# Capítulo 3 - Programação Orientada a Objetos

Aprender Java não garante que o projeto de software terá uma qualidade considerável, em termos de segurança, produtividade, manutenibilidade, ou seja, de fácil manutenção e reusabilidade. Aprender a fundo conceitos de programação orientada a objetos é essencial para a formação de um profissional da computação versátil, produtivo e que saiba lidar com os diversos problemas encontrados no mundo da programação. A POO foi criada para estreitar a relação do mundo real com o mundo virtual, com a ideia de simular conceitos, formas, estruturas e comportamentos de elementos do mundo real em um computador. O mundo é composto por objetos (ou “coisas”) e é de costume do homem classificar tais objetos. A cada novo objeto descoberto, uma nova classificação é definida. Nada mais justo do que criar um paradigma de programação que utilize objetos, não é?

## Aula 1 - Classes, Objetos e Instâncias

### 1.1 - Classes

Classe é o conceito base da POO e infelizmente é o mais complicado de aprender. Classe é a abstração de objetos do mundo real. Simples? Talvez. Para entender POO vamos definir o que é uma abstração.

#### 1.1.1 - Abstração



Figura 1. 1 – Paul Bailey: Terminal Light

<https://br.pinterest.com/pin/53269208066991767/>

Você já foi a um museu de artes plásticas? Lá podemos encontrar diversos tipos de pintura, desde as realistas, onde conseguimos entender com clareza o que ela representa, até a arte abstrata. Geralmente quando vemos uma arte abstrata pensamos: “Este borrão de tinta é arte? “. Na verdade, não é um mero borrão de tinta, pois existem mais coisas representadas naquele quadro do que conseguimos entender. Na Figura 1.1, o artista quis representar uma paisagem com poucas pinceladas. Ele abstraiu a paisagem a um nível tão alto que ela se reduziu a algumas pinceladas em amarelo, preto e azul. Para muitos não passa de um borrão de tintas, mas esta obra representa uma paisagem.

O fenômeno de simplificar ou não se ater a certos detalhes para reter apenas a informação básica é chamado de abstração. Usa a estratégia de que detalhes concretos são deixados vagos ou indefinidos, contando com a intuição e experiência para a interpretação destes detalhes omitidos.

Quanto mais alto o nível de abstração, menos detalhes teremos sobre as informações do objeto e consequentemente mais intuição e/ou experiência é requerida para que haja uma interpretação correta sobre as informações deste objeto.

Em programação orientada a objetos, classes são abstrações de objetos (ou “coisas”) do mundo real. Vamos agora entender o que realmente são as classes.

#### 1.1.2 - Classes

Nos capítulos anteriores tivemos vários exemplos de classe. Por exemplo, a classe Car era uma representação de um carro (objeto do mundo real). Claramente aquilo não é este veículo e existem milhares de detalhes que um carro possui, mas a classe Car os omitiu, pois ela é uma abstração de um carro real. Intuitivamente chamamos ela de carro, mas ela é apenas uma representação dele.

