

[< VOLTAR](#)

# Herança e Polimorfismo

Em Java temos à disposição recursos muito úteis para reaproveitamento de código, que minimizam o esforço da programação e otimizam nosso tempo e, tudo isso, por conta da orientação a objetos. Estamos falando da herança, quando uma classe recebe características e comportamentos de outras e do polimorfismo, quando uma classe tem a capacidade de assumir várias formas. Você aprenderá aqui os conceitos de herança e polimorfismo e sairá programando de uma forma bastante avançada.

## NESTE TÓPICO

- > Herança
- > Polimorfismo
- > Sobrecarga
- > Resumo da aula:
- > Referências



## Herança



O conceito de herança em Java é muito importante, mas antes de programarmos, o que é a herança?

Bem, no mundo real herança significa receber algo. E em Java, o conceito é parecido: Herança é quando uma classe possui as mesmas características e comportamentos de outra classe, **mais** (+) as suas próprias, ou seja, uma classe herda de outra quando queremos que ela seja exatamente como essa outra, mas com suas próprias características.

Imagine, por exemplo, a seguinte situação: Você foi contratado para desenvolver um sistema acadêmico de uma importante universidade. Uma das funcionalidades deste sistema é um cadastro de professores e alunos.

Os professores possuem suas próprias características, como matrícula, data de contratação, salário etc., assim como os alunos que possuem suas próprias características, como registro acadêmico (RA), data de ingresso, nota no vestibular, curso etc. Ambos, professores e alunos, são seres humanos, certo? Isso quer dizer que um aluno ou um professor é uma pessoa. Isso quer dizer, ainda, que eles possuem características que são

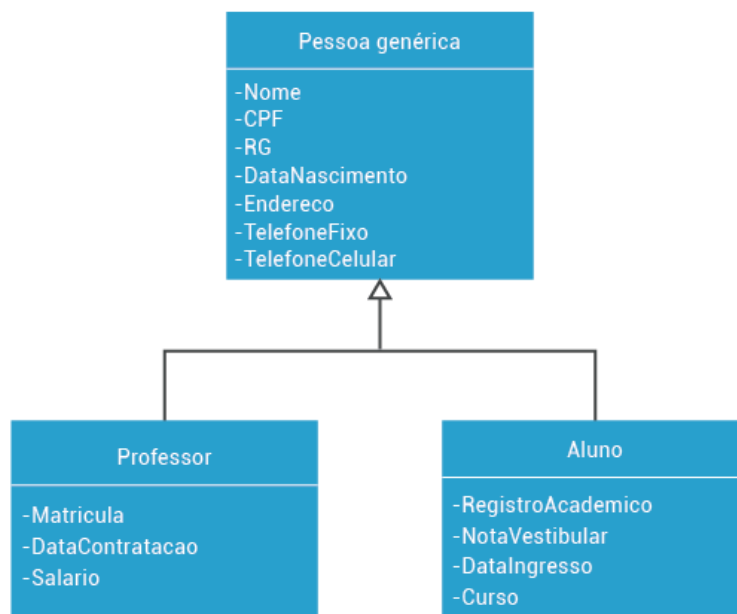
comuns, por exemplo tanto alunos quanto professores possuem um nome, um endereço de correspondência, um (ou mais) telefone(s), um CPF, um RG etc.

Resumindo: neste cenário, um aluno é uma pessoa. Um professor é uma pessoa. Ambos compartilham características, mas possuem, também, as suas próprias.

No mundo sem herança, para cada classe implementada, você teria de implementar, também, todas as características de cada um, inclusive as que eles compartilham. Seria muito código-fonte para resolver o problema e a manutenção deste código pode ser caótica. Mas nem tudo está perdido. A herança em Java existe e pode te ajudar muito nessa hora.

Lembre-se que, neste cenário, todo mundo é uma pessoa e pode, portanto, receber as características de uma pessoa genérica, além de suas próprias características.

A grande “sacada” da herança em Java é que você pode implementar uma única classe com as características que as demais irão compartilhar. A imagem abaixo mostra melhor essa ideia sendo aplicada neste cenário:



Representação visual do exemplo de herança de atributos

Note, na imagem acima, que conforme mencionado, as classes "Professor" e "Aluno" possuem suas próprias características, que são herdadas da classe "PessoaGenerica". Isso, na hora de codificar, irá fornecer um código bem mais limpo e fácil de entender.

