



Pilares da Programação Orientada a Objetos em Java

≡ Links	Pasta no GitHub: https://github.com/marlonprado04/DIO/tree/main/BOOTCAMP_desenvolvimento_java_com_cloud_aws/05_curs
📁 Categoria	Curso
⚙️ Status	In Progress
📖 Conteúdos DB	 Java
📅 Plano de estudos DB	Coding The Future - GFT e AWS Desenvolvimento Java com Cloud AWS
📅 Data	@14/08/2023
📌 Instituições e plataformas DB	 DIO

Pilares da Programação Orientada a Objetos em Java

▼ Sumário

[Pilares da Programação Orientada a Objetos em Java](#)

[Sumário](#)

[Antes de começar](#)

[Pilares de POO](#)

[Encapsulamento](#)

[Herança](#)

[Abstração](#)

[Polimorfismo](#)

[Na prática](#)

[Link da referência da aula completa](#)

[Encapsulamento](#)

[Link da referência da aula completa](#)

[Herança](#)

[Link da referência da aula completa](#)

[Abstração](#)

[Link da referência da aula completa](#)

[Polimorfismo](#)

[Modificador protected](#)

[Link da referência da aula completa](#)

[Interface](#)

[Link da referência da aula completa](#)

[Links e materiais adicionais](#)

[Tarefas adicionais](#)

Antes de começar

☒ [Criar pasta referente ao curso](#)

☒ [Adicionar link da pasta nos atributos do curso](#)

Pilares de POO

POO é um **paradigma de programação** baseado no conceito de objetos.

Como se trata de um contexto análogo ao mundo real tudo acaba sendo um objeto, por exemplo, conta bancária, aluno, instituição, etc.

A programação orientada a objetos é bem requisitada no mercado devido a possibilidade de reutilizar os códigos e a capacidade de representação do sistema com exemplos muito próximos do mundo real.

Para uma linguagem ser considerada orientada a objeto, ela deve seguir os **quatro pilares da orientação a objetos**, sendo eles:

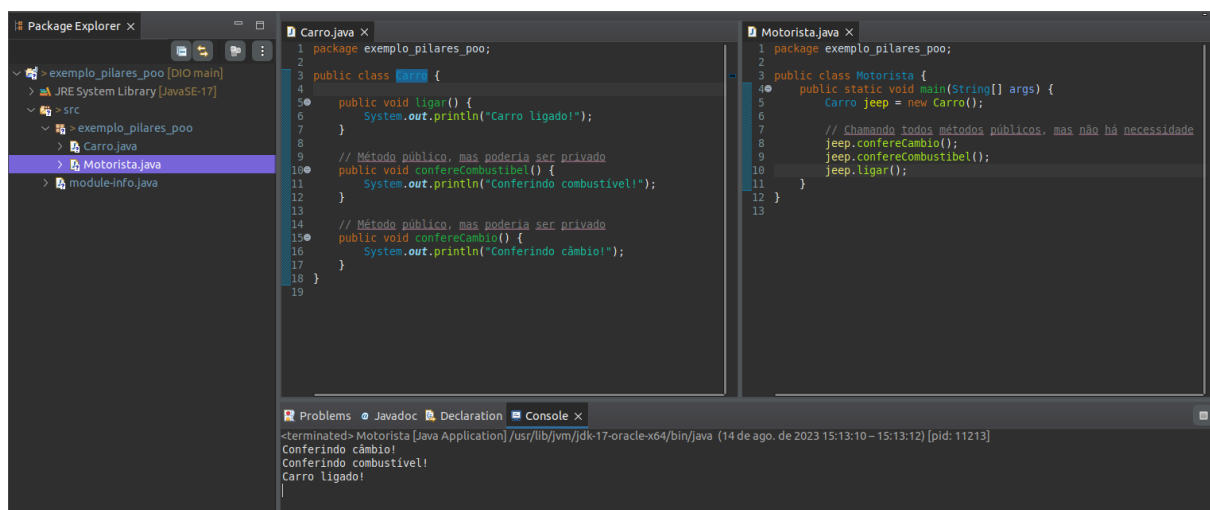
Encapsulamento

Nem tudo precisa estar visível, boa parte do algoritmo pode ser distribuído em métodos com finalidades específicas que complementam a aplicação.

Exemplo: *Ligar um veículo, exige muitas etapas para a engenharia, mas o condutor só visualiza a ignição, dar a partida e a “magia” acontece. Tudo que acontece entre dar a partida e a ignição se iniciar pode ser encapsulado.*

Exemplo prático: No código abaixo, ao invés de todos os métodos da classe **Carro** estarem públicos, podemos deixar alguns métodos privados (encapsulados) para que somente a classe **Carro** execute.