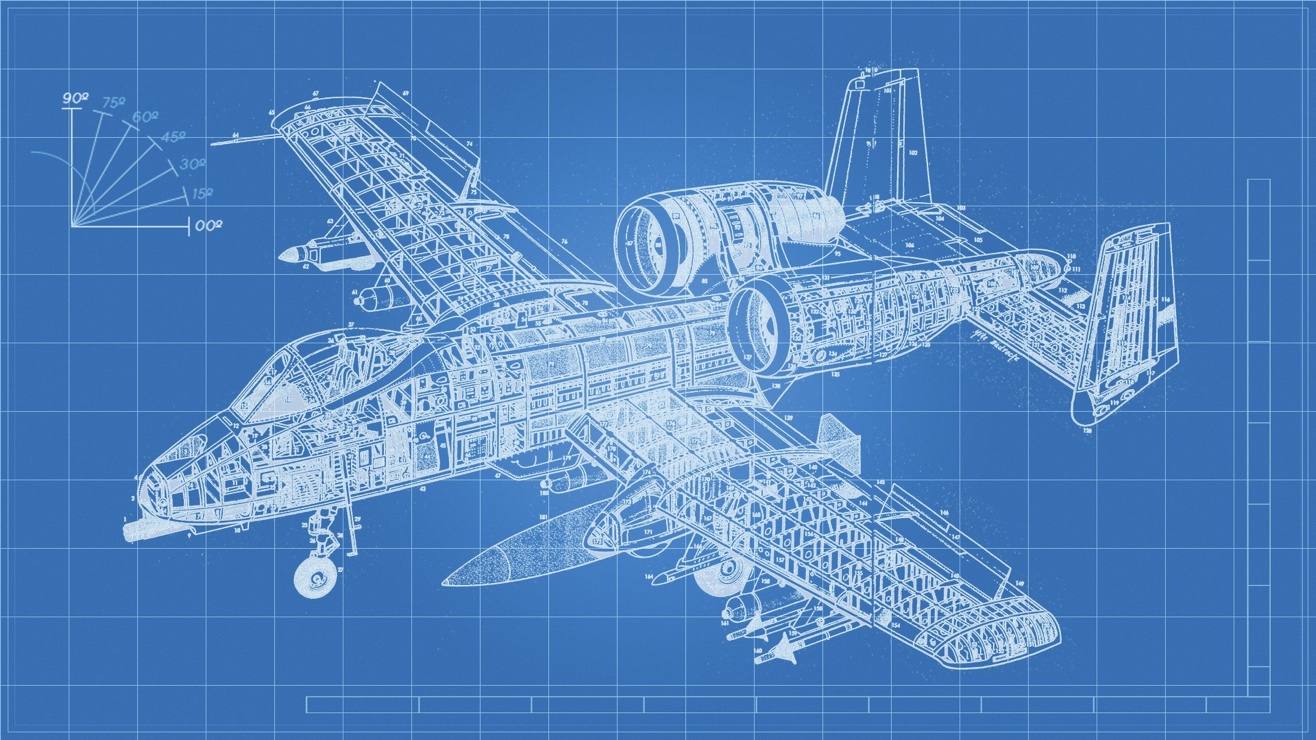


Figura 1.2 – Blueprint ilustrativo de um avião

<https://alpha.wallhaven.cc/wallpaper/184311>

A imagem acima é um blueprint de um avião. O blueprint é um tipo de tela ou suporte utilizado em desenho técnico para projetos de arquitetura, engenharia ou design. Ele é bastante similar a uma planta de casa, para quem nunca viu um desses.

Uma planta de casa por exemplo, tem a serventia de representar as medidas e dimensões de uma casa. Nela encontramos informações necessárias para termos uma ideia de como seria uma casa. Na imagem acima vemos como seria um avião. O blueprint é uma abstração do objeto a ser projetado, pois ele apresenta somente as informações que são relevantes ao projetista. Não podemos ver, por exemplo, a cor que o avião será, mas ainda sim um avião está representado ali.

Classes são como modelos ou blueprints que estão encarregados de descrever as características relevantes de um elemento. Eles podem ter qualquer quantidade de métodos para acomodar os níveis de acessibilidade de outros métodos.

No entanto, classes estão por traz da razão do porquê um programador que entrou na metade de um projeto de software entende o código do antigo programador do projeto. Já que os modelos definem os elementos, é menos desafiador para o novato entender os padrões daquele projeto, afinal a programação orientada a objetos também segue padrões. O uso de classes faz com que a estrutura do código seja mais clara e bem definida, mesmo para iniciantes no assunto.

### 1.2 - Objetos

Em uma definição rápida, objetos são coisas materiais que podem ser percebidas pelos sentidos. Em programação de objetos são elementos que possuem estados e comportamentos. Uma vez definidos, os elementos podem existir com suas próprias características, pois isto garante a eles a oportunidade de agregar algum valor a um componente do programa sem que haja a necessidade de uma estrutura muito extensa. Consequentemente, objetos são instâncias de classes.



