

PROGRAMAÇÃO ORIENTADA A OBJETOS

CAPÍTULO 3 – CRIANDO CLASSES A PARTIR DE CLASSES JÁ EXISTENTES, ISSO É POSSÍVEL?

Handerson Bezerra Medeiros

Introdução

Sabemos que podemos melhorar a segurança em nosso código, quando encapsulamos algumas informações dentro do programa. Mas será que existe alguma forma de reutilizar nosso código? A programação orientada a objeto vem de um paradigma no qual, a ideia central encontra-se na abstração do problema e, para garantirmos isso, temos alguns mecanismos que nos auxiliam nesta árdua missão.

O que você conhece sobre herança? Bem, não estamos falando sobre herança que recebemos das pessoas. Mas, se pensarmos bem, a ideia de herança, em programação orientada a objetos, acaba sendo bem semelhante. Recebemos uma herança, quando temos algum parentesco com aquela pessoa, e na programação, podemos criar classes mãe que terão seus filhos herdeiros. Interessante, né?

Neste capítulo, vamos aprender a deixar nosso código mais abstrato. Veremos o que significa herança e como podemos fazer isso. E por que trabalhar com a ideia de herança? O que isso contribui, na abstração do código?

Essas respostas vão ser trabalhadas, ao longo do texto, que também vai abordar as vantagens e desvantagens de sobrecarga e reescrita. Vamos conhecer e compreender o significado de polimorfismo e implementar códigos utilizando herança, de modo a compreender o que significa abstração de código.

Vamos estudar esse conteúdo a partir de agora, acompanhe!

3.1 Entendendo sobre herança

Para começar a entender sobre herança, vamos retomar o significado da palavra. Sabemos que herança significa herdar (transmitir) informações para outros elementos em código. É justamente esse conceito de herança que faz com que a programação orientada a objetos se torne única. Há vantagens e desvantagens na criação de heranças, mas quem decide se é o momento de usar ou não, é você, então, é necessário ter todo o conhecimento que te ajude a tomar essa decisão.

Então, como podemos criar herança em nossos códigos? Vamos entender.

3.1.1 Conceitos de herança

Em sua definição, quando utilizamos herança, significa utilizar de um mecanismo para construir uma classe com seus atributos e métodos, o qual chamamos de classe mãe. Essa classe mãe permite que outras classes, chamadas classe filhos, usem (herdem) todos os seus atributos e métodos. Ou seja, a classe mãe permite determinar outra classe, utilizando a definição de uma classe já implementada em nosso código.

Será que criar herança é útil? Vamos analisar o seguinte cenário da figura abaixo que retrata uma abstração do universo animal. Temos uma classe que chamamos de *Animal*, sabemos que podemos ter opções diferentes de animais. Por exemplo, animais aquáticos, terrestres e aéreos.

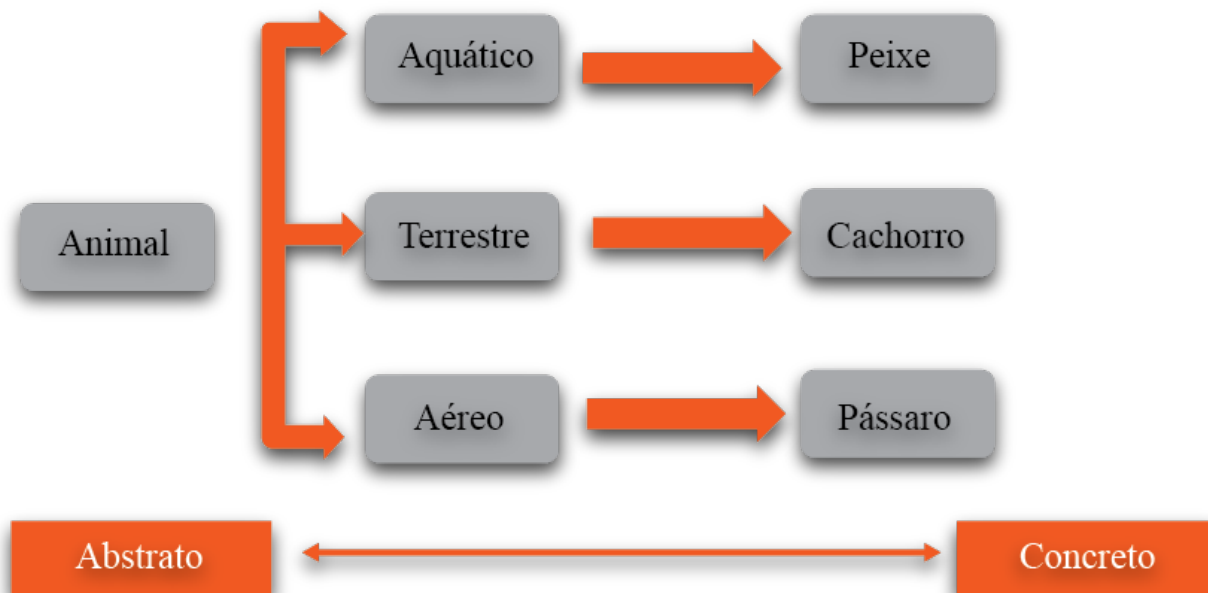


Figura 1 - Exemplo de um modelo de herança, na qual as classes aquático, terrestre e aéreo, herdam da classe animal.

Fonte: Elaborada pelo autor, 2018

Considerando como classes, as abstrações Animal, Aquático, Terrestre e Aéreo, podemos citar como classes dessas classes como sendo *peixe*, *cachorro* e *pássaro*. Analisando com mais detalhes, percebemos que todos possuem um atributo (característica) em comum, por exemplo, tipo de alimentação. Esse atributo especifica o tipo de alimentação daquele animal (aquático, terrestre ou aéreo), se será carnívoro, herbívoro ou onívoro.

Continuando, podemos observar que existe um atributo específico para os animais terrestres, velocidade de corrida. E ainda temos informações mais específicas, por exemplo, na classe *cachorro*, como raça, cor e porte.

Como podemos verificar pela ilustração acima, as classes *aquático*, *terrestre* e *aéreo* herdam da classe *animal*, tornando, assim, a classe *animal*, a classe mãe das demais. Já a classe *peixe* herda da classe *aquático*, como a classe *cachorro* herda da classe *terrestre* e por fim, a classe *pássaro* herda da classe *aéreo*.

Parece bem mais fácil agora! Você pode perceber que um dos aspectos da herança é que todas as outras classes que herdam de outra, consequentemente acabam herdando seus atributos e comportamentos. Ou seja, como as classes *aquático*, *terrestre* e *aéreo* herdam da classe *animal*, todas elas possuem o atributo tipo de alimentação. Como por consequência, também temos a classe *cachorro* herdando o atributo velocidade de corrida da classe *terrestre*.