Quando o código estiver sendo executado, será possível criarmos objetos dessas classes (Aluno e Professor) diretamente, sem a necessidade de instanciar a classe "PessoaGenerica", pois um objeto da classe "Professor" ou da classe "Aluno" assume automaticamente as características de "PessoaGenerica". Dizemos em "Javanês" que um objeto "Aluno", por exemplo, é ao mesmo tempo um objeto "PessoaGenerica".

Para implementar a herança em Java, é preciso implementar a classe "mãe" primeiro, pois não se pode herdar atributos de uma classe sem que ela exista, ou seja, você não irá conseguir criar um aluno ou um professor sem que exista, primeiro, a classe "PessoaGenerica".

A implementação da classe "PessoaGenerica" não possui nenhuma novidade de codificação, ou seja, é uma classe como qualquer outra. Um possível código para a classe "PessoaGenerica" pode ser visto abaixo:

```
1. public class PessoaGenerica {
2.
3.     //Para facilitar, vamos trabalhar com Strings aqui
4.     String nome, RG, cpf, endereco, telFixo, telCelular, dataNascimento;
5.
6.     //método simples para montar e retornar uma String com os dados da Pessoa
7.     public String retornaDados() {
8.         String dados = "";
9.         dados += "Nome: " + nome + "\n";
10.        dados += "\tRG: " + RG + "\n";
11.        dados += "\tCPF: " + String.valueOf(cpf) + "\n";
12.        dados += "\tTelefone fixo: " + telFixo + "\n";
13.        dados += "\tTelefone cel.: " + telCelular + "\n";
14.
15.        return dados;
16.    }
17. }
```

A diferença na implementação ocorre nas classes que herdam os atributos, neste caso, "Professor" e "Aluno". Precisaremos, na definição da classe, utilizar a palavra reservada **"extends"**. Só isso!



## BOAS PRÁTICAS DE PROGRAMAÇÃO

Para boas práticas de programação, cada uma das classes abaixo deve ser um arquivo novo no projeto!

Uma possível implementação da classe "Professor", pode ser vista abaixo:

```
1. public class Professor extends PessoaGenerica {    //Pronto. Agora a classe professor
2.     é uma PessoaGenerica
3.
4.     //Não é preciso declarar os atributos que já foram declarados na classe mãe, som
5.     ente os próprios:
6.     int matricula;
7.     String dataContrato;
8.     float Salario;
9. }
```

E o mesmo para a classe "Aluno", com suas próprias características:

```

1. public class Aluno extends PessoaGenerica { //Aluno é uma pessoa genérica
2.     //Atributos locais que pertencem apenas ao aluno:
3.     int registroAcademico;
4.     float notaVestibular;
5.     String curso, dataIngresso;
6. }

```

Repare que a classe "Pessoa" possui um método que retorna uma String. Esse método server apenas para montar uma String com os dados da pessoa, independente se ela é Aluno ou não. Mas como assim?

Lembra-se do conceito de herança, onde a classe que herda (a filha) recebe tudo que a classe mãe têm (características e comportamentos - atributos e métodos)? Então, as classes "Pessoa" e "Professor" vão ter, automaticamente, já implementado o método de retornar os dados. Contudo, o acesso a este método é um pouco diferente do convencional.

Veja uma possível implementação da classe principal (que contém o método main) abaixo, onde são criados dois objetos do tipo aluno e dois objetos do tipo professor, e impressos os dados de um de cada.

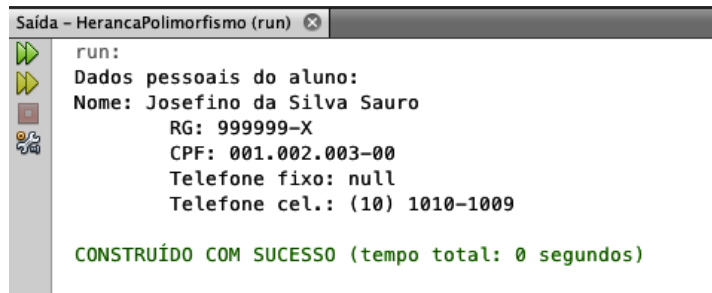
```

1. public class Principal {
2.
3.     public static void main(String args[]) {
4.         Aluno a1 = new Aluno();
5.         Professor p1 = new Professor();
6.
7.         //Veja que podemos acessar os atributos diretamente
8.         //mesmo eles sendo da classe PessoaGenerica.
9.         //Isso ocorre pois o aluno a1 é um Objeto da classe
10.        //aluno que, por sua vez, é filho de "PessoaGenerica"
11.        a1.nome = "Josefino da Silva Sauro";
12.        a1.registroAcademico = 1234567890;
13.        a1.cpf = "001.002.003-00";
14.        a1.RG = "999999-X";
15.        a1.curso = "Bacharelado em Artes Marciais";
16.        a1.dataNascimento = "10/01/1901";
17.        a1.dataIngresso = "01/10/2017";
18.        a1.endereco = "Rua das ruas, 10";
19.        a1.notaVestibular = 8;
20.        a1.telCelular = "(10) 1010-0910";
21.        a1.telCelular = "(10) 1010-1009";
22.
23.        //idem para professor...
24.        p1.nome = "Mestre Yoda";
25.        p1.Salario = 200.00F;
26.        p1.cpf = "000.000.000-10";
27.        p1.dataNascimento = "26/04/1810";
28.        p1.RG = "010101010-Y";
29.        p1.endereco = "Avenida das Galaxias Direita, 01";
30.        p1.matricula = 90020;
31.        p1.telCelular = "(299) 23232";
32.        p1.telFixo = "(299) 1212121";
33.        p1.dataContrato = "01/01/1901";
34.
35.        //Vamos imprimir os dados do aluno a1
36.        System.out.println("Dados pessoais do aluno: ");
37.        System.out.println(a1.retornaDados());
38.    }
39. }

