

# Generalização, Especialização, Herança, e Polimorfismo



Prof. Jeferson Souza, MSc. (thejefecomp)

[jeferson.souza@udesc.br](mailto:jeferson.souza@udesc.br)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

JOINVILLE  
CENTRO DE CIÊNCIAS  
TECNOLÓGICAS

# Conceitos Associados ao Desenho de Software (i.e. Programa) Orientado a Objetos

## Afinal, Para que servem os Conceitos de Generalização, Especialização, Herança, e Polimorfismo?

Os conceitos de Generalização, Especialização, Herança, e Polimorfismo são conceitos fundamentais do Paradigma de desenho de software (i.e. programa) orientado a objetos. Esses conceitos remetem à necessidade de identificar partes comuns, específicas, os relacionamentos entre elas, e as diferentes formas e comportamentos que sua utilização pode oferecer.

# Conceitos Associados ao Desenho de Software (i.e. Programa) Orientado a Objetos

**Do Geral ao mais específico é só uma questão de perspectiva...**

É somente uma questão de perspectiva incorporar os referidos conceitos, e o relacionamento entre eles por meio do paradigma orientado a objetos, a começar pelo forma de pensamento. Vamos a eles!

# Generalização

## Afinal, o que é Generalização?

O conceito de Generalização remete à identificação de partes comuns aos programas modelados por meio do paradigma orientado a objetos. Essas partes comuns configuram um modelo geral, o qual pode ser estendido a formas mais especializadas de acordo com a necessidade e os requisitos do programa. A Herança é identificada como o relacionamento associado à Generalização, a configurar uma relação de parentesco entre o modelo geral e sua representação mais específica.

# Generalização - Exemplo

## Exemplo de Generalização

Imaginem o grupo constituído por todas as espécies classificadas como Mamíferos. No caso de realizar a representação mais geral deste grupo, poderíamos criar uma classe chamada “Mamifero”, a qual conteria todos os atributos e métodos presentes em todos os Mamíferos, independente da espécie. Esta classe “Mamífero” representa uma Generalização.

```
public class Mamifero {  
  
    ...  
  
}
```

PS: No caso da linguagem de programação Java a Generalização também pode ser representada por um tipo especial de classe chamada de *Interface*.

# Especialização

## Afinal, o que é Especialização?

O conceito de Especialização remete à identificação de partes especializadas nos programas modelados por meio do paradigma orientado a objetos. Essas partes especializadas permitem atender aos requisitos de forma mais pontual. A parte mais específica de um relacionamento de Herança, e/ou a implementação específica de uma interface comum a diferentes tipos de classes, representam uma Especialização.

# Especialização - Exemplo

## Exemplo de Especialização

Vamos pensar em uma espécie específica de mamífero tal como as Baleias. Caso seja realizada a modelagem das Baleias por meio de uma classe chamada “Baleia”, a considerar o relacionamento com a classe “Mamífero” descrita no exemplo de Generalização, a referida classe “Baleia” representa uma Especialização. A classe “Baleia”, portanto, possui atributos, métodos, e implementações de métodos, que pertencem somente a espécie das Baleias.

```
public class Baleia extends Mamifero {  
    ...  
}
```

PS: No caso da linguagem de programação Java a Especialização também pode ser representada pela classe que implementa uma *Interface*.

# Herança

## Afinal, o que é Herança?

Herança é um conceito que permite estabelecer uma relação de parentesco entre duas classes de uma modelo orientado a objetos, a configurar duas extremidades bem definidas, as quais podem ser classificadas como classe Pai e classe Herdeira.



# Herança

## Classe Pai implica em Generalização

A classe Pai especifica, portanto, um modelo mais geral, o qual possui características que são comuns a todas as classes definidas como suas herdeiras. Do ponto de vista do relacionamento de Herança, a classe Pai é um exemplo de Generalização.

PS: Na linguagem de programação Java a classe **java.lang.Object** representa a classe Pai de todas as classes Java, a incluir as classes definidas em qualquer programa escrito com a referida linguagem.