Então, como podemos modelar isso? Sabemos que a classe *terrestre* herda tudo da classe *animal*, dessa forma podemos informar que a classe *terrestre*, na verdade, é uma extensão da classe *animal*. Para adicionarmos essa informação em código, fazemos o uso da palavra-chave *extends*.

```

public classe Animal{
//Atributos da classe mãe
private String tipo_alimentacao;
//Métodos
}
public class Terrestre extends Animal{
//Atributos exclusivos da classe terrestre
private double velocidade_da_corrida;
//Métodos
}
  
```

Quando implementamos dessa forma, fazemos com que a classe *Terrestre* herde todos os atributos e métodos da sua classe mãe, no caso, a classe *Animal*. Mas a classe mãe está declarando seus atributos e métodos como *private*, ainda é possível acessar?

Sim, pois informamos que existe uma extensão daquela classe. A diferença é que por mais que eu possa acessar essas informações, a classe que herdou não pode acessar essas informações de maneira direta. Mas não tem problemas, pois já aprendemos que podemos usar os métodos *get* e *set* para isso.

VOCÊ QUER LER?



Quando aprendemos herança, podemos cometer o erro de esquecer de trabalhar com encapsulamento. Um dos estudiosos de Java, Martin Fowler (2006), afirma que, na maioria das linguagens de programação, o uso da herança pode comprometer seriamente os benefícios que o encapsulamento nos oferece em código. Para entender essa discussão, leia: <<https://martinfowler.com/bliki/DesignedInheritance.html>>.

De acordo com Manzano e Costa Jr. (2014), precisamos entender algumas nomenclaturas que envolvem herança. Podemos chamar de superclasse, mãe ou tipo, as classes que fornecem a herança. Já as classes que herdaram de outras classes, podem ser chamadas de subclasse, filha ou subtipo. Chamamos de ancestral, a classe que aparece na hierarquia de classes, em uma posição acima da classe mãe. Já descendente significa toda classe que aparecer abaixo da classe mãe, em sua hierarquia. Podemos chamar o topo da hierarquia de raiz e uma classe sem filhos de folha. Lembrando sempre que as classes filhas jamais podem remover atributos e métodos da sua classe mãe. Mas será que utilizar do conceito de herança só tem vantagens em nosso código? Vamos analisar melhor para compreender suas vantagens e desvantagens. Como notamos, um dos principais objetivos da herança é juntar tudo que for igual e isolar aquilo que é diferente, criando, assim, uma árvore de hierarquia. Porém, quando utilizamos herança, enfraquecemos o conceito de encapsulamento, pois as classes mães e filhas mantêm uma extensão (dependência) uma da outra. Consequentemente, isso gera um forte acoplamento entre essas classes, e caso algo seja mudado na classe mãe, imediatamente as classes filhas também sofrem com essa alteração. Quando deixamos isso acontecer, significa quebrar o paradigma principal da programação orientada a objetos, para o qual devemos manter um baixo acoplamento entre as classes.

VOCÊ O CONHECE?



Já ouviu falar em James Gosling? Não? Mas tenho certeza que já ouviu falar na linguagem de programação Java, né? James Gosling é um programador canadense, considerado o Pai da linguagem de programação Java. Cientista da computação, James, enquanto realizava seu doutorado, inventou a linguagem de programação. Foi ele quem criou o projeto inicial e implementou o compilador e sua máquina virtual (SCHOFIELD, 2007).

E agora, quando devemos usar herança? Podemos utilizar as vantagens do conceito de herança quando sabemos que as classes filhas jamais precisarão ser objetos de outra classe.

Para melhorar nosso entendimento, podemos fazer o uso da relação “é um”, para decidirmos criar herança em nosso programa. Por exemplo, quando utilizarmos a relação “é um” enxergamos um relacionamento desejável na herança, como podemos ver nos exemplos, “terrestre é um animal”.

3.1.2 Herança múltipla

Vamos analisar um outro cenário, nosso programa deverá ser responsável pelos funcionários de uma escola. Aplicando o conhecimento que já vimos, podemos elaborar uma estrutura de herança, na qual a classe mãe será *Funcionario*, responsável pelos atributos e métodos comuns a todos os outros filhos. E como classes filhas podemos ter, *Coordenador*, *Professor*, *Auxiliar*. Criando nossas classes, por se tratar de uma herança, nossas classes filhas devem estender da classe mãe.

```
public classe Funcionario{
//Atributos da classe mãe
private int matricula;
private String nome;
//Métodos
}
public class Coordenador extends Funcionario{
//Atributos exclusivos da classe coordenador
private String curso_vinculado;
//Métodos
}
public class Professor extends Funcionario{
//Atributos exclusivos da classe professor
private String disciplina;
//Métodos
}
public class Auxiliar extends Funcionario{
//Atributos exclusivos da classe Auxiliar
private String departamento;
//Métodos
}
```

Como percebemos, utilizamos da palavra-chave *extends* para sinalizar que as classes filhas estendem de uma classe mãe. Quando usamos desse método, estamos criando uma herança simples. Pois, cada classe filha tem apenas uma classe mãe. E se quisermos que uma classe filha herde mais de uma classe mãe? Será possível?

E se o coordenador também for um professor? Será que precisamos duplicar informações para cadastrá-lo duas vezes? Uma como coordenador e outra como professor? Para resolvermos esse problema, podemos utilizar o conceito de herança múltipla. Utilizando esse método, fazemos com que uma classe filha herde mais de uma classe mãe, ou seja, ela herda os atributos e métodos combinados de todas as suas classes mãe (MARINHO, 2016).

Porém, a linguagem de programação Java não permite que seja utilizada herança múltipla. Em Java, se deixarmos que uma classe filha estenda de duas classes mãe, significa lidar com duas classes mãe que possuem a mesma estrutura, só que com implementações distintas. Na hora da execução, a classe filha não vai saber qual método de suas classes mãe deve utilizar.

Então, no nosso caso de o Coordenador também ser professor, significa que é necessário duplicar informações? Não, apesar de o Java não suportar herança múltipla, ele permite resolvermos essa situação de herança múltipla, fazendo o uso de interfaces. Mas, vamos entender, agora, como funciona a implementação da herança.

3.1.3 Implementação da herança

Vamos colocar em ação todos esses conceitos que aprendemos sobre herança. Como assim uma classe herda atributos e métodos de outra classe? Como isso acontece durante a execução do programa? Como modelamos essas classes? Vamos entender tudo isso agora.

Quando criamos um objeto, sabemos que ele é uma instância (representado) por uma classe concreta, que armazena atributos e métodos que podem ser utilizados por aquele objeto. Para entendermos o funcionamento de herança, vamos analisar a figura abaixo que representa em um conceito geral, a funcionalidade de herança que aprendemos.



Figura 2 - Exemplo de um modelo de herança durante a execução do programa, quando o objeto deverá utilizar os atributos da classe mãe.

Fonte: Elaborada pelo autor, 2018.