Código antes:



```
1 package exemplo.pilares.poo;
2
3 public class Carro {
4
5     public void ligar() {
6         System.out.println("Carro ligado!");
7     }
8
9     // Método público, mas poderia ser privado
10    public void confereCombustivel() {
11        System.out.println("Conferindo combustivel!");
12    }
13
14    // Método público, mas poderia ser privado
15    public void confereCambio() {
16        System.out.println("Conferindo câmbio!");
17    }
18 }
19
```

```
1 package exemplo.pilares.poo;
2
3 public class Motorista {
4     public static void main(String[] args) {
5         Carro jeep = new Carro();
6
7         // Chamando todos métodos públicos, mas não há necessidade
8         jeep.confereCambio();
9         jeep.confereCombustivel();
10        jeep.ligar();
11    }
12 }
13
```

Problems Javadoc Declaration Console X

```
<terminated> Motorista [Java Application] /usr/lib/jvm/jdk-17-oracle-x64/bin/java (14 de ago. de 2023 15:13:10 - 15:13:12) [pid: 11213]
Conferindo câmbio!
Conferindo combustivel!
Carro ligado!
```

Código depois:

```

Carro.java
1 package exemplo_pilares_poo;
2
3 public class Carro {
4
5     public void ligar() {
6         // Chamando métodos privados da classe
7         confereCambio();
8         confereCombustivel();
9
10        // Informando que carro está sendo ligado
11        System.out.println("Carro ligado!");
12    }
13
14    // Método privado
15    private void confereCombustivel() {
16        System.out.println("Conferindo combustivel!");
17    }
18
19    // Método privado
20    private void confereCambio() {
21        System.out.println("Conferindo câmbio!");
22    }
23 }
24

Motorista.java
1 package exemplo_pilares_poo;
2
3 public class Motorista {
4     public static void main(String[] args) {
5         Carro jeep = new Carro();
6
7         // Chamando apenas o método ligar que é público
8         jeep.ligar();
9     }
10 }
11

Problems | Javadoc | Declaration | Console
<terminated> Motorista [Java Application] /usr/lib/jvm/jdk-17-oracle-x64/bin/java (14 de ago. de 2023 15:16:17 - 15:16:17) [pid: 11386]
Conferindo câmbio!
Conferindo combustivel!
Carro ligado!

```

Herança

Características e comportamentos comuns podem ser elevados e compartilhados através de uma hierarquia de objetos.

Exemplo: Um Carro e uma Motocicleta possuem propriedades como placa, chassi, ano de fabricação e métodos como acelerar e frear. Logo, para não ser um processo de codificação redundante, podemos desfrutar da herança criando uma classe **Veículo** para que seja herdada por **Carro** e **Motocicleta**.

Exemplo prático: **Carros** e **Motos** podem ser duas classes que **possuem atributos similares**, então ao invés de declarar eles em cada classe, **podemos ter uma “classe pai” chamada Veiculo que fará a classe Carro e Moto herdar seus atributos ao adicionar** `extends Veiculo`

Código antes:

```

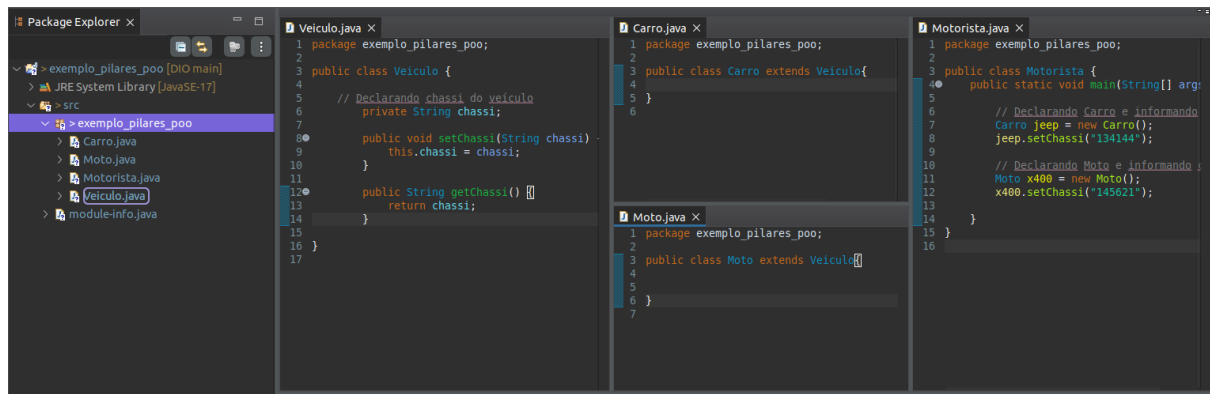
Moto.java
3 public class Moto {
4
5     // Declarando chassi da moto
6     private String chassi;
7
8     public void setChassi(String chassi) {
9         this.chassi = chassi;
10    }
11
12    public String getChassi() {
13        return chassi;
14    }
15 }

Carro.java
3 public class Carro {
4
5     // Declarando chassi do carro
6     private String chassi;
7
8     public void setChassi(String chassi) {
9         this.chassi = chassi;
10    }
11
12    public String getChassi() {
13        return chassi;
14    }
15 }

Motorista.java
1 package exemplo_pilares_poo;
2
3 public class Motorista {
4     public static void main(String[] args) {
5
6         // Declarando Carro e informando chassi
7         Carro jeep = new Carro();
8         jeep.setChassi("134144");
9
10        // Declarando Moto e informando chassi
11        Moto x400 = new Moto();
12        x400.setChassi("145621");
13    }
14 }
15
16

