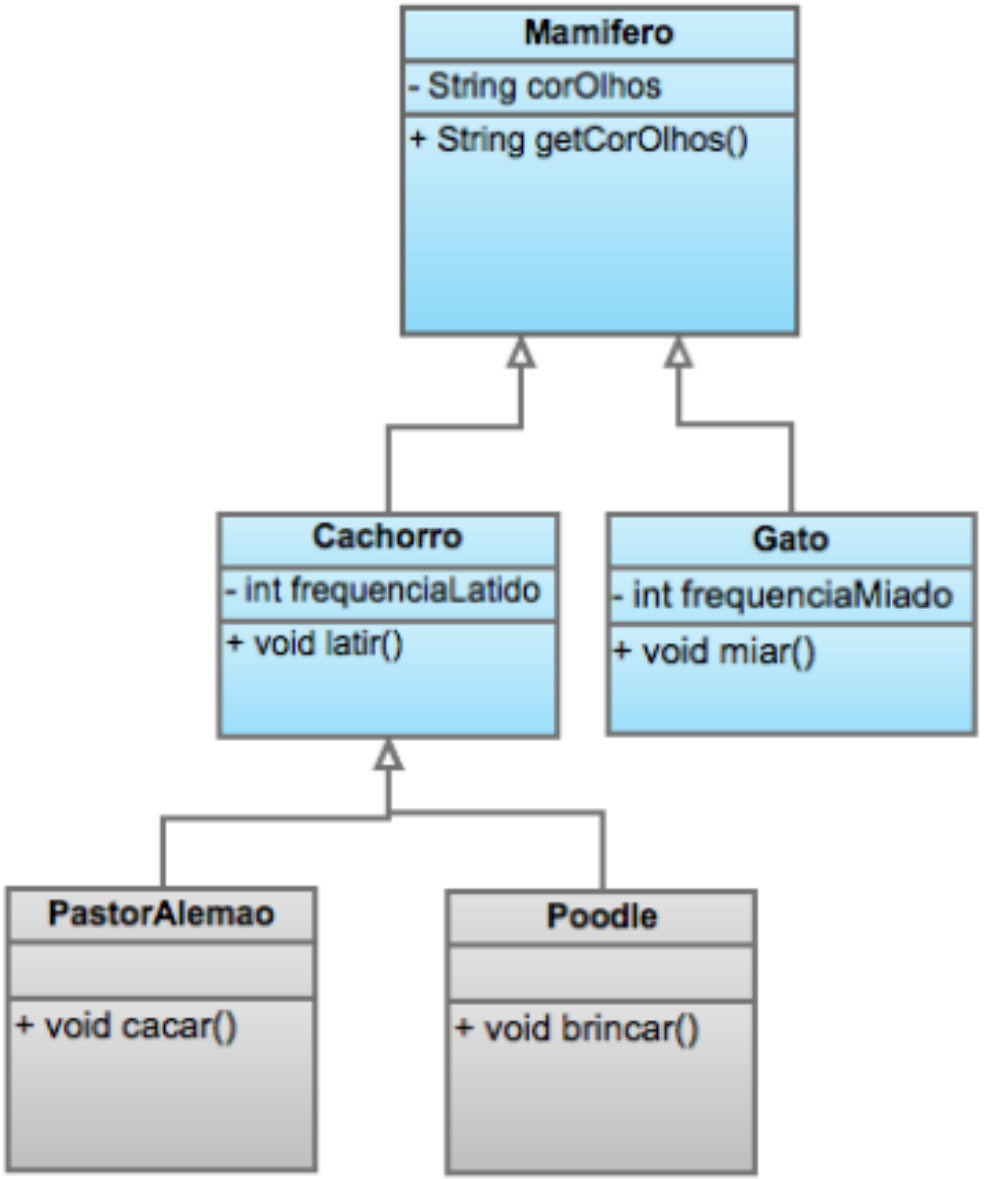
# Explorando o exemplo anterior


- Analisando-se a classe Cachorro , podemos ver que ela tem os seguintes atributos:
  - corOlhos (herdado da classe Mamífero);
  - frequenciaLatido (definido na própria classe Cachorro);
- e os seguintes métodos:
  - getCorOlhos() (herdado da classe Mamifero);
  - latir() (definido na própria classe Cachorro).
- Quando um objeto da classe Cachorro for instanciado, ele terá todos os atributos e métodos definidos na própria classe Cachorro mais as propriedades e métodos definidos na classe Mamífero (que foram herdadas).