Figura 1. – Diversos objetos

http://hellosculptureone.tumblr.com/page/2

Um objeto é capaz de armazenar estados através dos atributos, receber e manipular mensagens de outros objetos através de métodos e ainda podem apresentar comportamentos variados de acordo a situação/ambiente em que ele se encontra e/ou responsabilidades que a ele são designadas.

Métodos podem ser executados corretamente devido aos comportamentos e estados dos objetos. Para a realização de uma tarefa, um objeto em particular é associado a um comando ou função únicos. Desta forma, eles podem aderir certas instruções.

Classes e objetos estão sempre sendo mencionados juntos, devido a relação forte entre eles. Então não é incomum que os programadores iniciantes confundam ambos como sendo a mesma coisa.

Por exemplo, no capítulo anterior tivemos o exemplo Carro. Mas Carro é um objeto ou uma classe? Isso depende do ponto de vista. Um carro pode ser uma classe ou um objeto. Se estamos falando de uma instância Carro, é um objeto, outrora falando de tipo Carro, é uma classe. A confusão geralmente é na prosa, e não na prática da programação.

Uma consideração importante a se levar quando comparar as duas coisas é que classes não possuem um ciclo de vida. Objetos nascem, crescem e morrem, ou melhor, são inicializados, realizam suas tarefas e no fim são desalocados.

Assim, classes são elementos genéricos na programação. Em contrapartida objetos contém descrições específicas de um elemento. Descrições que diferenciam um objeto do outro, e a sua presença em um programa é sempre limitada. Se o objeto não está sendo mais utilizado, ele é descartado. O Java possui um recurso chamado Garbage Collector que faz o gerenciamento de memória do programa. Quando um objeto não está sendo mais usado ou sua referência foi perdida, o Garbage Collector elimina este objeto da memória do computador. Isto facilita a codificação de programas, pois os programadores não terão que desalocar os objetos da memória manualmente, e contribui para o aumento da performance do programa.

### 1.3 – Instâncias

Considere instância como uma outra maneira de dizer objeto. Um objeto que ocupa um espaço de memória é uma instância. Então para representar computacionalmente o objeto, usa-se o nome instância. Como visto nos capítulos anteriores, existem definições mais profundas para o que chamamos de instância. A principal é que ela é uma forma de diferenciar um objeto do outro. Vamos ao exemplo.

Suponha que definimos uma classe Carro que possui apenas dois atributos: marca e cor. E que uma fábrica de carros irá construir vários objetos Carro de mesma marca e mesma cor. Como iremos diferenciá-los? Podemos ter vários objetos Carro da marca Gol e cor branca, mas no computador eles são diferenciados entre si pois como dito acima, podem possuir as mesmas características, mas estão armazenados em lugares diferentes na memória do computador, então chamamos de instâncias. Objeto é uma instância da classe. Logo instância é um objeto.

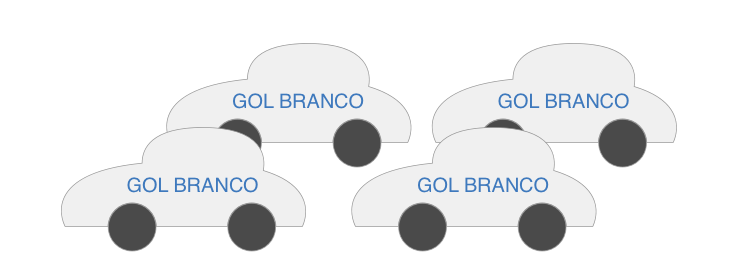


Figura 1. – Ilustração de vários carros com características idênticas

Agora imagine que existem dois objetos Carro. Um é Gol branco e outro é Gol preto. Possuem características iguais? Sim, mas são instâncias? Sim também. Apesar do atributo cor ser a característica que os diferenciam o que acontece se o dono do Gol preto quiser pintá-lo de branco? O computador ainda terá a referência do espaço de memória (que – substituir) onde os dois objetos estão armazenados e conseguirá diferenciá-los, por isso, mais uma vez instância é uma nova maneira de dizer objeto, mas com um teor mais computacional.