Analisando o cenário ilustrado na figura acima, vamos supor que o objeto deseja identificar o seu valor do atributo X. Como isso vai ocorrer? Para conseguir essa informação, o objeto deverá percorrer sua árvore hierárquica realizando consultas até achar a classe mãe que armazena esse atributo. Nesse caso, o objeto deverá consultar a classe filha 2.2 e não obterá retorno, dessa forma, sobe mais um nível hierárquico e consulta a classe filha 2, e ainda sem sucesso, ele consulta a classe mãe, encontrando, assim, o valor desejado. Podemos perceber pela imagem abaixo, o caminho percorrido pelo objeto.

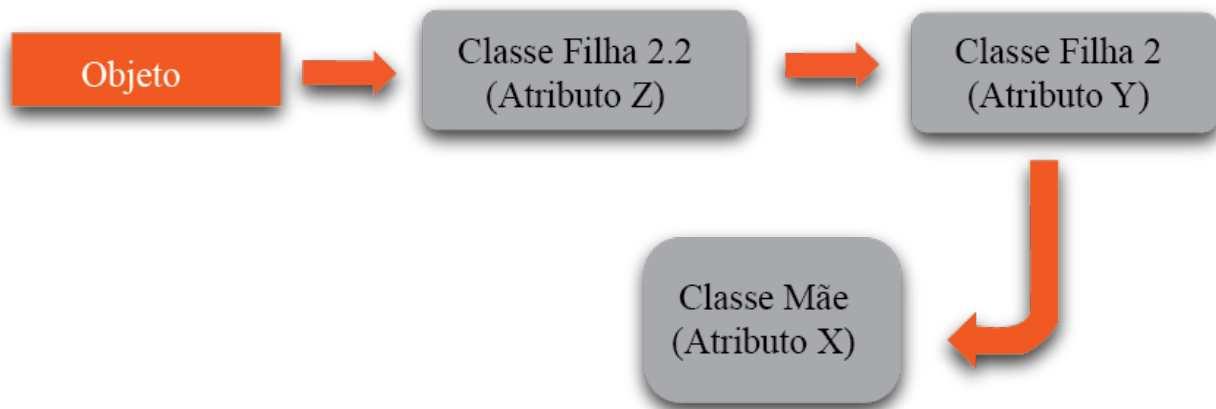


Figura 3 - Caminho percorrido pelo objeto para acessar o método especificado na classe mãe.

Fonte: Elaborada pelo autor, 2018.

Vamos implementar um exemplo mais prático? Uma loja de informática precisa de um programa que armazene produtos da sua loja. Os produtos existentes em loja são *Desktop*, *Tablet* e celular. Pensando como desenvolvedor, já enxergamos que todos os produtos são computadores e consequentemente compartilham de atributos e comportamentos iguais. Primeiro, precisamos identificar e criar uma classe mãe que irá ser responsável pela estrutura semelhante entre as classes filhas, no nosso caso, vamos chamar a classe mãe de *Computador*.

```

public classe Computador{
//Atributos da classe mãe
private int codigoDeBarra;
private int memoria;
private String processador;
//Métodos
public void ligar(){
System.out.println("Máquina ligada!");
}
public void desligar(){
System.out.println("Máquina desligada!");
}
}
  
```

Na nossa classe mãe *Computador*, colocamos os atributos (*codigoDeBarra* e *memoria*) e os métodos (*ligar* e *desligar*) que serão compartilhados para as classes filhas. Agora, criamos as classes filhas *Desktop*, *Tablet* e *Celular* que estendem da classe *Computador*, informando apenas os atributos e métodos que os diferem.

```

public class Desktop extends Computador{
//Atributos exclusivos da classe Desktop
private String placaDeVideo;
//Métodos
public void ligarVideo(){
System.out.println("Placa de vídeo ativada");
}
}

public class Tablet extends Computador{
//Atributos exclusivos da classe Tablet
private String cobertura;
//Métodos
  
```

```

public void iniciarConexao(){
System.out.println("Conexão ativada");
}
}
public class Celular extends Computador{
//Atributos exclusivos da classe Celular
private boolean ligacao;
//Métodos
public void fazerLigacao(){
System.out.println("Ligando!");
}
}

```

Como as classes filhas herdam (estende) a classe mãe, significa dizer que os atributos e métodos da classe *Computador* também são utilizadas nas classes *Desktop*, *Tablet* e *Celular*. Aprendemos em aulas anteriores, que o uso de construtores nas classes é importante para instanciarmos um novo objeto. Como podemos utilizar construtores com herança?

Vamos refazer nossa classe mãe computador e adicionar o seu método construtor.

```

public classe Computador{
//Atributos da classe mãe
private int codigoDeBarra;
private int memoria;
private String processador;
//Construtor
public Computador(int codigoDeBarra, int memoria, String processador){
this.codigoDeBarra = codigoDeBarra;
this.memoria = memoria;
this.processador = processador;
}
//Métodos
....
}

```

Sabendo que todas as classes devem ter seu método construtor, como adicionaremos os construtores nas classes filhas, de forma a utilizar o construtor da classe mãe? Para isso, os construtores das classes filhas devem acessar o construtor da classe mãe, utilizando o método *super*. Vamos implementar.

```

public class Desktop extends Computador{
//Atributos exclusivos da classe Desktop
private String placaDeVideo;
//Construtor
public Desktop(int codigo, int memoria, String processador, String placaDeVideo){
super(codigo, memoria, processador);
this.placaDeVideo = placaDeVideo ;
}
//Métodos
public String getPlacaDeVideo(){
return this.placaDeVideo;
}
public void setPlacaDeVideo(String placaDeVideo){
this.placaDeVideo = placaDeVideo;
}
}

```



```

}
public class Tablet extends Computador{
//Atributos exclusivos da classe Tablet
private String cobertura;
//Construtor
public Tablet(int codigo, int memoria, String processador, String cobertura){
super(codigo, memoria, processador);
this.cobertura = cobertura;
}
//Métodos
public String getCobertura(){
return this.cobertura;
}
public void setCobertura(String cobertura){
this.cobertura = cobertura;
}
}
public class Celular extends Computador{
//Atributos exclusivos da classe Celular
private boolean ligacao;
//Construtor
public Celular(int codigo, int memoria, String processador, boolean ligacao){
super(codigo, memoria, processador);
this.ligacao = ligacao;
}
//Métodos
public boolean getLigacao(){
return this.ligacao;
}
public void setLigacao(boolean ligacao){
this.ligacao = ligacao;
}
}

```

Mesmo que nossa classe mãe possua mais de um construtor, ainda assim, podemos utilizar o *super*. Pois devido à sobrecarga de métodos, a linguagem Java consegue distinguir qual construtor está sendo utilizado. De modo a entender uma possível saída desse programa, vamos instanciar objetos de cada classe filha e atribuir informações sobre eles. E, por fim, solicitar que o programa imprima essas informações ao usuário. Com o intuito de agilizarmos nossa simulação, entendemos neste caso que os métodos *get* e *set* das classes foram desenvolvidas. Vamos lá?

```

public static void main(String[] args) {
//Instanciando objetos de classe filha
Desktop desk = new Desktop(1001, 4, "intel", "amd");
Tablet tab = new Tablet(1100, 2, "dual", "3G");
Celular cel = new Celular(1011, 16, "asus", true);
//Imprimir informações do objeto desk
System.out.println("Informações do Desktop \n");
System.out.println("Código de barra: " + desk.getCodigoDeBarra());
System.out.println("Memória: " + desk.getMemoria());
System.out.println("Processador: " + desk.getProcessador());
}

