

****Herança****

**LINGUAGEM E TÉCNICAS DE PROGRAMAÇÃO (LTP)**

**Thiago Yamamoto**

Versão 1

HISTÓRICO DE REVISÕES

|  |  |  |  |
| --- | --- | --- | --- |
| Versão | Data | Responsável | Descrição |
| 1.0 | 20/10/2014 | Thiago Yamamoto | Versão Inicial do documento |
| 1.1 | 19/4/2015 | Márcia Dias | Revisão 1 |
| 1.2 | 29/11/2016 | Fernanda Meireles | Inserção de imagens |
|  | 03/05/2018 | Cristiane Melo | Revisão, formatação e ajustes para o template |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

FICHA CATALOGRÁFICA   
**[NÃO PREENCHER - PARA USO DO DEPTO DE EAD E BIBLIOTECA]**

A000a Sobrenome, Nome

Título [livro eletrônico] / Nome Sobrenome. -- São Paulo : Fiap, 2016.

x MB ; ePUB

Bibliografia.

ISBN 000-00-00000-00-0

Categoria. 2. Subcategoria. S., Nome. II. Título.

CDU 000.000.00

RESUMO

Herança é um dos principais pilares da programação orientada a objetos. A ideia é criar classes com base em classes existentes, dessa forma a que for criada herdará todos os comportamentos e atributos da existente.

Palavras-chave: Herança, Sobrescrita de Métodos, Polimorfismo e Construtores em Classes Estendidas.

LISTA DE Figuras

[Figura 4.1 – Hierarquia de classes 7](#_Toc513123530)

[Figura 4.2 – Classe com herança 8](#_Toc513123531)

[Figura 4.3 – Atributo da classe conta corrente 9](#_Toc513123532)

[Figura 4.4 – Atributo que armazena o limite do cheque especial 9](#_Toc513123533)

[Figura 4.5 – Método que retorna o saldo disponível da conta corrente 10](#_Toc513123534)

[Figura 4.6 – Operador instanceof 11](#_Toc513123535)

[Figura 4.7 – Operador instanceof 11](#_Toc513123536)

[Figura 4.8 – Operador instanceof 12](#_Toc513123537)

[Figura 4.9 – Método Sobrescrito na classe Conta Corrente 12](#_Toc513123538)

[Figura 4.10 – Método sobrescrito: @Override e super. 13](#_Toc513123539)

[Figura 4.11 – Classe Conta e Conta Corrente 14](#_Toc513123540)

[Figura 4.12 – Construtor da Classe Conta Corrente 15](#_Toc513123541)

[Figura 4.13 – Construtor da classe Conta Corrente com a chamada **super** de forma explícita 15](#_Toc513123542)

[Figura 4.14 – Construtor da classe Conta 15](#_Toc513123543)

[Figura 4.15 – Construtor da classe Conta Corrente invocando o construtor da superclasse 16](#_Toc513123544)

[Figura 4.16 – Instrução super utilizada para chamar o construtor da superclasse 16](#_Toc513123545)

Sumário

[4 herança 7](#_Toc513123900)

[4.1 Introdução 7](#_Toc513123901)

[4.1 Sobrescrita de métodos 12](#_Toc513123902)

# 4 herança

## 4.1 Introdução

Uma classe que herda de outra classe é chamada de subclasse*,* já a classe herdada é chamada de superclasse*.*

A herança é utilizada como forma de reutilizar os atributos e métodos de classes já definidas, permitindo assim derivar uma nova classe mais especializada a partir de outra classe mais genérica existente.

Uma classe só pode ter uma superclasse, ou seja, não é possível ter herança múltipla. Porém, uma classe pode ter um número ilimitado de subclasses*.*

Dessa forma, uma *subclasse*recebe todas as características da superclasse e de todas as outras classes acima dela. A hierarquia de classes se inicia com a classe **Object,** isto é, todas as classes a herdam direta ou indiretamente. Veja a hierarquia de classes:

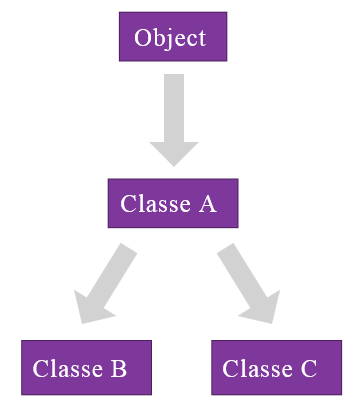


Figura 4.1 – Hierarquia de classes

Fonte: Elaborado pelo autor (2017)

A classe Object é a superclasse da classe A, e esta, por sua vez, é subclasse de Object.