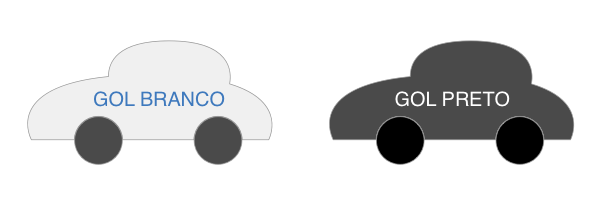


Figura 1. - Ilustração de carros com características diferentes

### 1.4 – Encapsulamento

Encapsulamento é uma técnica utilizada para não expor os dados internos de uma classe para seu usuário, tornando partes do código mais independentes e modularizadas. A modularização faz com que o código seja divido em módulos, ou blocos de código, independentes entre si. O encapsulamento geralmente é utilizado para restringir ações das classes de modo que elas atendam a regra de negócio de uma maneira segura. Por exemplo: imagine que temos uma classe Pessoa, e nela temos o atributo idade e ano de nascimento. Se uma pessoa nasceu em 1996 não poderia ter 30 anos de idade. Se o sistema permite que o usuário forneça tanto a idade quanto o ano de nascimento, ele deve pelo menos fazer a validação destes valores antes de cadastrá-lo. Vamos ver como ficaria em Java.

Primeiro criamos a classe Person e deixamos os seus atributos como privado para que o usuário não consiga utilizá-los livremente.

1. /\*\*
2. \* Classe Pessoa
3. \* @author pearson
4. \*/
5. **public** **class** Person {
6. **private** Integer birthYear;  // Ano de nascimento
7. **private** Integer age;        // Idade
8. }

Listagem 1. 1

Agora deveremos fazer o controle dos dados que serão atribuídos a idade e ano de nascimento. Para isto criaremos o encapsulamento dos atributos utilizando métodos getter e setter (obtentores e atribuidores respectivamente).

Getters e setters podem ser gerados automaticamente clicando com o botão direito dentro da classe, em seguida **Inserir Código** e depois **Getter() e Setter().**

A classe ficará da seguinte forma:

1. /\*\*
2. \* Classe Pessoa
3. \* @author pearson
4. \*/
5. **public** **class** Person {
6. **private** Integer birthYear;  // Ano de nascimento
7. **private** Integer age;        // Idade
9. **public** Integer getBirthYear() {
10. **return** birthYear;
11. }
13. **public** **void** setBirthYear(Integer birthYear) {
14. **this**.birthYear = birthYear;
15. }
17. **public** Integer getAge() {
18. **return** age;
19. }
21. **public** **void** setAge(Integer age) {
22. **this**.age = age;
23. }

26. }

Listagem 1. 2

Ainda não há o tratamento para manter a coesão entre idade e ano de nascimento.

1. /\*\*
2. \* Classe Pessoa
3. \* @author pearson
4. \*/
5. **public** **class** Person {
6. **private** Integer birthYear = 0;  // Ano de nascimento
7. **private** Integer age = 0;        // Idade
9. **public** Integer getBirthYear() {
10. // Garante que a data de nascimento estará sempre atualizada com a idade
11. Integer currentYear = Calendar.getInstance().get(Calendar.YEAR);
12. **return** currentYear - **this**.age;
13. }
15. **public** **void** setBirthYear(Integer birthYear) {
17. // Método Java que retorna o ano atual
18. Integer currentYear = Calendar.getInstance().get(Calendar.YEAR);
20. **if** (birthYear <= currentYear) { // Valida se não é um ano futuro
21. **this**.birthYear = birthYear;
22. **this**.age = currentYear - **this**.birthYear;
23. }**else**{
24. System.out.println("Ano de nascimento inválido");
25. }
27. }
29. **public** Integer getAge() {
30. // Garante que a idade estará sempre atualizada com a data de nascimento
31. Integer currentYear = Calendar.getInstance().get(Calendar.YEAR);
32. **return** currentYear - **this**.birthYear;
33. }
35. **public** **void** setAge(Integer age) {
37. // Método Java que retorna o ano atual
38. Integer currentYear = Calendar.getInstance().get(Calendar.YEAR);
40. **if** (age <= currentYear) {   // Valida se é uma idade possível
41. **this**.age = age;
42. **this**.birthYear = currentYear - **this**.age;
43. }**else**{
44. System.out.println("Idade inválida");
45. }
46. }
47. }

Listagem 1. 3

A partir daqui não se esqueça de importar as novas bibliotecas. Geralmente a IDE aponta a falta do import e nos dá a opção de importação. Adicione “import java.util.Calendar;” para o exemplo da Listagem 1.3.