```

Código depois:



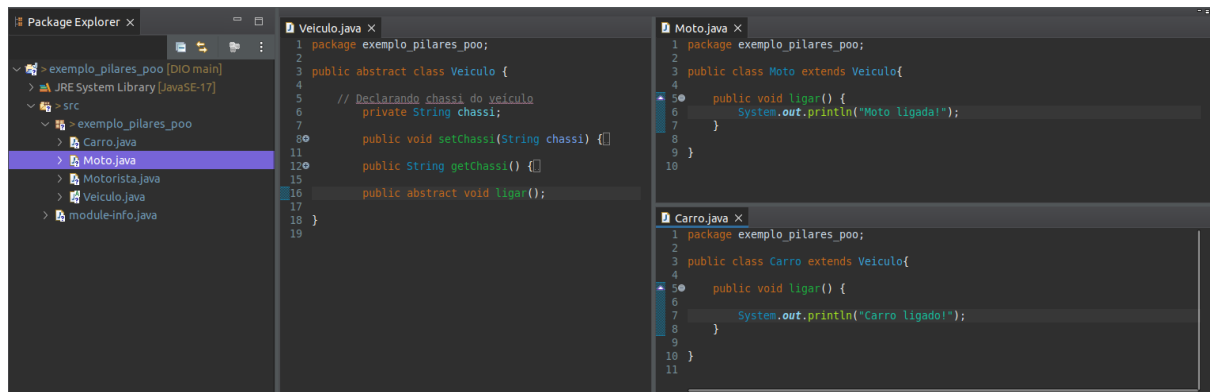
Abstração

É o processo de simplificar complexidade ao focar nos aspectos relevantes e ignorar os detalhes menos importantes. A abstração envolve ocultar os detalhes internos e complexos, permitindo aos usuários interagir com ele de maneira mais simples e compreensível.

Exemplo: **Veículo** determina duas ações como acelerar e frear, logo, estes comportamentos deverão ser **abstratos**, pois existem mais de uma maneira de se realizar a mesma operação.

Exemplo prático: Uma **moto** e um **carro** possuem funções semelhantes mas que podem ter diferenças, por exemplo, **ao ligar uma moto e um carro alguns passos diferentes precisam ser executados**. Dessa forma, ao podemos atribuir um **método ligar** à classe genérica **Veículo** e informar que essa classe possui um método **abstrato** que pode mudar de acordo com o tipo de veículo.

Código com abstração:

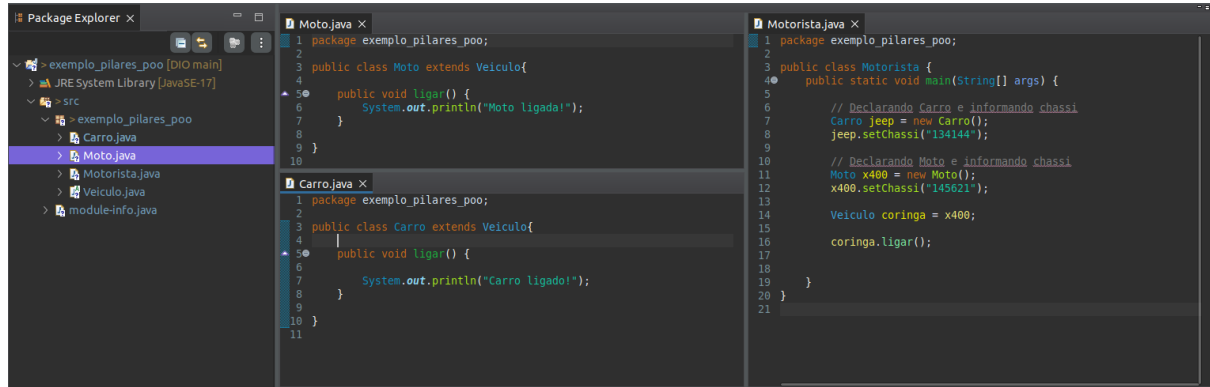


Polimorfismo

São as inúmeras maneiras de se realizar uma mesma ação.