# Operações de Abstração envolvidas na herança


- **Especialização:** operação em que, a partir de uma classe, identifica-se uma ou mais subclasses, cada uma especificando características adicionais em relação à classe mais geral;
- **Generalização:** operação de análise de um conjunto de classes que identifica características comuns a todas, tendo por objetivo a definição de uma classe mais genérica, a qual especificará essas características comuns.

# A classe cachorro pode ser ainda mais especializada



# Implementando herança


```java
// Arquivo Mamifero.java

public class Mamifero {
	private String corOlhos;

	public String getCorOlhos() {
		return this.corOlhos;
	}
}


// Arquivo Cachorro.java

public class Cachorro extends Mamifero {
	private int frequenciaLatido;

	public void latir() {
		System.out.println("Au au!");
	}
}


// Arquivo PastorAlemao.java

public class PastorAlemao extends Cachorro {

	public void cacar() {
		System.out.println("Estou caçando!");
	}
}
```

# Implementando herança

- Conforme explicado anteriormente, a herança permite que a classe filho herde as propriedades e métodos da classe pai. Isso quer dizer que, embora o método getCorOlhos() tenha sido definido na classe Mamifero , ele pode ser utilizado na classe Cachorro porque ele foi herdado. Veja abaixo:

```
public class Teste {
	public static void main(String[] args) {
		Cachorro r = new Cachorro();

		r.latir();
		System.out.println("Cor dos olhos = " + r.getCorOlhos());
	}
}
```

# OBSERVAÇÕES


- As classes filhas só conseguirão utilizar métodos e atributos que tenham sido marcados como **public** ou **protected** pela classe pai. Se algum atributo ou método for marcado como **private** pela classe pai, as subclasses não terão acesso a eles.
- Outro aspecto que merece destaque é que uma classe pode ter várias filhas, mas pode ter apenas um pai, o que é chamado herança simples. **Em Java não existe herança múltipla.**

# Sobreposição de métodos


- É possível que, em determinadas situações, ao modelarmos uma classe como subclasse de outra, seja necessário redefinir a implementação de um ou mais métodos. Esse processo é denominado **sobreposição**, **sobrescrita** ou **override** e envolve a declaração de métodos com assinatura idêntica tanto na superclasse como na(s) subclasse(s).

- Utilizando o exemplo anterior, vamos imaginar que o método latir() da classe PastorAlemao tenha que ser diferente da implementação feita na superclasse Cachorro . Nesse caso, teremos que sobrepor o método latir() da classe PastorAlemao para que ele seja diferente:

# Exemplo de sobreposição


```
public class PastorAlemao extends Cachorro {

	public void cacar() {
		System.out.println("Estou caçando!");
	}

	// Método abaixo será sobreposto...
	@Override
	public void latir() {
		System.out.println("Woof Woof!");
	}
}
```

# Teste de sobreposição


```
public class Teste {
      public static void main(String[] args) {
            Cachorro r = new Cachorro();
            PastorAlemao p = new PastorAlemao();

            r.latir();  // imprime "Au au!"
            p.latir();  // imprime "Woof woof!"
      }
}
```

- Conforme mostrado no exemplo, o método latir() do objeto Cachorro imprime "Au au!" na tela. O método latir() do objeto PastorAlemao também imprimiria "Au au!" na tela, porém ele foi sobreposto para imprimir "Woof woof!".
- No momento da execução do método sobreposto, o compilador Java opta pelo método relacionado ao tipo do objeto que o chamou.

# Anotação @Override


- A anotação @Override é opcional. Além de melhorar a legibilidade do código para o programador, ela informa explicitamente ao compilador que o método a seguir é uma sobreposição.
- Assim, se houver qualquer erro na declaração da assinatura do método que está sendo sobreposto, o compilador acusará.
- Sem a anotação @Override , por exemplo, se o método latir() for redefinido na subclasse como Latir() (com letra maiúscula), este último será entendido como um novo método e não como sobreposição.
- Não haverá erro de compilação, mas certamente o funcionamento do programa poderá se dar de forma diferente do planejado, o que pode tomar um tempo maior do programador para descobrir o problema.