Gaste um pouco do seu tempo interpretando este código. Dentro dos métodos getters e setters foram colocadas validações para as variáveis. Uma vez que a idade é atualizada, o ano de nascimento também atualiza e vice-versa. Vamos testar a classe Person no método main da classe Main:

1. **public** **static** **void** main(String[] args) {
3. Person person = **new** Person();
5. person.setAge(18);
6. System.out.println("Quem tem " + person.getAge() + " anos nasceu em " + person.getBirthYear());
8. person.setAge(24);
9. System.out.println("Quem tem " + person.getAge() + " anos nasceu em " + person.getBirthYear());
11. person.setBirthYear(1998);
12. System.out.println("Quem nasceu em " + person.getBirthYear() + " tem " + person.getAge() + " anos");
14. person.setBirthYear(1992);
15. System.out.println("Quem nasceu em " + person.getBirthYear() + " tem " + person.getAge() + " anos");
16. }

Listagem 1. 4

Execute o código e perceba que qualquer alteração que você fizer com a idade e ano os dados se manterão consistentes. A partir de agora, todas as classes terão que ser encapsuladas, ok?

### 1.5 – Resumo

No capítulo anterior vimos o básico de alguns conceitos de programação orientada a objetos. Nesta aula, foi dada uma introdução do que é abstração e aprendemos mais a fundo sobre Classes, Objetos, Instâncias e Encapsulamento. Você viu que os métodos setter e getter servem para controlar o que entra e o que sai do objeto e como é o seu uso em Java.

### 1.6 – Exercícios

|  |  |
| --- | --- |
| 1. | O que você entende por abstração? Qual a relação com POO? |
| 2. | O que são Classes em POO? Explique detalhadamente |
| 3. | O que são Objetos em POO? Explique detalhadamente |
| 4. | O que são Instâncias em POO? Explique detalhadamente |
| 5. | O que é encapsulamento? Explique com o exemplo da classe Pessoa |
| 6. | O que é getter? E setter? |
| 7. | O que diferencia um Objeto de outro? |
| 8. | No cotidiano, o que pode ser comparado a uma Classe? E Objetos? |

### 1.7 – TDP

#### O Prazo

Não há gente o suficiente para que o projeto seja finalizado dentro do prazo! Por enquanto todos os desenvolvedores estão alocados em outros projetos.

Um dos maiores problemas em um projeto de software. Um código profissional e com qualidade necessita de tempo e dedicação. Quando há um prazo apertado, geralmente contrata-se mais pessoas para desenvolver, porém as vezes um projeto não suporta tanta gente.

Quando há um excesso de desenvolvedores, os trabalhos podem ser conflitantes e a paralelização de atividades torna-se mais difícil.

Outra solução para atender um prazo apertado é diminuir a quantidade de artefatos.

Mas afinal o que são os artefatos?

#### Artefatos

Um artefato é qualquer documento ou algo que dê valor ao projeto. A próxima etapa seria a arquitetural. O arquiteto já identificou as classes, métodos e atributos e agora deveria colocá-las em um diagrama. Como estamos com o prazo apertado, vamos pular esta etapa e deixá-la para o final.

Para suprir a necessidade de mais desenvolvedores, você irá programar.

#### Tarefa

Implemente as classes identificadas. Teste todos os construtores, getters e setters.

Aula 2

## Herança e Interface

### 2.1 – Introdução

O mundo da POO é complexo devido a toda informação e responsabilidade que carrega, mas vamos facilitar alguns pontos. Daqui em diante veremos a parte mais importante do assunto, que te possibilitará criar códigos mais flexíveis para que você possa realizar alterações bem como facilitar a interação de outros programadores com o código que você escreveu, garantindo aqui maior tempo e agilidade.