# Herança

## Classe Herdeira implica em Especialização

A classe Herdeira representa um modelo mais específico, a possuir características únicas se comparada com a classe Pai, e outras classes herdeiras da mesma hierarquia. Do ponto de vista do relacionamento de Herança, a classe Herdeira é um exemplo de Especialização.

PS: Na linguagem de programação Java, a considerar o relacionamento de Herança com a classe **java.lang.Object**, todas as classes definidas por um programa, sem exceção, podem ser consideradas exemplos de Especialização.

# Tipos de Herança

Existem dois tipos de relacionamento de Herança: **Simples** e **Múltipla**.

## Herança Simples

É o relacionamento direto estabelecido entre a classe Herdeira e somente uma única classe Pai.

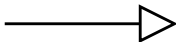
## Herança Múltipla

É o relacionamento direto estabelecido entre a classe Herdeira e mais de uma classe Pai ao mesmo tempo. A classe Herdeira possui características de todas as classes Pai, de forma similar ao que acontece com a nossa herança genética, onde herdamos características dos nossos pais.

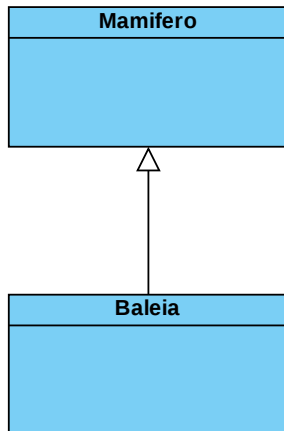
# Herança - Símbolo na Unified Modeling Language (UML) [OMG, 2017]

## Símbolo de Herança (definido como Generalization no termo da língua Inglesa)

Na *Unified Modeling Language* (UML) [OMG, 2017], o símbolo que representa um relacionamento de Herança é uma seta com ponta triangular não-preenchida, tal como especificada abaixo:

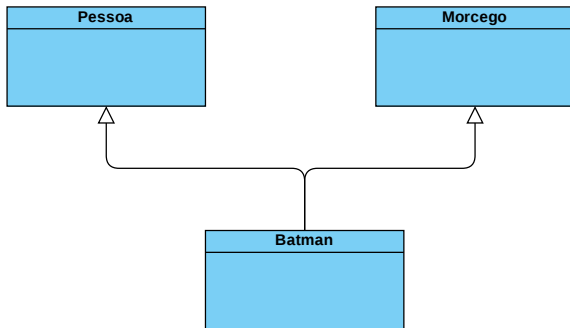


# Herança Simples - Exemplo



Exemplo de relacionamento de Herança simples entre as classes  
“Mamifero” e “Baleia”.

# Herança Múltipla - Exemplo



Exemplo de relacionamento de Herança múltipla com as classes “Pessoa” e “Morcego” como classes Pai, a dar origem ao famoso **Batman, o “Homem Morcego”** :-D.

# Herança em Java

## Herança é definida pelo uso da palavra reservada **extends**

Em Java, o relacionamento de Herança estabelecido entre duas classes é definido pelo uso da palavra reservada **extends** na classe Herdeira, a indicar a classe Pai no momento de sua declaração.

## Java só suporta Herança Simples...

Em Java, o relacionamento de Herança só pode ser de um único tipo: **Herança Simples**. Portanto, somente é possível herdar características de uma única classe Pai de forma direta. Entretanto, essa restrição é contornada pelo uso de um tipo especial de classe chamado de *Interface*.