```

```

System.out.println("Placa de vídeo: " + desk.getPlacaDeVideo ());
//Imprimir informações do objeto tab
System.out.println("Informações do Tablet \n");
System.out.println("Código de barra: " + tab.getCodigoDeBarra());
System.out.println("Memória: " + tab.getMemoria());
System.out.println("Processador: " + tab.getProcessador());
System.out.println("Cobertura: " + tab.getCobertura());
//Imprimir informações do objeto cel
System.out.println("Informações do Celular \n");
System.out.println("Código de barra: " + cel.getCodigoDeBarra());
System.out.println("Memória: " + cel.getMemoria());
System.out.println("Processador: " + cel.getProcessador());
System.out.println("Status da ligação: " + cel.getLigacao());
}

```

Como seria a saída desse nosso programa? Como saída, teríamos a impressão da figura abaixo, que representa o retorno do nosso programa para o usuário.

Informações do Desktop

Código de barra: 1001

Memória: 4

Processador: intel

Placa de Video: amd

Informações do Tablet

Código de barra: 1100

Memória: 2

Processador: dual

Cobertura: 3G

Informações do Celular

Código de barra: 1011

Memória: 16

Processador: asus

Status da ligação: true

Figura 4 - Retorno de impressão ao usuário do programa de exemplo, contendo informações dos objetos instanciados.

Fonte: Elaborada pelo autor, 2018.

Informações do Desktop

Código de barra: 1001

Memoria: 4

Processador: intel

Placa de Video: amd

Informações do Tablet

Código de barra: 1100

Memoria: 2

Processador: dual

Cobertura: 3G

Informações do Celular

Código de barra: 1011

Memoria: 16

Processador: asus

Status da ligação: true

Legal, né? Quando utilizamos todo o conceito de herança, conseguimos fazer um reaproveitamento de código e modelar nosso programa de maneira mais eficaz. Sabendo que as classes filhas herdam todos os atributos e métodos da classe mãe, e se existir um método que foi herdado, que não faz nenhum sentido para a classe filha? Como podemos proceder? Vamos descobrir?

3.2 Sobrecarga ou sobrescrita, qual é a diferença?

Em alguns casos, podemos notar que os métodos existentes na classe mãe, talvez não se adéquem a todas as suas classes filhas. Tornando assim, aquele método sem sentido para a sua classe filha. E agora, será que ainda podemos definir essa classe como filha? Não seria melhor deixar ela separada?

Vamos entender, agora, sobre o conceito de sobrescrita, que irá nos ajudar a modificar métodos da classe mãe, que não fazem um bom proveito em sua classe filha.

3.2.1 Conceitos

Vamos analisar a seguinte situação. Nosso programa tem uma classe mãe chamada *Produto*, nela constam os atributos e métodos que serão compartilhados pelas classes filhas. Implementando, poderíamos desenvolver a classe da seguinte maneira: criar atributos de código e valor do produto, como também um método responsável por fazer a verificação para saber se aquele produto é caro (por exemplo, valores acima de 200 reais são considerados caros).

```
public class Produto{
    private int codigo;
    private double valor;
    //Construtor
    public Produto(int codigo, double valor){
        this.codigo = codigo;
        this.valor = valor;
    }
    // Método
    public boolean precoCaro(){
        if(valor > 200){
            return true;
        }else return false;
    }
}
```

Nosso programa conta com vários tipos de produtos (Celular, Televisão, Pilha), e iremos posicioná-los como classes filhas, ou seja, classes que herdam da classe *Produto*. Implementando, criaremos as seguintes classes filhas.

```
public class Celular extends Produto{
    private int memoria;
    //Construtor
    public Celular (int codigo, double valor, int memoria){
        super(codigo, valor);
        this.memoria = memoria;
    }
}

public class Televisao extends Produto{
    private int polegadas;
    //Construtor
    public Televisao (int codigo, double valor, int polegadas){
        super(codigo, valor);
        this.polegadas = polegadas;
    }
}

public class Pilha extends Produto{
    private boolean recarregavel;
    //Construtor
    public Pilha (int codigo, double valor, boolean recarregavel){
        super(codigo, valor);
        this.recarregavel = recarregavel;
    }
}
```

```
}
```

Vamos supor que nosso programa vai instanciar um novo objeto do tipo televisão e, para isso, utiliza o método da sua classe mãe, para verificar se o produto é caro.

```
public static void main(String[] args) {  
    //Instanciando objeto de classe filha  
    Celular cel = new Celular(100, 500, 16);  
    Televisao tv = new Televisao(111, 300, 60);  
    Pilha pil = new Pilha (110, 20, false);  
    //Imprimir informações do método de verificação de valor dos objetos  
    System.out.println("Informações do Celular \n");  
    System.out.println("Valor é considerado caro? " + cel.precoCaro());  
    System.out.println("Informações da Televisão \n");  
    System.out.println("Valor é considerado caro? " + tel.precoCaro());  
    System.out.println("Informações da Pilha\n");  
    System.out.println("Valor é considerado caro? " + pil.precoCaro());  
}
```

Podemos notar que o resultado desse programa vai retornar as seguintes informações: Celular e televisão é caro; e Pilha não é caro.

Informações do Celular
Valor é considerado caro? True

Informações da Televisão
Valor é considerado caro? True

Informações da Pilha
Valor é considerado caro? False

Figura 5 - Retorno de impressão ao usuário do programa de exemplo, contendo informações dos objetos instanciados.

Fonte: Elaborada pelo autor, 2018.

Até então, parece fazer sentido, correto? Mas se analisarmos com mais calma, vamos perceber que o valor instanciado, no objeto TV, foi de 300 reais, com polegadas de 60. Agora vamos pensar, uma televisão de 60 polegadas, por 300 reais é realmente caro? Ou seja, às vezes herdamos métodos que não fazem sentido. E agora? Quando isso acontece, podemos redefinir os métodos que não satisfazem as classes que foram herdadas. Redefinimos ele de maneira que sobrescreva aquele método da classe mãe. Para fazermos isso, basta usar o auxílio da anotação *@Override* nas classes filhas, dessa forma realizaremos uma sobrescrita daquele método. Vamos verificar como ficaria, agora, nossa classe filha *Televisão* sobrescrevendo o método *precoCaro* da sua classe mãe.

```
public class Televisao extends Produto{
    private int polegadas;
    //Construtor
    public Televisao (int codigo, double valor, int polegadas){
        super(codigo, valor);
        this.polegadas = polegadas;
    }
    //Reescrita do método
    @Override
    public boolean precoCaro(){
        if(preco > 2000){
            return true;
        }else return false;
    }
}
```

Conforme Deitel e Deitel (2016), o uso da anotação *@Override* não é obrigatório. Porém, se colocarmos o método com *@Override*, esse método necessariamente precisa reescrever um método existente na sua classe mãe.