A classe A é a superclasse de B e C, desta forma, as classes B e C recebem todos os atributos e métodos da classe A e da classe Object.

A palavra-chave **extends** é utilizada na declaração de uma classe para determinar a sua superclasse. Caso a classe não tenha essa palavra-chave em sua declaração, a herança que existe é diretamente da classe Object.

Sintaxe:

[public] **class** <subclasse> **extends** <superclasse> {

}

Agora é hora de praticar! Vamos ajustar a classe Conta para que ela possua as subclasses ContaCorrente e ContaPoupanca. A classe Conta herda da classe Object.

A classe ContaCorrente possui o atributo tipo de conta, que define se a conta é básica, especial ou premium. Já a ContaPoupanca não possui esse tipo de definição.

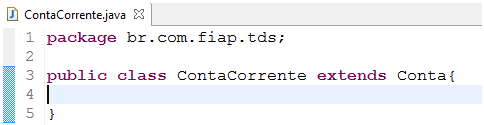


Figura 4.2 – Classe com herança

Fonte: Elaborado pelo autor (2017)

Após definir que a classe ContaCorrente é uma subclasse de Conta, vamos adicionar um atributo para definir o tipo de conta corrente.

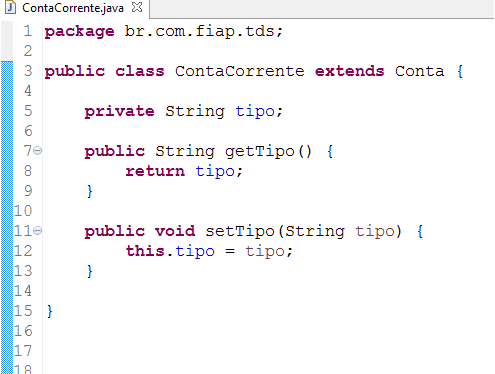


Figura 4.3 – Atributo da classe conta corrente

Fonte: Elaborado pelo autor (2017)

Observe que utilizamos o encapsulamento para proteger o atributo tipo, que define o tipo da conta corrente.

Vamos adicionar outro atributo que será utilizado para armazenar o valor do cheque especial, ou seja, o valor que o cliente pode utilizar junto do saldo da conta.



Figura 4.4 – Atributo que armazena o limite do cheque especial

Fonte: Elaborado pelo autor (2017)

Agora que vimos como adicionar atributos na classe filha, vamos adicionar métodos específicos à ContaCorrente. Uma conta corrente tem o comportamento de retornar o Saldo Disponível, que é a soma do saldo da conta com o limite do cheque especial:

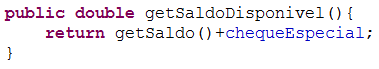


Figura 4.5 – Método que retorna o saldo disponível da conta corrente

Fonte: Elaborado pelo autor (2017)

Note que o método retorna o valor da soma do cheque especial com o saldo da conta. Para acessar o saldo da conta foi necessário utilizar o método *getSaldo()*, pois o atributo **saldo** está definido na classe pai como **private**, por isso não é visível na classe filha.

Uma forma de descobrir se a herança é adequada para as suas classes é seguir a regra do “é um”, que afirma que cada objeto da subclasse é um objeto da superclasse. Por exemplo, uma Conta Corrente é uma Conta, o que significa que a herança deve ser utilizada nesta situação. Naturalmente, o contrário não é verdadeiro, nem toda Conta é uma Conta Corrente.

Dessa forma, você pode utilizar um objeto de uma subclasse sempre que o programa esperar por um objeto da superclasse. Assim, é possível atribuir um objeto do tipo Conta Corrente em uma variável do tipo Conta.