```



E a saída esperada para essa execução deste programa é:



```
run:
Dados pessoais do aluno:
Nome: Josefino da Silva Sauro
RG: 999999-X
CPF: 001.002.003-00
Telefone fixo: null
Telefone cel.: (10) 1010-1009

CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

#### Resultado da execução do exemplo de herança

Note que na saída não foram impressos os dados do aluno, apenas os dados da “PessoaGenerica”, cujo aluno herda. Isso ocorre pois o método que monta estes dados está na classe “PessoaGenerica” e não na classe “Aluno” e quem conhece os objetos de “PessoaGenerica” é o aluno e não ao contrário, ou seja, a classe mãe não conhece suas classes filhas e, portanto, não tem acesso.

Isso quer dizer que se quisermos que sejam recuperados os dados completos do aluno, precisaremos implementar esse método na classe “Aluno”, mas a grande vantagem da herança é que uma parte dele já foi implementado. Para isso, precisaremos usar a palavra reservada *super*.

Para este exemplo, vamos reimplementar a classe “Aluno”, reaproveitando o método de montagem dos dados da “PessoaGenerica” e apenas acrescentando os dados do aluno no método. Essa reimplementação ficará assim (repare na palavra reservada “super” no método de montagem dos dados):



```
1. public class Aluno extends PessoaGenerica { //Não precisa criar uma nova classe, ape
nas refaça a classe "Aluno"
2.
3.     int registroAcademico;
4.     float notaVestibular;
5.     String curso, dataIngresso;
6.
7.     public String retornaDadosAluno() {
8.         String dados;
9.
10.        //Pega os dados da classe herdada
11.        dados = super.retornaDados();
12.
13.        //Acrescenta os dados do aluno:
14.        dados += "\tRegistro Academico: " + registroAcademico + "\n";
15.        dados += "\tNota no Vestibular: " + notaVestibular + "\n";
16.        dados += "\tCurso: " + curso + "\n";
17.        dados += "\tData de ingresso: " + dataIngresso + "\n";
18.
19.        return dados;
20.    }
21. }
```

E na hora de chamar, na classe principal, basta acessa-la da seguinte forma:

```
1. System.out.println(a1.retornaDadosAluno());
```

Para praticar, faça o mesmo com a classe `Professor`, ou seja, um método na classe `Professor` que pega dos dados da classe mãe e os junta com os atributos locais. Acrescente na classe principal instruções para imprimir, também, os dados (completos) do professor que criamos. Aqui, é esperado que seu código tenha o mesmo comportamento da classe `Aluno`, contudo com os dados do `Professor`.

## Polimorfismo

O polimorfismo, em Java, é um recurso poderosíssimo permitido pela maravilha da orientação a objetos. Essa é uma forma bastante avançada de programação e, por isso, é muito importante que você tenha entendido os principais conceitos de orientação a objetos e herança.

O polimorfismo é a capacidade de um objeto comportar-se como outro. Isso mesmo, se for necessário, você pode dar um comportamento diferente para um objeto já implementado.

Uma das formas de aplicarmos o polimorfismo é utilizando a sobrescrita de métodos herdados, ou seja, uma classe mãe possui um método que precisa ser reescrito na classe filha, mas você não quer alterar o nome deste método (o reimplementado, como fizemos no exemplo da herança). Neste caso, você poderá reescrever o método com a sobrescrita. Estamos falando, agora, do `@override`.

Para tal, vamos usar o mesmo exemplo acima, do cenário acadêmico. Note que no código final da classe `aluno`, criamos um método chamado `retornaDadosAluno()`, onde ele pega os dados montados na classe mãe (através do *super*) e acrescenta os seus próprios dados para retornar.

Uma outra forma de reescrever este método é deixando-o com o mesmo nome que na classe mãe. Contudo, isso daria um conflito de nomes, pois a classe `Aluno` está herdando **tudo** de `PessoaGenerica`, certo? É aí onde colocaremos a anotação de reescrita. Exatamente antes da declaração, acrescentaremos a palavra reservada com um `@` na frente, para indicar que aquele método está sendo sobrescrito.

A nova implementação, então, da classe `Aluno`, será assim (não precisa criar uma nova classe, apenas reimplementa-la):