3.2.2 Sobrescrita vs sobrecarga

Quando estudamos construtores, aprendemos que podemos criar vários métodos construtores em uma determinada classe, após isso, o programa em execução, verificando os parâmetros passados, identifica qual construtor deverá ser utilizado. Ou seja, realiza uma sobrecarga nesse método construtor.

Vamos analisar o seguinte exemplo: precisamos criar um método que faça a multiplicação de dois números. Porém, em determinada situação, os números obrigatoriamente precisam ser inteiros e, em outra situação, eles precisam ser *double*. Para resolver isso, podemos criar dois métodos com o mesmo nome, que recebem como parâmetros tipos diferentes, como também retornam tipos diferentes.

```
public int multiplicacao(int num1, int num2){
    int multiplicacao;
    multiplicacao = num1 * num2;
    return multiplicacao;
}
public double multiplicacao(double num1, double num2){
    double multiplicacao;
    multiplicacao = num1 * num2;
    return multiplicacao;
}
```

Sobrecarga e sobrescrita significa a mesma coisa? A princípio, fica parecendo que essas duas técnicas são executadas da mesma forma. Porém, precisamos ter bastante cuidado, pois eles não são a mesma coisa. Quando

utilizamos sobrecarga, significa criar métodos com o mesmo nome, dentro de uma classe, mas cada método recebe parâmetros diferentes. Como também, o retorno desses métodos pode ser distinto, os modificadores de acesso podem mudar, ou simplesmente adicionar uma nova exceção.

Diferenças	Sobrecarga	Sobrescrita
Lista de parâmetros	Deve ser modificado	Não deve ser modificado
Tipo de retorno	Pode modificar	Não pode modificar

Quadro 1 - Retorno de impressão ao usuário do programa de exemplo, contendo informações dos objetos instanciados.

Fonte: Elaborado pelo autor, adaptado de SEPE; MAITINO (2017).

Para não restar dúvidas, precisamos ficar bem atentos a suas diferenças, como vemos no quadro acima, no qual, relacionamos os pontos de diferença entre sobrecarga e sobrescrita.

3.2.3 Implementação

Para entender melhor todos esses conceitos aprendidos sobre herança, sobrescrita e sobrecarga, vamos aplicar em um cenário. Vamos implementar um programa, no qual precisamos criar uma classe mãe chamada *Empregado*, essa classe será responsável por armazenar os atributos e métodos compartilhados pelas classes filhas *Professor*, *Coordenador* e *Auxiliar*.

```
public classe Empregado{
    private int matricula;
    private String nome;
    private int cpf;
    private double salario;
    //Construtores
    public Empregado(){
    }
    public Empregado(int matricula, String nome, int cpf, double salario){
        this.matricula = matricula;
        this.nome = nome;
        this.cpf = cpf;
        this.salario = salario;
    }
    //Método
    public double salarioComBonificacao(){
        return this.salario = salario + (salario * 0.10);
    }
}
```

Como podemos notar, a classe mãe possui dois métodos construtores, nos quais é possível ser instanciado um objeto com, ou sem, parâmetros. Logo após, existe um método que retorna o salário do empregado com sua bonificação que, por padrão, todo final de ano, os empregados recebem 10% a mais no seu salário. Antes de criarmos as classes filhas, precisamos saber que a bonificação do coordenador é de 15%, ou seja, diferente dos demais empregados. Então como ficaria nossa implementação?

```
public class Professor extends Empregado{
```



```

private String disciplina;
//Construtor
public Professor(int matricula, String nome, int cpf, double salario, String disciplina){
    super(matricula, nome, cpf, salario);
    this.disciplina = disciplina;
}
public Professor(){
    super();
}
}
public class Auxiliar extends Empregado{
    private String departamento;
    //Construtor
    public Auxiliar(int matricula, String nome, int cpf, double salario, String departamento){
        super(matricula, nome, cpf, salario);
        this.departamento = departamento;
    }
    public Auxiliar (){
        super();
    }
}
public class Coordenador extends Empregado{
    private String curso;
    //Construtor
    public Coordenador (int matricula, String nome, int cpf, double salario, String curso){
        super(matricula, nome, cpf, salario);
        this.curso = curso;
    }
    public Coordenador (){
        super();
    }
    //método com sobrescrita
    @Override
    public double salarioComBonificacao(){
        return this.salario = salario + (salario * 0.15);
    }
}

```

Como podemos notar, para conseguirmos obedecer a regra de negócio estabelecida sobre a bonificação do coordenador, criamos um `@override` (sobrescrita) do método `salarioComBonificacao` da classe mãe `Empregado`. Para concluirmos nossa simulação, vamos verificar a classe de execução abaixo.

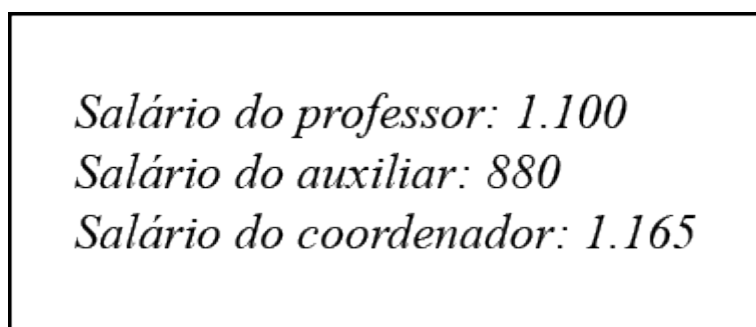
```

public static void main(String[] args) {
    //Instanciando objeto de classe filha
    Professor prof = new Professor(123, "Maria", 12345678912, 1.000, "POO");
    Auxiliar aux = new Auxiliar(234, "João", 45632178952, 800, "RH");
    Coordenador coo = new Coordenador(568, "Ana", 14785236982, 1.500, "TI");
    //Imprimir informações do método de verificação de valor dos objetos
    System.out.println("Salário do professor: " + prof.salarioComBonificacao());
    System.out.println("Salário do auxiliar: " + aux.salarioComBonificacao());
    System.out.println("Salário do coordenador: " + coo.salarioComBonificacao());
}

```

}

Podemos observar que os três objetos (*prof*, *aux* e *coo*) foram instanciando, usando o construtor com parâmetros, ou seja, realizou sobrecarga de métodos com o construtor. Logo após, quando realizamos a chamada do método *salarioComBonificacao*, podemos ver pela figura abaixo o retorno ao usuário.



Salário do professor: 1.100
Salário do auxiliar: 880
Salário do coordenador: 1.165

Figura 6 - Retorno de impressão ao usuário do programa de exemplo, contendo informações dos objetos instanciados.

Fonte: Elaborada pelo autor, 2018.

Simples, né? Será que ainda podemos melhorar? Vamos conhecer, agora, outro princípio da programação orientada a objetos, o polimorfismo.