Exemplo: **Veículo** determina duas ações como acelerar e frear, primeiramente, precisamos identificar se estaremos nos referindo a **Carro** ou **Motocicleta**, para determinar a lógica de aceleração e frenagem dos respectivos veículos.

Código:



Para ilustrar a proposta dos Princípios de POO, no nosso cotidiano, vamos simular algumas funcionalidades dos aplicativos de mensagens instantâneas pela internet.

Vamos descrever em UML e depois em código, algumas das principais funcionalidades de qualquer serviço de comunicação instantânea pela internet, inicialmente pelo MSN Messenger e depois inserindo os demais, considerando os princípios de POO.

MSNMessenger
+ enviarMensagem() : void
+ receberMensagem() : void
+ validarConectadoInternet() : void
+ salvarHistoricoMensagem() : void

- Todos os métodos da classe são public (tudo realmente precisa estar visível ?);
- Só existe uma única forma de se comunicar via internet (como ter novas formas de se comunicar mantendo a proposta central ?).

Código base:


```
public class MSNMessenger {
    public void enviarMensagem() {
        System.out.println("Enviando mensagem");
    }
    public void receberMensagem() {
        System.out.println("Recebendo mensagem");
    }
    public void validarConectadoInternet() {
        System.out.println("Validando se está conectado a internet");
    }
    public void salvarHistoricoMensagem() {
        System.out.println("Salvando o histórico da mensagem");
    }
}
```

Link da referência da aula completa

Pilares do POO

J **Java Básico**

Pilares do POO

 <https://glysns.gitbook.io/java-basico/programacao-orientada-a-objetos/pilares-do-poo>

 Powered By GitBook

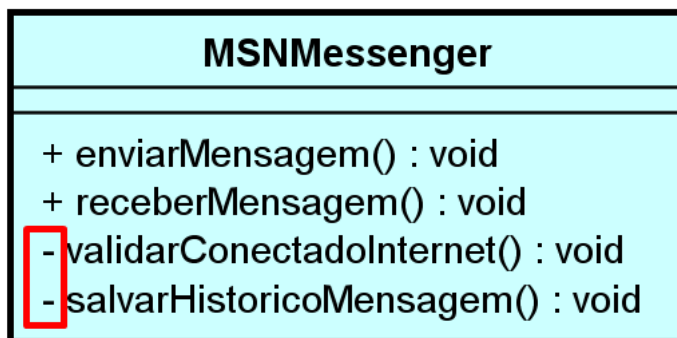
Encapsulamento

Nem tudo precisa estar disponível para todos

Já imaginou, você instalar o MSN Messenger e ao querer enviar uma mensagem, fosse solicitado a você verificar se o computador está conectado a internet e depois, pedir para você salvar a mensagem no histórico? ou, se ao tentar enviar um SMS pelo celular, primeiro você precisasse consultar manualmente o seu saldo?

Quanto ao MSN Messenger, para nós, só é relevante saber que podemos enviar e receber a mensagem, logo, as demais funcionalidades poderão ser consideradas privadas (private). E é aí que se caracteriza a necessidade do pilar de Encapsulamento. O que esconder?

▼ UML



▼ Código modificado

```
public class MSNMessenger {
    public void enviarMensagem() {
        //primeiro confirmar se esta conectado a internet
        validarConectadoInternet();

        System.out.println("Enviando mensagem");

        //depois de enviada, salva o histórico da mensagem
        salvarHistoricoMensagem();
    }

    public void receberMensagem() {
        System.out.println("Recebendo mensagem");
    }

    //métodos privadas, visíveis somente na classe
    private void validarConectadoInternet() {
        System.out.println("Validando se está conectado a internet");
    }
    private void salvarHistoricoMensagem() {
        System.out.println("Salvando o histórico da mensagem");
    }
}
```

Link da referência da aula completa

Encapsulamento

Nem tudo precisa ser estar disponível para todos!

<https://glysns.gitbook.io/java-basico/programacao-orientada-a-objetos/pilares-do-poo/encapsulamento>