```
1. public class Aluno extends PessoaGenerica {
2.
3.     int registroAcademico;
4.     float notaVestibular;
5.     String curso, dataIngresso;
6.
7.     //Note que o nome do método é exatamente o mesmo da classe mãe
8.
9.     @Override
10.    public String retornaDados() {
11.        String dados;
12.
13.        //Pega os dados da classe herdada
14.        dados = super.retornaDados();
15.
16.        //Acrescenta os dados do aluno:
17.        dados += "\tRegistro Academico: " + registroAcademico + "\n";
18.        dados += "\tNota no Vestibular: " + notaVestibular + "\n";
19.        dados += "\tCurso: " + curso + "\n";
20.        dados += "\tData de ingresso: " + dataIngresso + "\n";
21.
22.        return dados;
23.    }
24. }
```

E, na hora de chama-lo na classe “Principal”, será preciso alterar para o nome original, assim:

```
1. //...
2.     //Na hora de chamar, você chama o mesmo método:
3.     System.out.println(a1.retornaDados());
4. //...
```



O resultado final é exatamente o mesmo mostrado anteriormente, apenas alteramos a forma de implementação (estilo de programação), usando reescrita do método. Isso é fazer um objeto comportar-se como outro, pois um objeto aluno está se comportando como um objeto da classe “PessoaGenerica”.

Para praticar, tente fazer o mesmo com a classe “Professor”, ou seja, tente reescrever o método de retorno de dados do professor usando polimorfismo e imprima os dados no console.

## Sobrecarga

A sobrecarga de métodos e construtores é um tipo de polimorfismo e é extremamente comum na programação orientada a objetos. Em português sobrecarga significa “além da capacidade”, certo? Bem em “Javanês” é mais ou menos a mesma coisa. Ah! A sobrecarga pode ocorrer tanto em métodos da classe como em métodos construtores da classe!

A sobrecarga em Java é a capacidade que linguagem oferece para que você tenha mais de um método com o mesmo nome. Sim, exatamente o que você leu, você pode ter mais de um método com o mesmo nome na mesma classe. Contudo, a **assinatura** destes métodos deve ser diferente.

Mas o que é **assinatura** de um método? A assinatura do método são os parâmetros que ele recebe, ou seja, ele pode não receber nenhum parâmetro ou pode receber quantos parâmetros forem necessários.

Para ficar um pouco mais claro, vamos usar, ainda, o mesmo cenário acadêmico.

Imagine agora que, há duas formas de se criar um objeto “aluno”: Primeiro o cria de forma vazia e depois acrescenta-se os dados OU cria-lo diretamente, já com os dados prontos. Neste caso, precisaremos sobrecarregar o construtor da classe.

Veja como ficará a implementação dessa classe, então:





```

1. public class Aluno extends PessoaGenerica {
2.
3.     int registroAcademico;
4.     float notaVestibular;
5.     String curso, dataIngresso;
6.
7.     //Construtor de Aluno, vazio
8.     public Aluno() {
9.         //Ao criar a instância de um aluno, é preciso criar a instância de
10.         //uma pessoa genérica com o construtor padrão da herança:
11.         super();
12.     }
13.
14.     //Sobrecarregando o construtor, com outra assinatura (já
15.     //recebendo os atributos):
16.     public Aluno(String nome, String RG, String cpf, String endereco,
17.         String telFixo, String telCelular, String dataNascimento,
18.         int registroAcademico, float notaVestibular,
19.         String curso, String dataIngresso) {
20.
21.         super();
22.
23.         //Atribui os dados para o objeto mãe
24.         //Como os nomes dos atributos são iguais, é preciso dizer a qual
25.         //estamos referenciando. Usamos a palavra reservada "this"
26.         //para indicar que estamos usando os parametros locais (recebidos
27.         //nos atributos do construtor):
28.         this.RG = RG;
29.         this.nome = nome;
30.         this.cpf = cpf;
31.         this.endereco = endereco;
32.         this.telCelular = telCelular;
33.         this.telFixo = telFixo;
34.         this.dataNascimento = dataNascimento;
35.
36.         //Agora atribui os valores locais:
37.         this.registroAcademico = registroAcademico;
38.         this.notaVestibular = notaVestibular;
39.         this.curso = curso;
40.         this.dataIngresso = dataIngresso;
41.     }
42.
43.     //Note que o nome do método é exatamente o mesmo da classe mãe
44.     @Override
45.     public String retornaDados() {
46.         String dados;
47.
48.         //Pega os dados da classe herdada
49.         dados = super.retornaDados();
50.
51.         //Acrescenta os dados do aluno:
52.         dados += "\tRegistro Academico: " + registroAcademico + "\n";
53.         dados += "\tNota no Vestibular: " + notaVestibular + "\n";
54.         dados += "\tCurso: " + curso + "\n";
55.         dados += "\tData de ingresso: " + dataIngresso + "\n";
56.
57.         return dados;
58.     }
59. }

```



Já que mexemos no construtor, poderemos refazer, também, a classe principal, que ficará assim:

```
1. public class Principal {
2.
3.     public static void main(String args[]) {
4.         //Agora podemos criar o aluno com os dados diretamente no construtor dele
5.         /*
6.         Lembre-se que os atributos devem ir na mesma ordem do construtor da classe
7.         A ordem é:
8.
9.         String nome, String RG, String cpf, String endereco,
10.        String telFixo, String telCelular, String dataNascimento,
11.        int registroAcademico, float notaVestibular,
12.        String curso, String dataIngresso
13.
14.        */
15.        Aluno a1 = new Aluno("Josefino Sauro", "001101-X", "001.002.003-00", "Rua da
16.        s Ruas, 10",
17.        "(44) 22302-323", "(444) 2323-2323", "10/02/1901",
18.        1902192, 8.0F, "Bacharelado em Artes Marciais", "10/10/2017");
19.
20.        //Vamos imprimir os dados do aluno a1
21.        System.out.println("Dados pessoais do aluno: ");
22.        //Na hora de chamar, você chama o mesmo método:
23.        System.out.println(a1.retornaDados());
24.    }
25. }
```

Para praticar, tente sobrecarregar, também, o construtor de “Professor” e instancia-lo da mesma forma como foi criada a instancia de aluno.

Para reforçar os conceitos apresentados aqui, não deixe de assistir o vídeo abaixo, que mostra a implementação deste cenário, passo a passo, com as explicações sobre herança e polimorfismo.



./videos/370292133.mp4



## Resumo da aula:

Nesta aula, você aprendeu a:

- Que herança é criar classes com as mesmas características e comportamentos de outras
- Que polimorfismo é quando um objeto se comporta como outro

- Dentro de polimorfismo você aprendeu a sobrecarga e a reescrita (*override*)
- Exemplos de herança
- Exemplos de polimorfismo com sobrescrita e sobrecarga

Pronto, chegamos ao final de mais uma aula com importantes conceitos. Pratique bastante, pois estes conceitos são bastante usados no mundo Java e muito importantes. Muita calma! Estes conceitos são sim bastante avançados, não se preocupe se você não entendeu tudo de forma perfeita. Aprender os conceitos é importante, mas a única forma de praticá-los é colocando em prática, ou seja, programando. Então mãos à obra e vamos programar.

## Quiz

Exercício Final

Herança e Polimorfismo



INICIAR ➤

## Referências

Deitei P. e Deitel H., 2010, Java : Como programar, 8ª Edição, Pearson Prentice Hall

Teruel, E. C., 2015, Programação Orientada a Objetos com Java - sem mistérios - 1ª Ed., Editora Uninove

Schildt, H., 2015, Schildt, Java para iniciantes : crie, compile e execute programas Java rapidamente, Bookman



Avalie este tópico



ANTERIOR

Gerando e Interceptando Execuções

Biblioteca

(https://www.uninove.br/conheca-

a-

uninove/biblioteca/sobre-

a-

biblioteca/apresentacao/)

Portal Uninove

(http://www.uninove.br)

Mapa do Site



Índice

Ajuda?

(https://ava.un

Abstração de Curso=)

® Todos os direitos reservados