Exemplo:

Conta conta = new Conta();

Conta cc = new ContaCorrente();

As variáveis que armazenam uma referência a um objeto são polimórficas. Isso quer dizer que uma variável de uma superclasse pode receber qualquer objeto de suas subclasses.

Podemos atribuir o objeto que está referenciado na variável **cc** a uma variável do tipo ContaCorrente, para isso é necessário realizar um **cast:**

ContaCorrente c1 = (ContaCorrente) cc;

O cast é forçar um objeto ser de outro tipo em um momento. Neste caso, forçamos o objeto a ser do tipo ContaCorrente para atribuirmos em uma variável do tipo ContaCorrente. O cast é composto pelos parênteses () e a classe que queremos forçar o objeto a se transformar naquele momento.

Observe que se tentarmos realizar o cast e o objeto não for do tipo ou subtipo da classe que queremos forçar, o Java irá lançar a exceção ClassCastException:

ContaCorrente c2 = (ContaCorrente) conta;

A variável **conta** faz referência a um objeto do tipo Conta e não do tipo ContaCorrente. Assim, a exceção será lançada.

Para verificar se o objeto é do tipo de uma classe, podemos utilizar a instrução **instanceof.** Essa instrução retorna *true*caso o objeto a esquerda do operador é do tipo (classe) especificado à direita do operador. Exemplo:

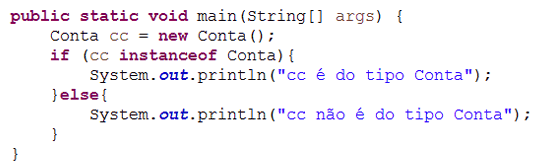


Figura 4.6 – Operador instanceof

Fonte: Elaborado pelo autor (2017)

No exemplo, estamos testando se a variável cc possui um objeto do tipo Conta. Então, qual é o resultado da execução?

O resultado é “cc é do tipo Conta”. E se alteramos o programa para instanciar um objeto do tipo ContaCorrente ao invés do tipo Conta, qual será o resultado da execução?

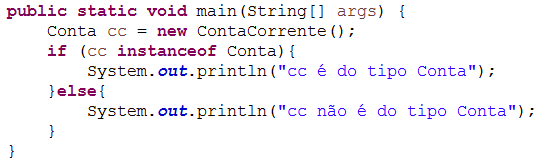


Figura 4.7 – Operador instanceof

Fonte: Elaborado pelo autor (2017)

A resposta é que cc é do tipo conta, pois uma ContaCorrente também é uma Conta.

Mas se alterarmos novamente o programa anterior para ficar dessa forma:

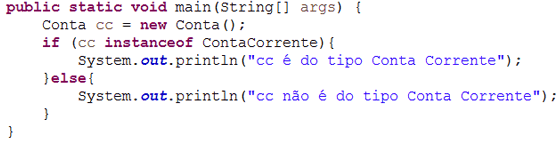


Figura 4.8 – Operador instanceof

Fonte: Elaborado pelo autor (2017)

Agora estamos testando se o objeto da variável cc é do tipo ContaCorrente. A resposta da execução é “cc não é do tipo Conta Corrente”. Pois uma Conta não é necessariamente uma Conta Corrente.

## 4.1 Sobrescrita de métodos

Outra diferença, para o nosso sistema, é que para a conta poupança não há taxa para efetuar um saque. Na conta corrente do tipo básica existe uma taxa para a retirada de dinheiro.

Dessa forma, precisamos sobrescrever o comportamento do método **retirar** na classe Conta Corrente. Sobrescrever um método é redefinir na subclasse um método existente na superclasse. Assim, quando o método retirar for chamado de um objeto do tipo Conta Corrente, o método chamado será o retirar definido na classe Conta Corrente e não da classe Conta.

Para isso, precisamos implementar na classe **Conta Corrente** um método retirar que tenha a mesma assinatura do método retirar da classe **Conta**.