# O uso de SUPER


- Em determinadas situações, precisamos sobrepor um método herdado, porém ao invés implementarmos ele de forma diferente, precisamos apenas adicionar um novo comportamento, mantendo o funcionamento anterior.
- No exemplo da classe PastorAlemao , vamos supor que o método latir() precisa fazer o mesmo que o método latir() da classe Cachorro (imprimir “Au au!”), porém ele precisa imprimir "Terminei o latido" após latir.

# Exemplo SUPER

- Da mesma forma que o operador this acessa os métodos e atributos da própria classe, o operador super acessa os métodos e atributos da superclasse (como se fosse um this da superclasse);

```
public class PastorAlemao extends Cachorro {

	public void cacar() {
		System.out.println("Estou caçando!");
	}

	public void latir() {
		super.latir(); // Imprime "Au au!"
		System.out.println("Terminei o latido");
	}
}
```

# Herança e Construtores


- Quando um objeto de uma subclasse é criado, ao executar seu construtor, primeiramente será executado o construtor da superclasse de forma automática.
- Na subclasse, podemos chamar explicitamente qualquer construtor da superclasse através do comando super() (construtor padrão) ou super(lista de argumentos) (outros construtores). Se um construtor da superclasse não for chamado explicitamente, o construtor padrão (sem argumentos) será chamado.

# Herança e Construtores


- O exemplo a seguir mostra a chamada do construtor da superclasse em diversos níveis da hierarquia. Todos os atributos envolvidos estão marcadas como private , o que faz com que elas estejam indisponíveis para as classes filhas através da herança.

# // Arquivo Mamifero.java


```
class Mamifero {
    private String corOlhos;
    public Mamifero(String novaCor) {
        this.corOlhos = novaCor;
    }
}
```

# // Arquivo Cachorro.java


```
class Cachorro extends Mamifero {
    private int frequenciaLatido;
    // Cachorro tem 2 construtores:
    public Cachorro(){
        super(); // chama construtor da superclasse
        this.frequenciaLatido = 0;
    }
    public Cachorro(String cor, int freqLatido) {
        super(cor); // chama construtor da superclasse passando a cor
        this.frequenciaLatido = freqLatido;
    }
}
```

# // Arquivo PastorAlemao.java


```
class PastorAlemao extends Cachorro {
    private int areaCaca;
    public PastorAlemao() {
        // Chamando o construtor sem parâmetros da classe Cachorro
        super();
        // Agora inicializamos as propriedades da classe
        // PastorAlemao
        this.areaCaca = 13;
    }
}
```

# Explorando o exemplo anterior


- Cada uma das classes da hierarquia chama o construtor da sua superclasse.
- É importante mostrar que o construtor da classe PastorAlemao invoca o construtor de sua superclasse ( Cachorro ), que por sua vez chamará o construtor de sua superclasse ( Mamifero ).
- Quando super estiver sendo usado para invocar o construtor da superclasse, deverá ser o primeiro comando no construtor da subclasse, caso contrário, teremos um erro de compilação.

# Resumo do capítulo


- Herança permite o compartilhamento de métodos e atributos entre classes em uma hierarquia.
- Propriedades e métodos marcados como public ou protected são visíveis pelas classes filho através da herança. Elementos marcados como private não são visíveis pelas classes filho.
- A classe mais alta na hierarquia chama-se superclasse (ou classe pai), já as mais baixas (que herdam da superclasse) chamam-se subclasses (ou classes filho).
- A sobreposição de métodos serve para mudarmos a implementação de um método na classe filho se a implementação da classe pai não é adequada para o contexto.
- Através do operador super é possível acessar métodos e propriedades não sobrepostos da classe pai. Ex.: super.latir().
- A chamada super() é equivalente a chamar o construtor da classe pai.

# Herança

Autor
Prof. Doglas André Finco
doglas.andref@uffs.edu.br

jorgertrento@uffs.edu.br