# Herança em Java

## De acordo com James Gosling, o criador da Linguagem Java...

Em seu *white paper* intitulado “**Java: An Overview**” [Gosling, 1995], o qual descreve o ponto de vista das decisões fundamentais tomadas no desenho do “core” da linguagem Java, decisões estas que são observadas até as versões atuais, James Gosling é categórico em afirmar que:

**“JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance<sup>\*</sup>, and extensive automatic coercions.”**

**\* A explicação sobre como a linguagem Java dá a volta à restrição da Herança múltipla é fornecida mais a frente nas transparências, a indicar o porquê da não existência do famigerado conceito de “Herança múltipla de Interfaces”.**



# Herança em Java

## Exemplo de Declaração

```
public class Mamifero {  
  
    ...  
}  
  
public class Baleia extends Mamifero {  
  
    ...  
}
```

No exemplo acima, a declaração da classe “Baleia” especifica um relacionamento de Herança com a classe “Mamifero”, classe esta referenciada após a palavra reservada **extends**.

# Herança em Java - Classe Abstrata

## Classe Abstrata

A definição de uma classe abstrata constitui uma boa estratégia para representar um modelo mais geral, o qual precisa obrigatoriamente de classes mais especializadas para ter suas características materializadas. As classes abstratas são definidas pela palavra reservada **abstract**.

# Herança em Java - Classes Abstratas

## Regras para definição de classe abstrata [Boyarsky&Selikoff, 2015]

1. Classes abstratas não podem ser instanciadas diretamente;
2. Classes abstratas podem ser definidas com qualquer número de métodos abstratos e não-abstratos, a incluir nenhum;
3. Classes abstratas não podem ser marcadas como **private** **protected**, ou **final**;
4. Uma classe abstrata ao estender outra classe abstrata herda todos métodos abstratos da classe Pai como se fossem definidos por ela;
5. A primeira classe concreta a estender uma classe abstrata deve fornecer uma implementação para todos os métodos abstratos herdados.



# Herança em Java - Exemplo de Classe Abstrata

## Exemplo de Classe Abstrata

Ao modificar a classe “Mamifero” para uma representação abstrata, a mesma fica declarada da seguinte forma:

```
public abstract class Mamifero {  
    ...  
}  
  
public class Baleia extends Mamifero {  
    ...  
}
```

Percebe-se no exemplo acima que apesar da classe “Mamifero” ter sido convertida em uma classe abstrata, o seu relacionamento de Herança com a classe “Baleia” continua a ser declarado da mesma forma, por meio da palavra reservada **extends**.

# Herança em Java - Exemplo de Classe Abstrata

## Exemplo de Classe Abstrata com método abstrato

```
public abstract class Mamifero {  
    public abstract String getNome();  
}  
  
public class Baleia extends Mamifero {  
    public String getNome(){  
        return "Baleia";  
    }  
}
```

O método abstrato *getNome()* precisa, obrigatoriamente, de uma implementação concreta na classe "Baleia".

# Herança em Java - Classes Abstratas

## Regras para definição de método abstrato [Boyarsky&Selikoff, 2015]

1. Métodos abstratos só podem ser definidos em classes abstratas;
2. Métodos abstratos não podem ser marcados como **private** ou **final**;
3. Métodos abstratos não podem ter corpo, ou seja, ser implementados na classe abstrata na qual foram declarados;
4. Implementar um método abstrato em uma classe Herdeira (i.e. subclasse) segue as mesmas regras utilizadas para sobrescrever qualquer outro método em Java (assunto para Polimorfismo): o nome e a assinatura devem ser os mesmos, e a visibilidade do método na subclasse deve ter, ao menos, a mesma visibilidade declarada na classe Pai.

# Herança em Java - Como dar a volta à restrição da Herança Simples?

## Simplesmente diz-se: Oláaaaaaa Interface!!! :-D\*

Como sabe-se, a linguagem Java suporta somente herança simples. Entretanto, o desenho da linguagem permite-nos utilizar um tipo especial de classe para dar a volta à restrição imposta. Este tipo especial de classe é denominado *Interface*.

## \*Referência aos Animaniacs da Warner

Os sensacionais macaquinhos que possuem o seguinte bordão:

**Hellooooooooo Nurse!!! (Oláaaaaaa Enfermeira!!!) :-D**

# Herança em Java - Como dar a volta à restrição da Herança Simples?

## Simplesmente diz-se: Oláaaaaaa Interface!!! :-D\*

Como sabe-se, a linguagem Java suporta somente herança simples. Entretanto, o desenho da linguagem permite-nos utilizar um tipo especial de classe para dar a volta à restrição imposta. Este tipo especial de classe é denominado *Interface*.