3.3 Aplicando polimorfismo

Quando estávamos estudando herança, em um dos nossos exemplos, criamos uma classe mãe chamada *Animal*, que possuía todas as características em comum para as suas classes filhas (*Aquático*, *Terrestre* e *Aéreo*). Vamos analisar, apesar de todas as classes filhas herdarem as características da classe mãe, cada um desses objetos filhos pode se comportar com algumas especificações bem distintas. Como podemos criar um método na superclasse que sirva para todos esses objetos? Usamos polimorfismo para solucionar essas diferenças. Vamos aprender!

3.3.1 Conceitos e vantagens

Em um conceito geral, polimorfismo significa várias formas. Ou seja, mesmo que os objetos criados sejam da mesma classe mãe, eles podem se comportar de alguma maneira diferente. Então, eles poderão ter várias (poli) formas (morfismo).

VOCÊ SABIA?



Sabemos, que hoje em dia temos uma variedade de linguagens de programação no mercado, que acarretam dúvidas, quando estamos escolhendo por qual linguagem vamos usar. Mesmo assim, a maioria dos desenvolvedores optam por Java. Sabe por que? Java proporciona aos desenvolvedores gravar seus programas em uma plataforma e executarem virtualmente em qualquer outra. Como também, é possível desenvolver aplicações robustas e eficientes para *mobile*, microcontroladores, sensores, dentre outros (ORACLE, [s/d]).

Mas em código, isso significa o que? Para não restar dúvidas, polimorfismo nada mais é do que a habilidade dos objetos de classes distintas conseguirem responder a um mesmo método de diferentes maneiras. Vamos imaginar o seguinte cenário, uma empresa de doces decidiu criar um controle para ligar e desligar seus equipamentos ao mesmo tempo, sem precisar ter que ir em cada equipamento e fazer isso individualmente.

Lembre-se, todos os equipamentos são controlados pelo mesmo controle. Quando o usuário apertar o botão de ligar os equipamentos, todos eles (batedeira, liquidificador e forno) recebem o mesmo comando, enviado pelo controle e devem ser ligados. Porém, como já sabemos, cada equipamento deverá agir de maneira diferente. Por exemplo, podemos falar que quando acionada a opção de ligar a batadeira tem que começar a rodar, o liquidificador tem que acionar a hélice de corte e o forno deve acender o gás.

Notamos, que todos os equipamentos respondem ao mesmo comando, porém de formas diferentes. Ou seja, podemos afirmar que isso trata-se de polimorfismo.

Para isso acontecer, a linguagem Java procura o objeto na memória, verifica o comportamento desse objeto, para depois decidir qual método deve ser chamado. Ou seja, ele relaciona o objeto com sua classe de verdade e não com a sua classe mãe.

3.3.2 Tipos de polimorfismo

Sabendo que o polimorfismo permite que uma classe mãe possua classes filhas, com comportamentos diferentes, como podemos aplicar isso? Podemos aplicar polimorfismo de três maneiras diferentes: sobrecarga, sobreposição e inclusão.

Começando pelo polimorfismo de sobrecarga, percebemos que o nome sobrecarga já nos acompanha nos nossos estudos. Polimorfismo de sobrecarga permite que um método com o mesmo nome, possua comportamentos diferentes. Essa decisão de qual comportamento ser atendido é decidido pelos parâmetros que são recebidos por ele. Podendo receber tipos e valores diferentes por parâmetro, como também, não receber nenhum parâmetro de entrada. Isso nos lembra alguma coisa que já vimos? Sim, nossos construtores. Sabemos que podemos criar construtores que executam de forma diferente dependendo dos parâmetros que forem recebidos. Ou seja, a forma como o construtor funciona é um polimorfismo de sobrecarga.

VOCÊ QUER VER?



Percebeu que herança e polimorfismo estão bem interligados na implementação? Na verdade, podemos até entender que a utilização de herança permite que aconteça o polimorfismo. Os conceitos podem se misturar, mas, durante o desenvolvimento, fica bem claro onde fica a herança e onde fica o polimorfismo. Aprenda um pouco mais sobre a implementação de herança e polimorfismo, acessando o vídeo (GRONER, 2016): <<https://www.youtube.com/watch?v=pMPlngyWHLM>>.

Aplicando em outro cenário, vamos criar um programa que deverá calcular a área de uma figura geométrica. Como você já sabe, esse cálculo vai variar, de acordo com a figura geométrica. Por exemplo, no quadrado, basta multiplicar por dois a medida de um lado (área = L^2), já em um retângulo seria a base multiplicado pela altura (área = $B \times A$).

```
public class Area{
    private double x;
    private double y;
    //Métodos com polimorfismo de sobrecarga
    public double calcularArea(double x){
        return x * x;
    }
    public double calcularArea(double x, double y){
        return x * y;
    }
}
```

Observe que temos dois métodos com o mesmo nome *calcularArea* e o que difere entre eles são os parâmetros de entrada. No primeiro método é recebido apenas um parâmetro, já no segundo, é necessário que sejam repassados dois parâmetros de entrada para o método executar. Os dois métodos possuem o mesmo objetivo, calcular a área de uma figura geométrica, porém, realizam essa ação de maneiras diferentes.

Nosso próximo tipo é polimorfismo de sobreposição. Qual seria a diferença entre sobrecarga e sobreposição? No caso da sobrecarga, os métodos com o mesmo nome são criados dentro da mesma classe, já em sobrescrita, significa que as classes filhas possuem o mesmo método da sua classe mãe, que irá sobrepor aquele método da classe mãe. Também reconhece esse conceito? Pois é, lembra do nosso exemplo de bonificação? Nele, a classe mãe *Empregado* possuía um método *salarioComBonificacao* que realizava um aumento de 10% no salário. E uma das classes filhas, chamada *Coordenador* também possuía esse mesmo método dentro da classe, porém com a diferença de cálculo, que no nosso caso, seria de 15%.

Ou seja, polimorfismo de sobrescrita se trata de uma redefinição de métodos em classes descendentes (classes filhas). Para isso acontecer, um método de uma classe filha deverá possuir o mesmo nome de um método da classe mãe, dessa forma, acontece a sobreposição.