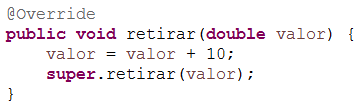


Figura 4.9 – Método Sobrescrito na classe Conta Corrente

Fonte: Elaborado pelo autor (2017)

A anotação **@Override** marca o método, identificando que o método está sobrescrevendo um método de sua superclass.

O método retirar soma a taxa de retirada (10) ao valor a ser subtraído do saldo. Como não temos acesso direto ao saldo e não podemos alterar o seu valor na subclasse (não existe o método setSaldo() na classe Conta), precisamos utilizar o método retirar que está na classe Conta. A palavra **super** é utilizada para referenciar a superclasse, assim a instrução **super.retirar(valor)** está chamando o método retirar que está na classe Conta.

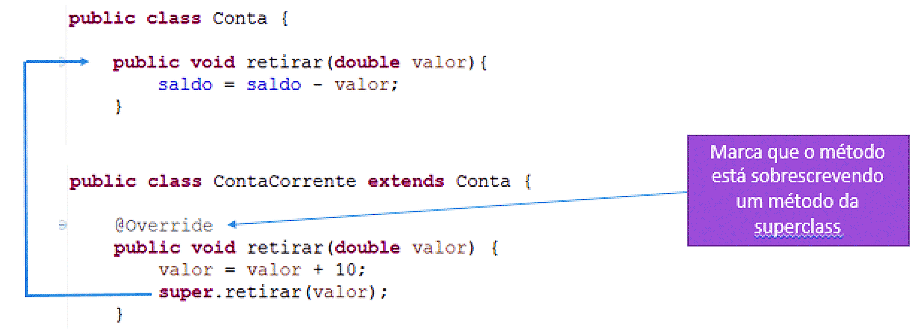


Figura 4.10 – Método sobrescrito: @Override e super.

Fonte: Elaborado pelo autor (2017)

Isso faz parte de um dos pilares da Orientação a Objetos: o **polimorfismo**.

A palavra polimorfismo quer dizer várias formas, na orientação a objetos representa que um objeto pode ser referenciado de várias formas. Quando sobrescrevemos um método na subclasse, o que determina se o método que será chamado é da subclasse ou da superclasse é o tipo de instância do objeto. Por exemplo:

Conta conta = new Conta();

conta.retirar(100);

O método chamado será o definido na classe Conta.

ContaCorrente conta = new ContaCorrente();

conta.retirar(100);

O método chamado será o definido na classe ContaCorrente.

Simples, certo? E no exemplo abaixo? Qual será o método chamado?

Conta conta = new ContaCorrente();

conta.retirar(100);

Neste caso, o método definido na ContaCorrente será invocado. Pois, o objeto que está em conta é do tipo ContaCorrente.

Observe que podemos redefinir o comportamento de uma classe em sua subclasse e assim um objeto pode se comportar de maneira diferente ao invocar um método, dependendo do seu tipo de criação.

Construtores em Classes Estendidas

Os construtores das subclasses **sempre** precisam chamar um construtor da superclasse. E para isso, a instrução **super** é utilizada.

Observe as classes Conta e ContaCorrente:

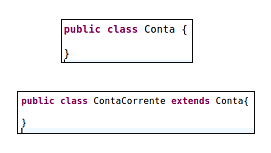


Figura 4.11 – Classe Conta e Conta Corrente

Fonte: Elaborado pelo autor (2017)

Os atributos e métodos foram omitidos para focarmos nos construtores. Essas classes têm construtores?

A resposta é sim. Apesar de não estar definido, elas possuem o construtor padrão (sem argumentos) que é fornecido pelo Java. O construtor padrão chama o construtor da superclasse direta, ou seja, o construtor da classe **ContaCorrente** chama o construtor da classe **Conta** e o construtor da classe **Conta** chama o construtor da classe **Object**.

Vamos definir um construtor para a classe ContaCorrente. Esse construtor recebe como parâmetro o valor do tipo da Conta.

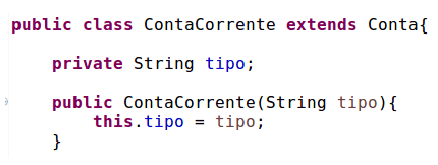


Figura 4.12 – Construtor da Classe Conta Corrente

Fonte: Elaborado pelo autor (2017)

Agora temos um construtor para a classe ContaCorrente. Esse construtor chama o construtor de sua superclasse?