## Afinal, o que é uma Interface?

Uma *Interface* é um “tipo especial de classe abstrata” presente na linguagem de programação Java. Diferente de uma classe abstrata “tradicional”, uma *Interface* é declarada com uma palavra reservada específica denominada **interface**, a definir, principalmente, assinaturas de métodos que necessitam ser concretizados com implementação especializada.



# Herança em Java - Como dar a volta à restrição da Herança Simples?

## Declarar uma Interface

A declaração de uma *Interface* é realizada da seguinte forma:

```
<modificador_de_acesso> abstract interface <nome_da_interface> {  
    atributos (estáticos) e métodos...  
}
```

Onde:

1. <modificador\_de\_acesso> pode ser **public** ou o *default*, i.e., sem modificador de acesso;
2. O classificador opcional **abstract** é assumido como presente na declaração de uma interface, ou seja, não é preciso incluí-lo;
3. A palavra reservada **interface** é obrigatória;
4. <nome\_da\_interface> define o nome da *Interface* a ser especificada.

# Herança em Java - Como dar a volta à restrição da Herança Simples?

## Regras para definição de Interface [Boyarsky&Selikoff, 2015]

1. *Interfaces* não podem ser instanciadas diretamente;
2. Uma *Interface* não precisa ter nenhum método;
3. Uma *Interface* não pode ser marcado como **final**;
4. Todas as *Interfaces* de mais alto nível possuem acesso público ou *default*. Todas são consideradas abstratas, mesmo a não ter o classificador explicitamente definido. Logo, a presença de um método marcado como **private**, **protected**, ou **final** remete a um erro de compilação;
5. Todos os métodos *non-default* possuem (por definição) o modificador **public** e o classificador **abstract** em suas definições.

# Herança em Java - Como dar a volta à restrição da Herança Simples?

## Exemplo de declaração de Interface

```
public interface ICintoDeUtilidades{  
    void lancarBatCorda(float extensao);  
    void obterLocalizacao(long idBatLocalizador);  
    void acionarBatLaser(long idBatLaser, int intensidade);  
}
```

PS: Existe uma convenção a dizer que toda *Interface* deve começar com a letra maiúscula "I", a ser seguida por outra palavra a começar por letra maiúscula. Entretanto, essa convenção não é obrigatória, mas algo que pode ser utilizado para diferenciar Classes e *Interfaces* pela nomeação.

# Herança em Java - Como dar a volta à restrição da Herança Simples?

## O uso da palavra reservada **extends** com Interfaces

A palavra reservada **extends** também pode ser utilizada na definição de uma *Interface*, a configurar uma extensão dos métodos declarados em outras *Interfaces*. Diferente da Herança simples onde não pode-se ter mais de uma classe Pai, na extensão de *Interfaces* é possível estender múltiplas *Interfaces* ao mesmo tempo.

# Herança em Java - Como dar a volta à restrição da Herança Simples?

## Exemplo do uso da palavra reservada **extends** com Interfaces

```
public interface IBatCorda{  
    void lancarBatCorda(float extensao);  
}  
  
public interface IBatLocalizador{  
    void obterLocalizacao(long idBatLocalizador);  
}  
  
public interface IBatLaser{  
    void acionarBatLaser(long idBatLaser, int intensidade);  
}  
  
public interface ICintoDeUtilidades extends IBatCorda, IBatLocalizador, IBatLaser{ }
```

PS: Pode-se verificar que ao estender mais de uma *Interface*, todas são separadas por vírgula. Na prática, seria o mesmo que declarar todos os métodos presentes nas *Interfaces* IBatCorda, IBatLocalizador, e IBatLaser diretamente na *Interface* ICintoDeUtilidades. **O Batman que se cuida :-D...**