Aprendemos até agora representar coisas do mundo real e manipulá-las, mas programação orientada a objetos não passa de uma forma de representar os dados? Não, vai muito além disso.

Existem diversas técnicas e padrões a serem utilizados em POO. Classes e objetos são apenas ferramentas destas técnicas e as primeiras a serem apresentadas serão **Herança** e **Interface**.

### 2.2 – Herança

Primeiro vamos definir o que é herança. Existem algumas definições fora da computação que podem ser importantes: genética e patrimonial.

* **Herança patrimonial:** É um conjunto de princípios jurídicos que formalizam perante a lei a distribuição do patrimônio aos sucessores legais de uma pessoa que morreu. Em outras palavras, são bens de uma pessoa que morreu deixados para seus sucessores.
* **Herança genética:** É processo pelo qual um [organismo](https://pt.wikipedia.org/wiki/Organismo) ou [célula](https://pt.wikipedia.org/wiki/C%C3%A9lula) adquire ou torna-se predisposto a adquirir características semelhantes à do organismo ou célula que o gerou, através de informações codificadas ([código genético](https://pt.wikipedia.org/wiki/C%C3%B3digo_gen%C3%A9tico)) que são transmitidas à descendência.

Como programação orientada a objetos tem o intuito de imitar a vida real, a herança em POO é fortemente relacionada a herança genética, mas não se aplica somente a seres biológicos. Em outras palavras a herança em computação é a predisposição de um objeto herdar características de outro.

Agora tente responder as seguintes perguntas:

* Como você representaria um gato e um leão, usando uma estrutura de herança?
* O gato seria a versão especializada de um leão?
* Se ambos herdassem características de uma mesma classe, quais seriam?

Vamos lá, o leão e o gato possuem diversas características semelhantes. Em um primeiro momento, poderíamos dizer que o leão é melhor dotado do que um gato, pois ele possui diversas características que colocadas lado-a-lado com as características de um gato definiríamos que o leão é um ser evoluído do gato, já que é mais forte, caçador e menos vulnerável. Então a herança seria que o leão herda do gato. Veja a seguinte representação:



Figura 2. - Exemplo de herança com baixa semântica

Ref Gato: <http://www.iconarchive.com/show/ios7-icons-by-icons8/Animals-Cat-icon.html>  
  
Ref Leão: <https://dribbble.com/shots/1889217-Lion>

O sentido da herança é reaproveitar as características de uma outra classe, mas ela tem que fazer sentido *semanticamente, ou seja, a representação deve fazer sentido no nosso sistema de linguagem natural (dizer que um leão é um gato não faz sentido)*. Sabemos que gatos tem características particulares que um leão não tem e vice-versa. Para a herança fazer sentido, a classe que especializa (filha ou subclasse) a classe especializada (mãe ou superclasse) tem que afirmar a relação “É UM”, ou seja, na herança acima, estamos dizendo que um Leão “É UM” Gato. Sabemos que isto é falso, então pense em alguma outra relação de herança entre eles.

Dica: você pode criar novas classes e colocá-las na relação de herança.

Veja uma sugestão (POO é vasta, não existe apenas uma solução para o problema):