Vamos implementar nosso exemplo da empresa de doces? Vamos criar uma classe mãe (*Equipamento*) que descreverá os comportamentos. Nossa classe mãe possuirá os atributos (*tempo*, *voltagem* e *marca*) e métodos (*ligar* e *desligar*), que são compartilhados entre as classes filhas.

```
public class Equipamento{
    private double tempo;
    private double voltagem;
```

```

private String marca;
//Métodos
private void ligar(){
System.out.println("Ligar equipamento!");
}
private void desligar(){
System.out.println("Desligar equipamento!");
}
}

```

Lembre-se que o comportamento de cada equipamento será diferente, quando acionado o método *ligar*. A batedeira tem que começar a bater, o liquidificador tem que acionar a hélice de corte e o forno deve acender o gás. Vejamos como podemos usar o polimorfismo de sobrescrita nessas classes filhas.

```

public void Batedeira extends Equipamento{
// Método com polimorfismo de sobreposição
@Override
public void ligar(){
System.out.println("Bater!");
}
}
public void Liquidificador extends Equipamento{
// Método com polimorfismo de sobreposição
@Override
public void ligar(){
System.out.println("Girar!");
}
}
public void Forno extends Equipamento{
// Método com polimorfismo de sobreposição
@Override
public void ligar(){
System.out.println("Acender!");
}
}

```

Até o momento, estamos com a herança (classes mãe e filhas criadas), mas no nosso exemplo, possuímos um controle, o qual deverá controlar as ações do usuário de ligar e desligar os equipamentos. Tendo em vista que o método *ligar* existe nas classes filhas e classe mãe, como o controle sabe qual equipamento ligar quando for acionado o método *ligar*? Como ele sabe qual método *ligar* ele deve utilizar, já que existem três tipos diferentes? Para resolvermos isso, basta adicionar no método *construtor*, da classe do controle, um parâmetro que receberá o tipo do equipamento que deve ser ligado. Vamos ver como ficaria nossa classe de controle.

```

public class Controle{
private Equipamento equipamento;
//Construtor
public Controle(Equipamento equipamento){
this.equipamento = equipamento;
}
//Método
public void ligar(){
equipamento.ligar();
}
}

```

```
}
```

De forma a concluirmos nosso programa, vamos simular uma possível ação do nosso programa. Vamos analisar o programa abaixo.

```
public static void main(String[] args) {  
    //Instanciando objeto de classe filha  
    Liquidificador liquidificador = new Liquidificador();  
    //Instanciando objeto controle passando por parâmetro o objeto liquidificador  
    Controle controle = new Controle(liq);  
    //Chamada de método  
    controle.ligar();  
}
```

Podemos observar que criamos um equipamento do tipo liquidificador, e após criarmos o controle, já identificamos qual objeto será controlado. Dessa forma, quando a instrução *ct.ligar()* for acionada, nosso programa saberá que o método utilizado é o método *ligar* da classe *liquidificador*, fazendo com que o resultado do nosso programa seja a palavra: Girar!

VOCÊ SABIA?



Utilizar herança significa aumentar o acoplamento entre as classes. Ou seja, aumenta a dependência que uma classe tem da outra. Então, enquanto desenvolvedores, precisamos ter bastante cuidado quando estivermos implementando herança. Pois, devido a essa dependência criada entre classe mãe e classe filha, torna-se uma tarefa árdua realizar mudanças mais pontuais no nosso programa (CAELUM, 2018, p. 131).

Por fim, temos o polimorfismo de inclusão, que trata justamente da capacidade de substituição de métodos possíveis devido ao uso de herança. Não entendeu? Vamos analisar com mais cuidado o nosso programa da empresa de doces.

Quando criamos a classe *Controle*, significa que utilizamos o comportamento polimórfico para expressar qual objeto deve ser controlado naquele momento, informando o objeto por parâmetro. Se as classes filhas (*batedeira*, *liquidificador* e *forno*) não estivessem presentes em uma estrutura de herança, a classe *controle* deveria criar um método *ligar*, para manusear separadamente cada objeto diferente. Ou seja, da forma que criamos nosso programa, garante que, para interagir entre os objetos possíveis, só é necessário passar o tipo de equipamento para a classe *Controle*. Essa estrutura da classe *Controle*, de receber qualquer tipo de objeto das classes filhas, é o que chamamos de polimorfismo de inclusão. Interessante, né?

Resumindo, trabalhar com polimorfismo, na linguagem Java, significa controlar todas as formas de uma maneira simples, visualizando a estrutura geral, mas sem deixar de identificar as especificações de cada objeto.

3.4 Aprendendo sobre abstração

Até o momento, nós vimos formas de facilitar nosso desenvolvimento, utilizando herança e polimorfismo, para economizarmos um pouco de código no nosso programa e criando atributos e métodos mais genéricos que são compartilhados por outras classes.

Será que temos como melhorar nosso código?

Vamos continuar nessa busca, para aprender mais!

3.4.1 Conceitos e objetivos

Aplicando os conhecimentos adquiridos, vamos analisar o seguinte cenário: vamos imaginar que nossa empresa de doces vende seus produtos para clientes que podem estar cadastrados como pessoa física ou jurídica. Para melhorar nossa regra de negócio, sabemos que podemos criar uma classe mãe *Cliente* que possua duas classes filhas (*Física* e *Jurídica*).

Na hora de realizar a venda, a empresa precisa acessar no programa, as informações desse cliente, que neste caso, o objeto só poderia ser, ou da classe *Física*, ou da classe *Jurídica*.

Da forma que modelamos nosso código, fica nítido que a classe mãe *Cliente* está sendo utilizada apenas para garantir que possamos fazer polimorfismo. Em outras palavras, quando for adicionado um novo cliente, significa que estaremos sempre instanciando ele em alguma classe filha. Notamos que, mesmo com a existência de uma classe mãe, não faz sentido permitir que seja instanciado um objeto da classe mãe.

Como podemos resolver esse problema? Para isso usamos nossas classes abstratas. Vamos aprender!

3.4.2 Classes abstratas e herança

Até aqui, estávamos lidando com classes concretas, ou seja, classes que são utilizadas para gerar um determinado objeto. Mas, voltando ao exemplo de cliente, percebemos que não devemos instanciar diretamente um objeto na classe *Cliente*, correto? Precisamos então proibir que essa classe seja instanciada em código, para isso usamos a palavra-chave *abstract*, quando estamos desenvolvendo esse tipo de classe. De uma maneira conceitual, vamos implementar a classe mãe *Cliente* de maneira abstrata.

```
public abstract class Cliente{  
    private String nome;  
    private int documento;  
    //construtores e métodos  
    ...  
}
```

Como já aprendemos, a classe *Cliente* modela quais informações são compartilhadas para suas classes filhas. Como agora declaramos a classe como abstrata, se durante a execução do programa for solicitado que seja instanciado um objeto do tipo *Cliente*, nosso programa retornará com um erro de compilação. Por que agora nossa classe mãe nos ajuda a gerar polimorfismo e herança e, não mais, podendo ser instanciada.

VOCÊ QUER LER?



Quem estuda sobre programação orientada a objetos, percebe que trabalhar com classes abstratas é relativamente simples. Porém, precisamos ter bastante cuidado para identificar quais são as situações adequadas para optarmos pelo uso delas. Para ajudar nessa escolha, você pode aprofundar seus estudos com o pacote java.io (JAVATPOINT, [s/d]). Quer entender melhor como funciona o pacote java.io? Conheça mais: <https://www.javatpoint.com/java-io>.

Quando realizamos essa condição de instanciação de objeto, garantimos uma melhor qualidade e consistência ao nosso programa. E como fica o acesso as informações em uma classe abstrata?