# Herança em Java - Como dar a volta à restrição da Herança Simples?

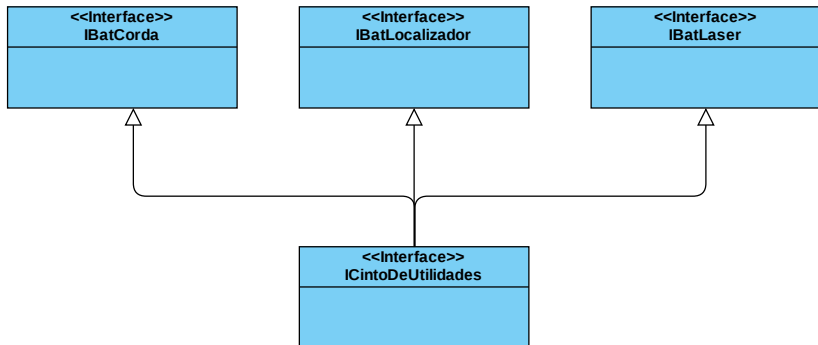
## Detalhe do uso da palavra reservada **extends** com Interfaces

Ao realizar a concretização de uma *Interface* com múltiplas extensões, uma classe deve especificar uma implementação para cada um dos métodos *non-default\** presentes em todas as *Interfaces* referenciadas por meio da palavra reservada **extends**. Internamente, a máquina virtual Java representa essas extensões como um tipo especial de hierarquia, o qual assemelha-se à Herança Múltipla suportada, e.g., pela linguagem de programação C++.

Na minha humilde opinião, simplesmente genial :-D...

\* A partir da versão 8 da linguagem de programação Java é possível definir métodos em uma interface que possuem uma implementação *default*, ou seja, um comportamento pré-estabelecido.

# Herança em Java - Como dar a volta à restrição da Herança Simples?



Exemplo da extensão de *Interfaces* na Linguagem de Programação Java.

# Concretização de Interfaces

## Afinal, o que é Concretização de Interfaces?

A **Concretização** de *Interfaces* permite especificar um comportamento especializado à uma definição mais generalizada de métodos, a qual é especificada por meio da declaração de uma *Interface*. A Concretização de *Interfaces* define, portanto, mais uma forma de ter modelos mais generalizados, os quais devem ser concretizados em classes especializadas.



# Concretização de Interfaces

## Interface implica em Generalização

Uma *Interface* específica, portanto, um modelo mais geral, o qual possui métodos com assinaturas comuns a todas as suas concretizações. Do ponto de vista do relacionamento de Concretização, a *Interface* é um exemplo de Generalização.

# Concretização de Interfaces

## Classe Concreta implica em Especialização

Uma Classe Concreta concretiza uma *Interface*, e portanto, implementa os diferentes comportamentos de seus métodos de uma forma especializada. Do ponto de vista do relacionamento de Concretização, a Classe Concreta é um exemplo de Especialização.

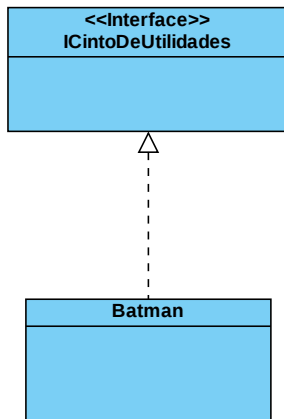
# Concretização - Símbolo na Unified Modeling Language (UML) [OMG, 2017]

## Símbolo de Concretização (definido como Realization no termo da língua Inglesa)

Na *Unified Modeling Language* (UML) [OMG, 2017], o símbolo que representa um relacionamento de Concretização é uma seta tracejada com ponta triangular não-preenchida, tal como especificada abaixo:



# Concretização de Interfaces: Exemplo



Exemplo de relacionamento de Concretização (i.e. *Realization*)

# Concretização de Interfaces em Java

**Concretização é definida pelo uso da palavra reservada **implements****

Em Java, o relacionamento de Concretização estabelecido entre uma Classe e uma *Interface* é definido pelo uso da palavra reservada **implements** na Classe a concretizar uma dada *Interface*.