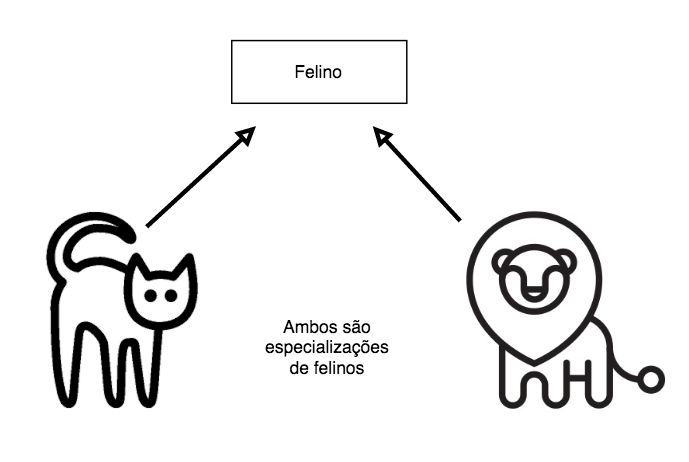


Figura 2. – Herança bem definida semânticamente

Aqui criamos a classe Felino, Gato “É UM” Felino e Leão “É UM” Felino, então esta herança faz sentido semanticamente. Existem características que definem um Felino, e Gato e o Leão compartilham destas características. Este foi um exemplo de herança e do relacionamento “É UM”.

Quando projetamos um código utilizando herança, devemos inserir todo código comum em uma classe e informar as classes mais específicas que a classe comum (mais abstrata) é a sua superclasse. Quando dizemos que uma subclasse herda de uma superclasse, também podemos dizer que ela estende ou especializa a superclasse. Isto significa que a subclasse possuirá todos métodos e atributos de sua superclasse.

Para especificar uma herança em Java, basta utilizar a palavra reservada extends na declaração da classe, como no seguinte exemplo:

**public** **class** MySubclass **extends** MySuperclass

#### 2.2.1 - A classe Object

No subtítulo 3.2.3 da aula 3 (Testando os atributos da classe Car) da unidade 2 foi mencionado o método toString() e dizemos que ele era pertencente a classe Object. Naquele momento você deve ter sentido pelo menos um pouco de confusão, mas eis a explicação.

Agora que sabemos o que é herança em POO, existe uma classe chamada Object, e ela seria a “mãe de todos os objetos”. A classe Object é a mais abstrata em Java, e todas as outras classes são especialização de Object (ou herdam de Object). Isto porque existem métodos que são comuns para todos os objetos, como por exemplo o toString() (que converte qualquer objeto em String), o isEqual() (que é utilizado para comparar um objeto com o outro) e etc. Existem vários métodos em comum e que são obrigatório para todas as classes, por isso herda-se de Object.

Já que todas as classes herdam de Object, o Java permite a omissão da instrução extends para a classe Object, ou seja, não é necessário informar que uma classe herda de Object pois já é implícito.

### 2.3 – Herança Múltipla

Agora que você entendeu herança, vamos dar o exemplo Animal. Na imagem abaixo mostraremos a árvore de herança mais detalhada do que a vista no subtítulo 2.2 desta aula.

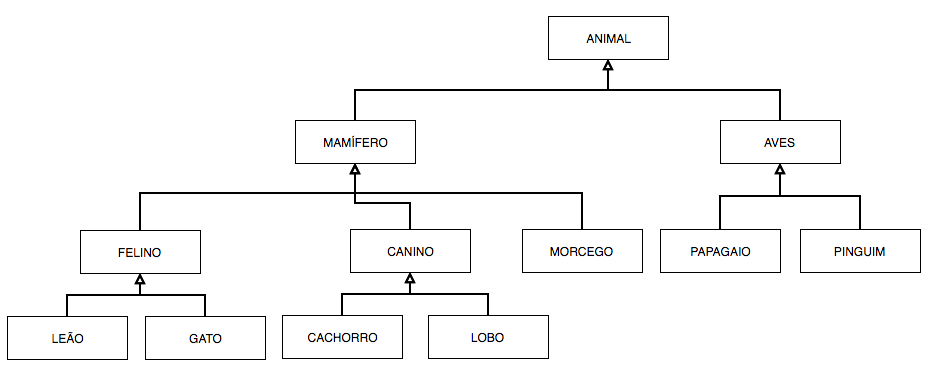


Figura 2.- Árvore de herança do reino animal

Na Figura 2.3, temos uma estrutura de herança bem consistente. Vemos uma árvore de especialização que mostra diversos níveis: Animal, Mamíferos, Felino, Gato e etc. Agora resolva o seguinte problema:

Gato, Cachorro e Papagaio são domésticos. Como poderíamos representar os animais domésticos na estrutura de herança? Você herda características de seu pai e sua mãe, não é? Então intuitivamente faríamos uma herança múltipla como na Figura 2.4 a seguir.