J Java Básico

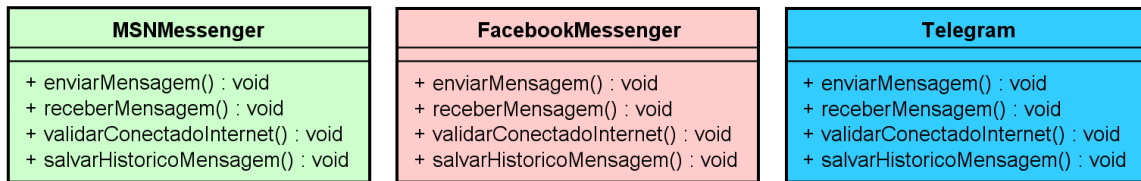
Encapsulamento

Powered By GitBook

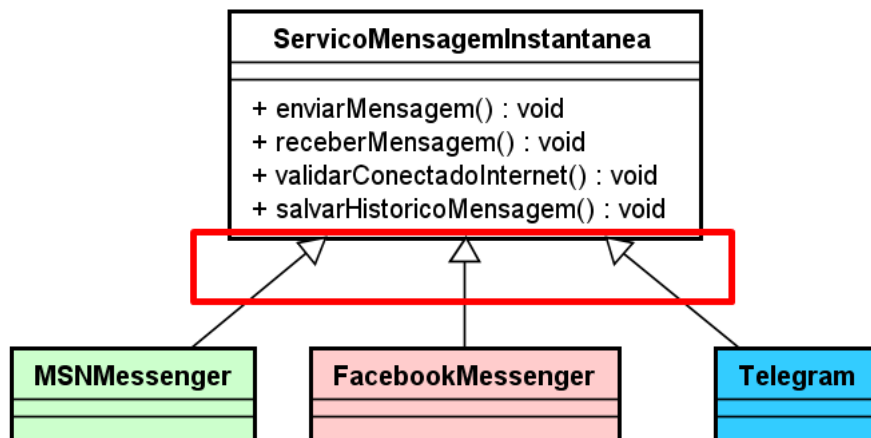
Herança

Nem tudo se copia, as vezes se herda

Imagine ter que criar vários serviços de mensagem instantânea que utilizam os mesmos métodos.



A melhor abordagem para isso seria utilizar o conceito de herança de POO.



Dessa forma, todos os serviços herdam o método da classe pai, como nos códigos abaixo:

▼ Classe ServicoMensagemInstantanea

```

//a classe MSNMessenger é ou representa
public class ServicoMensagemInstantanea {
    public void enviarMensagem() {
        //primeiro confirmar se esta conectado a internet
        validarConectadoInternet();
        System.out.println("Enviando mensagem");
        //depois de enviada, salva o histórico da mensagem
        salvarHistoricoMensagem();
    }
    public void receberMensagem() {
        System.out.println("Recebendo mensagem");
    }

    //métodos privadas, visíveis somente na classe
    private void validarConectadoInternet() {
        System.out.println("Validando se está conectado a internet");
    }
    private void salvarHistoricoMensagem() {
        System.out.println("Salvando o histórico da mensagem");
    }
}
  
```

▼ Classe MSNMessenger


```
public class MSNMessenger extends ServicoMensagemInstantanea{
}
```

▼ Classe FacebookMessenger

```
public class FacebookMessenger extends ServicoMensagemInstantanea {
}
```

▼ Classe Telegram

```
public class Telegram extends ServicoMensagemInstantanea {
}
```

▼ Classe ComputadorPedrinho

```
public class ComputadorPedrinho {
    public static void main(String[] args) {

        MSNMessenger msn = new MSNMessenger();
        msn.enviarMensagem();
        msn.receberMensagem();

        FacebookMessenger fbm = new FacebookMessenger();
        fbm.enviarMensagem();
        fbm.receberMensagem();

        Telegram tlg = new Telegram();
        tlg.enviarMensagem();
        tlg.receberMensagem();

    }
}
```

Será que todos os sistemas de mensagens, realizam as suas operações de uma mesma maneira? e agora ? este é um trabalho para os pilares **Abstração** e **Polimorfismo**.

Link da referência da aula completa

Herança

Nem tudo se copia, às vezes se herda.

<https://glysns.gitbook.io/java-basico/programacao-orientada-a-objetos/pilares-do-poo/heranca>

J Java Básico

Herança

Powered By GitBook

Abstração

Para você ser, é preciso você fazer.

Sabemos que qualquer sistema de mensagens instantâneas realiza as mesmas operações de Enviar e Receber Mensagem, dentre outras operações comuns ou exclusivas de cada aplicativo disponível no mercado.

Mas será que as ações realizadas, contém o mesmo comportamento ? Acreditamos que não.

Já imaginou a **Microsoft** falar para o **Facebook**: *"Ei, toma meu código do MSN!"*.

Observem a nova estruturação dos códigos abaixo, com base na implementação apresentada no pilar Herança.