**Java suporta Concretização de Múltiplas Interfaces...**

Em Java, uma Classe pode concretizar mais de uma *Interface* ao mesmo tempo, a possibilitar, portanto, a materialização de um comportamento similar ao da Herança Múltipla presente, e.g., na linguagem de programação C++.

# Concretização de Interfaces em Java

## Exemplo de Declaração Com Uma Interface

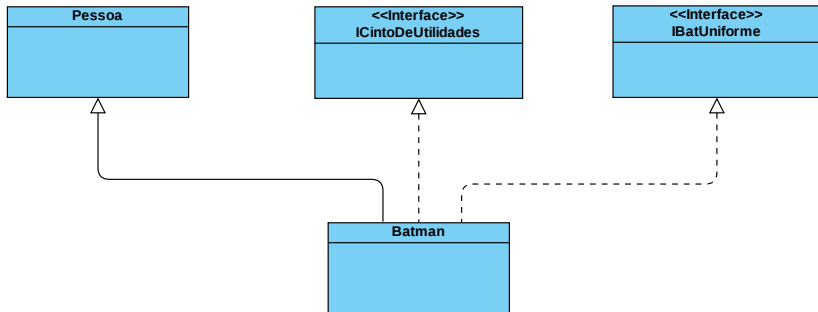
```
public class Batman implements ICintoDeUtilidades{  
    void lancarBatCorda(float extensao){  
        ...  
    }  
    void obterLocalizacao(long idBatLocalizador){  
        ...  
    }  
    void acionarBatLaser(long idBatLaser, int intensidade){  
        ...  
    }  
}
```

# Concretização de Interfaces em Java

## Exemplo de Declaração Com Múltiplas Interfaces

```
public class Batman implements ICintoDeUtilidades, IBatUniforme{  
    void lancarBatCorda(float extensao){  
        ...  
    }  
    void obterLocalizacao(long idBatLocalizador){  
        ...  
    }  
    void acionarBatLaser(long idBatLaser, int intensidade){  
        ...  
    }  
    void materializarUniforme(){  
        ...  
    }  
}
```

# Java: Batman Begins :-D...



Versão Java do Batman  
(Não está muito mais próximo da Realidade?)



# Variáveis de Interface em Java

## Afinal, o que são variáveis de Interface?

Variáveis de *Interface* são atributos que podem ser definidos diretamente na *Interface*, mas que só podem ser declarados consoante as seguintes regras:

1. são considerados, por definição, **public**, **static**, e **final**. Qualquer marcação contrária implica em um erro de compilação;
2. O valor de uma variável de *Interface* deve ser definido no momento de sua declaração, a considerar o classificador **final**.

# Variáveis de Interface em Java - Exemplo de declaração [Boyarsky&Selikoff, 2015]

## Exemplo de declaração com três (3) possíveis variações

```
public interface PodeNadar{  
    int PROFUNDIDADE_MAXIMA = 100;  
    final static boolean SUBMERSO = true;  
    public static final String TIPO = "SUBMERSSIVO";  
}
```

As variáveis de *Interface* assumem o papel de constantes e, por possuir característica estática, não são “herdadas” por outras *Interfaces*, ou concretizações de *Interface*.

# Métodos de Interface default em Java

## Afinal, o que são métodos de Interface default?

Introduzidos a partir da versão 8 da Linguagem Java, os métodos de *Interface default* adicionam um comportamento padrão à definição de um dado método de uma *Interface*. Os métodos de *Interface default* são definidos com a palavra reservada **default**, a possuir como característica marcante uma implementação concreta realizada diretamente na *Interface*.