3.4.3 Métodos abstratos

Vamos imaginar, agora, a seguinte regra de negócio para nossa empresa de doces: de acordo com o tipo de cliente que está realizando a compra, os descontos são diferenciados. Vamos supor que o cliente do tipo *físico*, ganha 10%, enquanto o cliente do tipo *jurídico*, ganha 15%. Como podemos implementar isso?

Sabendo que existe essa diferenciação no valor do desconto, não faz sentido nós colocarmos esse método na nossa classe mãe. Mas se o método não existir na classe mãe, como chamaremos o método, quando for referência a uma classe filha? Já sei, podemos escrever uma implementação geral na classe mãe, e sobrescrever esse método nas classes filhas. Mas e se esquecermos de sobrescrever nas classes filhas?

Para resolver isso, usando de um recurso da classe abstrata, no qual informamos dentro da classe abstrata que determinado método será abstrato, ou seja, esse método deverá sempre ser descrito nas suas classes filhas. Vamos adicionar esse método de desconto na nossa classe abstrata *Cliente*.

```
public abstract class Cliente{
    private String nome;
    private int documento;
    //Método abstrato
    public abstract double desconto();
}
```

Notamos que, para criarmos métodos abstratos, basta apenas adicionar a palavra-chave *abstract* na declaração do método e não implementar nenhuma instrução interna a esse método. Pois, agora, é dever das classes filhas sobrescreverem esse método, da maneira que deve ser. Como ficaria nossas classes filhas *Física* e *Jurídica*?

```
public class Fisica extends Cliente{
    //Método
    @Override
    public double desconto(){
        System.out.println("Autorizado desconto de 10%");
    }
}

public class Juridica extends Cliente{
    //Método
    @Override
    public double desconto(){
        System.out.println("Autorizado desconto de 15%");
    }
}
```

Caso alguma classe filha não sobrescreva o método *desconto()*, o programa não conseguirá compilar. Pois agora, devido a usarmos classes e métodos abstratos, precisamos que as classes filhas tornem essas informações abstratas, definidas pela classe mãe, em informações concretas.

CASO



Vimos que a herança múltipla entre classes acontece quando uma subclasse possuir duas ou mais superclasses, ou seja, a subclasse é "filha" de mais de uma classe. Dessa forma, é possível combinar os atributos e métodos de várias superclasses existentes para a formação de uma nova classe. Na linguagem Java, não conseguimos implementar esse conceito de herança múltipla, pois a plataforma não dá esse tipo de suporte.

Mas para entender esse conceito em prática, vamos desenvolver um programa na linguagem C++, que dá suporte a herança múltipla. Vamos criar um programa, no qual temos um relógio que mostra a hora e o calendário. Devido ao comportamento ser bem diferente, vamos criar duas classes mãe, uma para o funcionamento do relógio e outra para o funcionamento do calendário.

```
class Relogio { protected: int hora; int min; int segundo; public: Relogio(int hora, int mimuto, int segundo); void alterar_hora (int hora, int mimuto, int segundo); void mostrar_hora (); }  
class Calendario { protected: int mes; int dia; int ano; public: Calendario(int mes, int dia, int ano); void alterar_data (int mes, int dia, int ano); void mostrar_data (int &mes, int &dia, int &ano); }
```

Como nosso objeto deverá ser um Relógio/Calendário, precisamos criar uma classe que consiga herdar os comportamentos das classes mãe Relógio e Calendário.

```
class Relogio_Calendario : public Relogio, public Calendario{ public: Relogio_Calendario(int mes, int dia, int ano, int hora, int minuto, int segundo); void mostrar_data_hora(); }
```

Percebemos na classe filha *Relogio_Calendario* que ela estende as duas classes mãe criadas anteriormente. Dessa forma, um objeto criado do tipo *Relogio_Calendario*, possui todos os comportamentos da classe Relógio e da classe Calendário.

Percebemos que utilizar todo esse conceito de herança, polimorfismo e classes abstratas torna nosso programa mais reutilizável.

Ao final do conteúdo, entendemos que trabalhar com herança nos ajuda a compartilhar atributos de uma mesma classe. Entendemos isso implementando códigos, e compreendendo o significado de polimorfismo e abstração de código.

Síntese

Chegamos ao final deste capítulo. Aprendemos a desenvolver nossos programas de maneira a deixá-los mais reutilizáveis. Estudando conceitos de herança e polimorfismo, conseguimos pensar melhor em um modelo de negócio para nosso programa, gerando, assim, uma solução mais eficaz e de maior qualidade.

Neste capítulo, você teve a oportunidade de:

- compreender o significado de herança e saber identificar quando é adequado utilizá-la, solucionando problemas entre classes;
- aprender sobre sobrecarga e sobrescrita de código, entendendo sobre a necessidade de utilizar essas vantagens;
- estudar sobre polimorfismo e seus tipos, desenvolvendo soluções junto com a herança;

- aplicamos classes e métodos abstratos para evitar problemas de compilação.

Bibliografia

CAELUM. **Java e orientação a objetos**. Caelum, 2018.

CESTA, A. A.; RUBIRA, C. M. F. **Tutorial: a linguagem de programação Java e orientação a objetos**. Instituto de Computação, Julho 1996; Atualizado em 2009. Campinas: Unicamp, 2009. Disponível em: <<http://www.ic.unicamp.br/~cmrubira/JAVATUT14PDF.pdf>>. Acesso em: 15/08/2018.

DEITEL, P; DEITEL, H. **Java: como programar**. 10. ed. São Paulo: Pearson: 2016.

FOWLER, M. **Designed In Heritance**. Portal MartinFowler.com, 2006. Disponível em: <<https://martinfowler.com/bliki/DesignedInheritance.html>>. Acesso em: 22/08/2018.

GRONER, L. 2016. **Curso de Java 40: Orientação a Objetos: Herança e Polimorfismo: sobrecarga de métodos**. Canal Loiane Groner, YouTube. Publicado em 1 de fev. de 2016. Disponível em: <<https://www.youtube.com/watch?v=pMPlngyWHLm>>. Acesso em: 22/08/2018.

JAVATPOINT. **Java I/O Tutorial**. Portal JavaTpoint, [s/d]. Disponível em: <<https://www.javatpoint.com/java-io>>. Acesso em: 22/08/2018.

MANZANO, J. A. G.; COSTA JR., R. **Programação de Computadores com Java**. São Paulo: Érica, 2014. 127p.

MARINHO, A. L. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil, 2016. 167p.

ORACLE. **Obtenha informações sobre a Tecnologia Java**. Portal Java, [s/d]. Disponível em: <https://java.com/pt_BR/about/>. Acesso em: 22/08/2018.

SCHOFIELD, J. James Gosling, o pai do Java. **Portal Gazeta do Povo**, publicado em 16/04/2007. Disponível em: <<http://www.gazetadopovo.com.br/tecnologia/james-gosling-o-pai-do-java-afz7nojdx57547vdfwa6zim>>. Acesso em: 22/08/2018.

SEPE, A.; MAITINO, R. N. **Programação orientada a objetos**. Londrina: Editora e Distribuidora Educacional AS, 2017. 176p.