▼ Classe ServicoMensagemInstantanea

```
public abstract class ServicoMensagemInstantanea {  
    public abstract void enviarMensagem();  
    public abstract void receberMensagem();  
}
```

▼ Classe MSNMessenger

```
public class MSNMessenger extends ServicoMensagemInstantanea{  
    public void enviarMensagem() {  
        System.out.println("Enviando mensagem pelo MSN Messenger");  
    }  
    public void receberMensagem() {  
        System.out.println("Recebendo mensagem pelo MSN Messenger");  
    }  
}
```

▼ Classe FacebookMessenger

```
public class FacebookMessenger extends ServicoMensagemInstantanea {  
    public void enviarMensagem() {  
        System.out.println("Enviando mensagem pelo Facebook Messenger");  
    }  
    public void receberMensagem() {  
        System.out.println("Recebendo mensagem pelo Facebook Messenger");  
    }  
}
```

▼ Classe Telegram

```
public class Telegram extends ServicoMensagemInstantanea {  
    public void enviarMensagem() {  
        System.out.println("Enviando mensagem pelo Telegram");  
    }  
    public void receberMensagem() {  
        System.out.println("Recebendo mensagem pelo Telegram");  
    }  
}
```

Antes, com a herança, havia uma única forma de realizar o envio e recebimento de mensagens. Agora cada sistema deve informar sua lógica de envio e recebimento de mensagem.

A abstração em POO determinar que todos precisam fazer, mas cada um deve informar a sua maneira.



Em Java, o conceito de abstração é representado pela palavra reservada `**abstract**` e métodos que **NÃO** possuem corpo na classe abstrata (pai).

Link da referência da aula completa

Abstração

Para você ser, é preciso você fazer.

<https://glyns.gitbook.io/java-basico/programacao-orientada-a-objetos/pilares-do-poo/abstracao>

J Java Básico

Abstração

Powered By GitBook

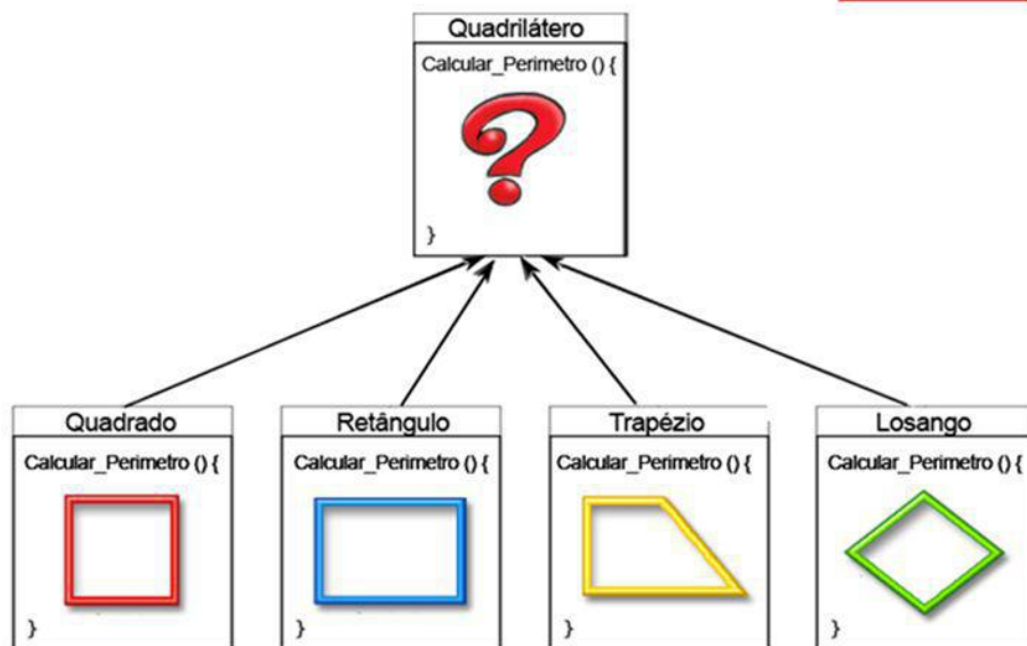
Polimorfismo

Um mesmo comportamento, de várias maneiras.

Qualquer classe que deseja representar um serviço de mensagens, basta estender a classe `ServicoMensagemInstantanea` e implementar, os respectivos métodos *abstratos*.

O que vale reforçar é que cada classe terá a mesma ação, executando procedimentos de maneira especializada.

Polimorfismo – O que é?



Vejamos o código abaixo:

```
public class ComputadorPedrinho {  
    public static void main(String[] args) {  
        ServicoMensagemInstantanea smi = null;  
    }  
}
```

```

/*
    NÃO SE SABE QUAL APP
    MAS QUALQUER UM DEVERÁ ENVIAR E RECEBER MENSAGEM
*/
String appEscolhido="???";

if(appEscolhido.equals("msn"))
    smi = new MSNMessenger();
else if(appEscolhido.equals("fbm"))
    smi = new FacebookMessenger();
else if(appEscolhido.equals("tlg"))
    smi = new Telegram();

smi.enviarMensagem();
smi.receberMensagem();
}
}

```

Modificador protected

O modificador **protected** está muito **associado à herança** que é um dos pilares de POO.