# Métodos de Interface default em Java

## Flexibilidade e compatibilidade garantidas...

Os métodos de *Interface default* permitem uma maior flexibilidade na codificação e manutenção do código-fonte, além de fornecer compatibilidade com implementações realizadas em versões anteriores da linguagem. Antes da versão 8, caso fosse necessário adicionar um método novo à uma *Interface*, todas as suas concretizações também seriam modificadas, mesmo que a implementação do método novo fosse igual para todas. Com os métodos de *Interface default* basta adicionar o novo método na *Interface* sem a necessidade de modificar as referidas concretizações, as quais podem usar o comportamento padrão.

# Métodos de Interface default em Java - Exemplo [Boyarsky&Selikoff, 2015]

## Exemplo de definição de método de Interface default

```
public interface Andar{  
    public default int getVelocidade() {  
        return 5;  
    }  
}
```

# Métodos de Interface estáticos em Java

## Afinal, o que são métodos de Interface estáticos?

A linguagem de Programação Java também permite a definição de métodos de *Interface* estáticos, i.e, métodos que são acessíveis diretamente na *Interface*, sem a necessidade de uma instância oriunda de uma concretização. Métodos de *Interface* estáticos são definidos com a palavra reservada **static**, e assim como as variáveis de *Interface*, não são “herdados” por outras *Interfaces*, ou concretizações de *Interface*.

# Métodos de Interface estáticos em Java

## Métodos **static** != **default**

Os métodos de *Interface* estáticos não são iguais ao métodos de *Interface default*, a significar que:

1. Métodos de *Interface* estáticos não podem ser sobrescritos, enquanto que métodos de *Interface default* podem;
2. Métodos de *Interface* estáticos são acessados diretamente na *Interface*, enquanto que métodos de *Interface default* devem ser acessados por meio de concretizações de *Interface*.

# Métodos de Interface estáticos em Java - Exemplo [Boyarsky&Selikoff, 2015]

## Exemplo de definição de método de Interface default

```
public interface Pular{  
    static int getAlturaPulo() {  
        return 8;  
    }  
}
```



# Polimorfismo

## Afinal, o que é Polimorfismo?

O conceito de Polimorfismo no contexto da orientação a objetos remete à possibilidade de observação de múltiplas formas e múltiplos comportamentos durante a execução de um programa, e a interação com seus diferentes objetos. Um objeto pode, portanto, assumir diferentes formas a depender do referencial que é utilizado para acessar os seus valores, e executar os seus comportamentos associados. Pode-se dizer que **a hierarquia de classes estabelecida com a aplicação do conceito de Herança é uma das bases de observação do conceito de Polimorfismo.**

# Polimorfismo

## Generalização e Especialização são conceitos de suporte ao Polimorfismo...

Os conceitos de Generalização e Especialização são conceitos de suporte ao polimorfismo, a definir modelos gerais e específicos que podem ser utilizados para criar, acessar, e manipular, de formas distintas, instâncias de objetos.

# Polimorfismo

## Polimorfismo permite...

1. Múltiplas formas de acessar e manipular o mesmo objeto;
2. Múltiplos comportamentos para a mesma assinatura de método, i.e., “o mesmo método”; **[method overriding]**
3. Múltiplos comportamentos para diferentes assinaturas de métodos com o mesmo nome, associados à mesma referência. **[method overloading]**

# Pense em Polimorfismo como...



Imagem: Pixabay (<https://pixabay.com>)

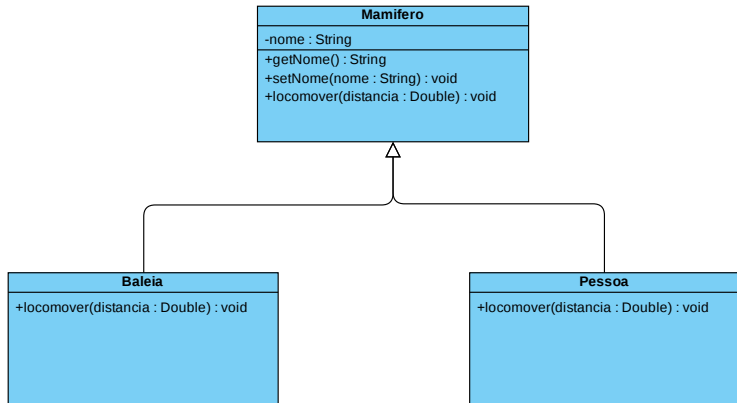
Um evento à fantasia, onde pessoas (consideradas objetos da mesma hierarquia) podem assumir papéis e/ou comportamentos distintos, a depender do referencial, do momento, e da interação observada.