A resposta é sim. Isso é feito automaticamente pelo Java, pois a classe Conta possui o construtor padrão. Então o código abaixo, com a instrução **super( )** na primeira linha do construtor é redundante, pois o Java irá fornecer a instrução caso não seja definido.

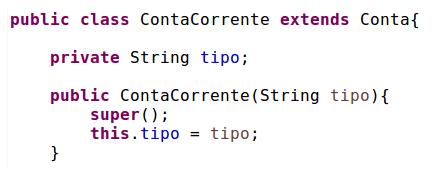


Figura 4.13 – Construtor da classe Conta Corrente com a chamada **super** de forma explícita

Fonte: Elaborado pelo autor (2017)

Outras duas regras dos construtores são:

1) Não são herdados.

2) A chamada do construtor da superclasse deve ser sempre feita na primeira linha do construtor da subclasse.

Dessa forma, se implementarmos um construtor que recebe um parâmetro na classe Conta, a classe ContaCorrente não vai herdar esse construtor:

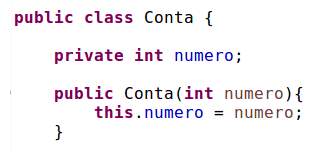


Figura 4.14 – Construtor da classe Conta

Fonte: Elaborado pelo autor (2017)

A partir do momento em que implementamos esse construtor na classe Conta, a classe ContaCorrente começa a exibir um erro de compilação. Isso acontece, pois o construtor da classe ContaCorrente deve chamar o construtor da classe Conta, porém agora, a classe Conta tem somente o construtor que recebe um parâmetro do tipo **int**.

Para consertar o erro, devemos fazer com que o construtor da classe ContaCorrente chame o construtor definido na classe Conta. Para isso, utilizamos a instrução **super**, passando o parâmetro do tipo inteiro:

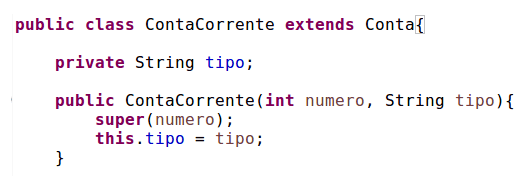


Figura 4.15 – Construtor da classe Conta Corrente invocando o construtor da superclasse

Fonte: Elaborado pelo autor (2017)

Dessa forma, o construtor da subclasse chama o construtor da superclasse:

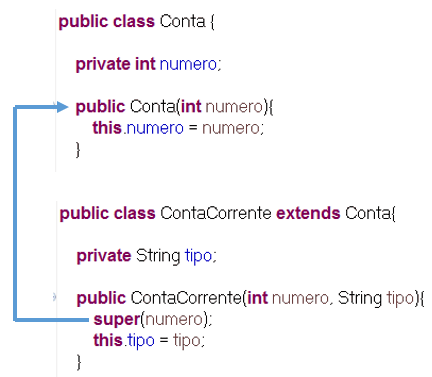


Figura 4.16 – Instrução super utilizada para chamar o construtor da superclasse

Fonte: Elaborado pelo autor (2017)

Resumindo, se o **super** não for chamado, o compilador acrescenta uma chamada ao construtor padrão: super( );

Se não existir um construtor padrão na superclasse, haverá um erro de compilação e será necessário chamar explicitamente a instrução **super**, passando os parâmetros do construtor da superclasse. REFERÊNCIAS

BARNES, David J. **Programação Orientada a Objetos com Java:** uma introdução prática utilizando Blue J. São Paulo: Pearson, 2004.

CADENHEAD, Rogers; LEMAY, Laura. **Aprenda em 21 dias Java 2 Professional Reference.** 5.ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL, Harvey. **Java Como Programar**. 8.ed. São Paulo. Pearson, 2010.

HORSTMANN, Cay; CORNELL, Gary. **Core Java**: Volume I. Fundamentos. 8.ed. São Paulo: Pearson 2009.

SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java.** Rio de Janeiro: Alta Books, 2010.