Sabemos que cada aplicativo, costuma salvar as mensagens em seus respectivos servidores cloud, mas e quanto validar se está conectado a internet? Não poderia ser um mecanismo comum a todos ? Logo, qualquer classe filha, de **ServicoMensagemInstantanea** poderia desfrutar através de herança, esta funcionalidade.



Mas fica a reflexão do que já aprendemos sobre visibilidade de recursos: Com o modificador **privat** somente a classe conhece a implementação, quanto que o modificador **public** todos passarão a conhecer. Mas gostaríamos que, somente as classes filhas soubessem. Bem, é aí que entra o modificador **protected**.

O código do serviço pai ficaria:

```

public abstract class ServicoMensagemInstantanea {

    public abstract void enviarMensagem();
    public abstract void receberMensagem();

    //mais um método que todos os filhos deverão implementar
    public abstract void salvarHistoricoMensagem();

    //somente os filhos conhecem este método
    protected void validarConectadoInternet() {
        System.out.println("Validando se está conectado a internet");
    }
}

```

Link da referência da aula completa

Polimorfismo

Um mesmo comportamento, de várias maneiras.

<https://glysns.gitbook.io/java-basico/programacao-orientada-a-objetos/pilares-do-poo/polimorfismo>

J Java Básico

Polimorfismo

Powered By GitBook

Interface

Antes de tudo, **NÃO** estamos nos referindo a interface gráfica.

Como vimos anteriormente, **Herança** é um dos pilares de POO, mas uma curiosidade que se deve ser esclarecida, na linguagem Java, é que a mesma **não permite o que conhecemos como Herança Múltipla**.

A medida que vão surgindo novas necessidades, novos equipamentos (objetos), que nascem para atender as expectativas de oferecer ferramentas com finalidades bem específicas, como por exemplo: Impressoras, Digitalizadoras, Copiadoras e etc.

Observem que não há uma especificação de marca, modelo e ou capacidades de execução das classes citadas acima, isto é o que consideramos o nível mais abstrato da orientação a objetos, que denominamos como **interfaces**.

Ilustração de interfaces dos equipamentos citados acima:



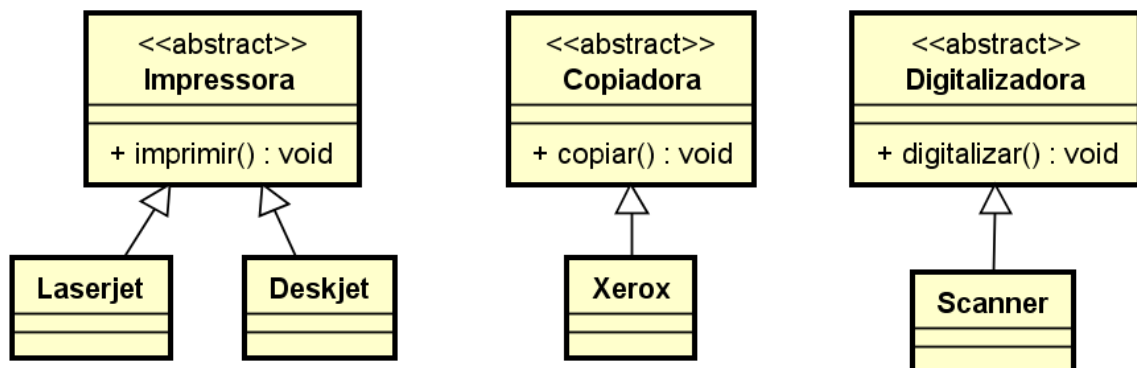
Então, o que você está dizendo é que **interfaces**, é o mesmo que **classes**? Um molde para representação dos objetos reais?

Como citado acima, **Java não permite herança múltipla**, logo, vamos imaginar que, **recebemos o desafio de projetar uma nova classe, para criar objetos que representam as três características citadas acima e que iremos denominar de EquipamentoMultifunional**.