# Polimorfismo - Sobrescrita de Métodos

## Sobrescrita de métodos (method overriding)

A **sobrescrita de métodos** (*method overriding*) é um conceito importante do Polimorfismo, o qual permite definir um novo comportamento para um método definido/especificado em um elemento de um nível superior da hierarquia.

# Polimorfismo - Exemplo de Sobrescrita de Métodos



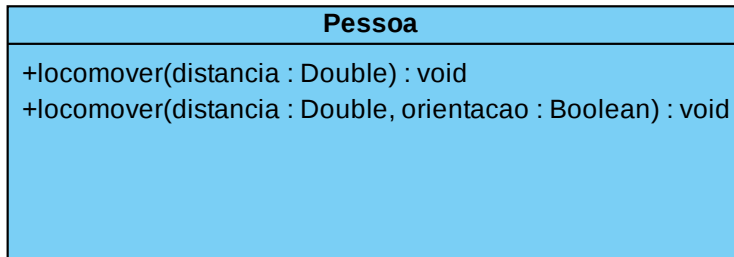
No exemplo apresentado, ambas as classes Baleia e Pessoa sobrescrevem o método `locomover()`, a possuir comportamentos distintos em cada uma das classes apresentadas.

# Polimorfismo - Sobrecarga de Métodos

## Sobrecarga de métodos - method overloading

A **sobrecarga de métodos** (*method overloading*) é um conceito que permite especificar diferentes comportamentos para um método de mesmo nome, porém, com assinatura distinta de assinaturas existentes. A sobrecarga de métodos é realizada a partir de um mesmo referencial, a indicar diferentes formas de execução de comportamentos ditos “similares”.

# Polimorfismo - Exemplo de Sobrecarga de Métodos



No exemplo apresentado, o método *locomover()* da classe Pessoa possui duas assinaturas, a especificar comportamentos similares, porém distintos. Estes comportamentos são executados com a invocação dos diferentes métodos.



# Bibliografia



GOSLING, J. “**Java: An Overview**”. *White paper*. February, 1995.



BOYARSKY, J. and SELIKOFF, S. “**Oracle Certified Associate Java SE 8 Programmer I: Study Guide**”. Sybex. Indianapolis, Indiana, USA. 2015.



BOYARSKY, J. and SELIKOFF, S. “**Oracle Certified Associate Java SE 8 Programmer II: Study Guide**”. Sybex. Indianapolis, Indiana, USA. 2016.

# Bibliografia (Continuação)

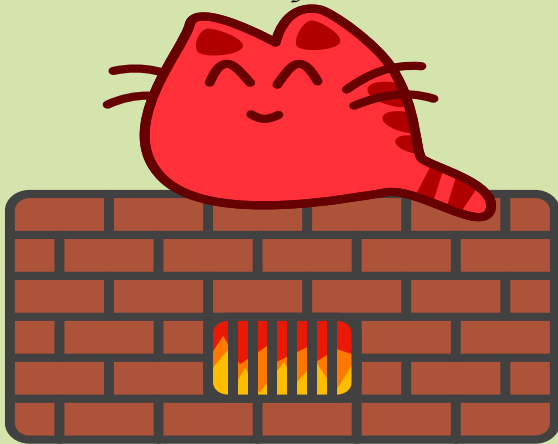


OMG “**OMG Unified Modeling Language (OMG UML) - Version 2.5.1**”. Object Management Group (OMG).

Dezembro, 2017. Disponível em:

<https://www.omg.org/spec/UML/2.5.1/PDF>. Acesso em:  
21 Jun. 2021.

*That's it folks!*



*Thank you for your attention!*