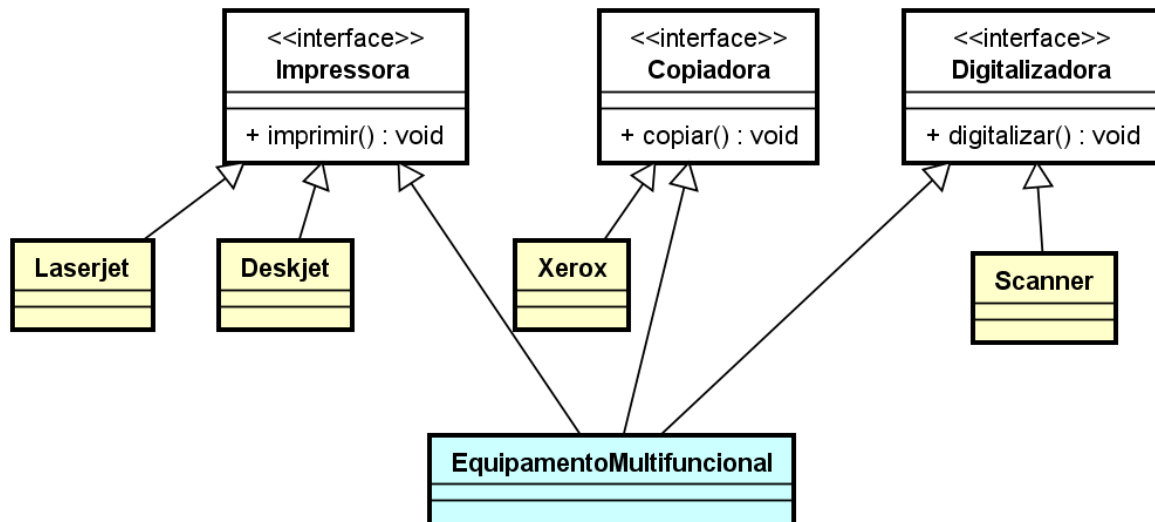


Vamos visualizar o diagrama:

Cenário 1:



Cenário 2:



Abaixo um exemplo de como ficaria a implementação:

The screenshot shows an IDE with several Java files open. The **Copiadora.java** file contains the following code:

```

1 package equipamentos.copiadora;
2
3 public interface Copiadora {
4     // Método obrigatório de copiadoras
5     public void copiar();
6 }
7
8

```

The **Impressora.java** file contains the following code:

```

1 package equipamentos.impressora;
2
3 public interface Impressora {
4     // Método obrigatório de impressoras
5     public void imprimir();
6 }
7
8

```

The **Digitalizadora.java** file contains the following code:

```

1 package equipamentos.digitalizadora;
2
3 public interface Digitalizadora {
4     // Método obrigatório de digitalizadoras
5     public void digitalizar();
6 }
7
8

```

The **EquipamentoMultifuncional.java** file contains the following code:

```

1 package equipamentos.multifuncional;
2
3 // Importando interfaces para implementar
4 import equipamentos.copiadora.Copiadora;
5 import equipamentos.digitalizadora.Digitalizadora;
6 import equipamentos.impressora.Impressora;
7
8 public class EquipamentoMultifuncional
9 implements Copiadora, Digitalizadora, Impressora {
10
11     // Criando método obrigatório de copiadoras
12     public void copiar() {
13         System.out.println("Copiando via Multifuncional");
14     }
15     // Criando método obrigatório de digitalizadoras
16     public void digitalizar() {
17         System.out.println("Digitalizando via Multifuncional");
18     }
19     // Criando método obrigatório de impressoras
20     public void imprimir() {
21         System.out.println("Imprimindo via Multifuncional");
22     }
23 }
24
25

```

The **Fabrica.java** file contains the following code:

```

1 package estabelecimento;
2
3 // Importando interfaces criadas para testar uso
4 import equipamentos.impressora.Impressora;
5 import equipamentos.digitalizadora.Digitalizadora;
6 import equipamentos.copiadora.Copiadora;
7
8 // Importando equipamento multifuncional
9 import equipamentos.multifuncional.EquipamentoMultifuncional;
10
11 public class Fabrica {
12
13     public static void main(String[] args) {
14         // Declarando equipamento multifuncional
15         EquipamentoMultifuncional em = new EquipamentoMultifuncional();
16
17         // Atribuindo equipamento multifuncional às interfaces específicas
18         Impressora impressora = em;
19         Digitalizadora digitalizadora = em;
20         Copiadora copiadora = em;
21
22         // Realizando operações das interfaces através do equipamento
23         // Multifuncional
24         impressora.imprimir();
25         digitalizadora.digitalizar();
26         copiadora.copiar();
27 }
28

```

The **Console** window shows the output of the program:

```

<terminated>-Fabrica [Java Application] /usr/lib/jvm/jdk-17-oracle-x64/bin/java (14 de ago. de 2023 17:40:41 - 17:40:42) [pid: 19973]
Imprimindo via Multifuncional
Digitalizando via Multifuncional
Copiando via Multifuncional
  
```

E para encerrar, uma das mais importantes ilustrações, quanto ao uso de interfaces para, desenvolvimento de componentes revolucionários, é apresentado em 2007 por nada mais nada menos que Steve Jobs ao lançar o primeiro **iPhone** da história.

Steve Jobs apresenta primeiro iPhone legendado (2007)



Link da referência da aula completa

Interface

J Java Básico

Interface

<https://glysns.gitbook.io/java-basico/programacao-orientada-a-objetos/interface>

Powered By GitBook

▼ Links e materiais adicionais

▼ Tarefas adicionais

- ☐ Salvar arquivos adicionais na pasta referente dentro do GitHub