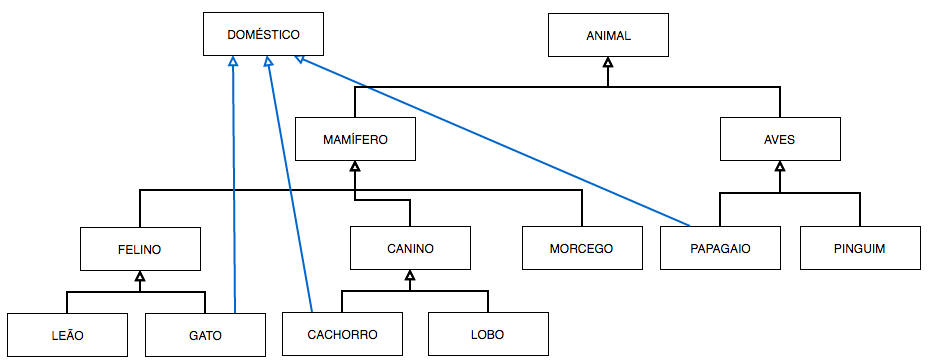


Figura 2. – Árvore de herança com exemplo de herança múltipla

Inserimos uma classe Doméstico e forçamos que Papagaio, Gato e Cachorro especializem ela além de suas especializações originais. Isto se chama *herança múltipla*, mas ela é **extremamente perigosa.**

Com herança múltipla, erros de projeto são mais propícios. A cada herança múltipla criada pode levar a algumas complexidades desnecessárias, pois você precisará de adicionar regras especiais para lidar com possíveis ambiguidades. Em Java não é possível fazer o uso de herança múltipla. Ela trouxe as *Interfaces* para nos salvar.

### 2.4 – Interfaces

Herança múltipla apresenta alguns problemas como dito acima, por isso Java usa interfaces ao invés disto. Interface é um recurso que obriga um determinado grupo de classes a seguirem um contexto em comum. Uma Interface implica que a classe que a implementa deverá ter métodos e atributos determinados pela interface, contudo estes métodos podem ser implementados de uma forma diferente para cada classe. A grosso modo, uma Interface é um contrato que quando assinado pela classe ele deverá ser cumprido.

Nosso exemplo dos animais usando representação com interface ficaria da seguinte maneira (Figura 2.5):

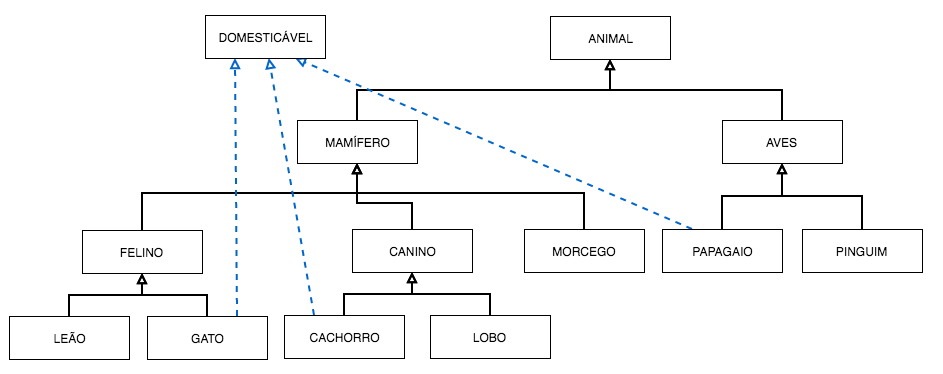


Figura 2. – Árvore de herança com exemplo de interface

Mudamos a linha das setas para não confundirmos e também nomeamos a classe Doméstico com o sufixo “ável”.

*DICA: Como boa prática para identificarmos uma Interface é nomeá-las com o sufixo “ável” (no código faríamos em inglês, “able”), além de indicar que interfaces dão habilidades às classes de ser ou fazer algo.*

Agora identifique os seres voadores, dentre os mostrados anteriormente, e relacione-os em uma interface.

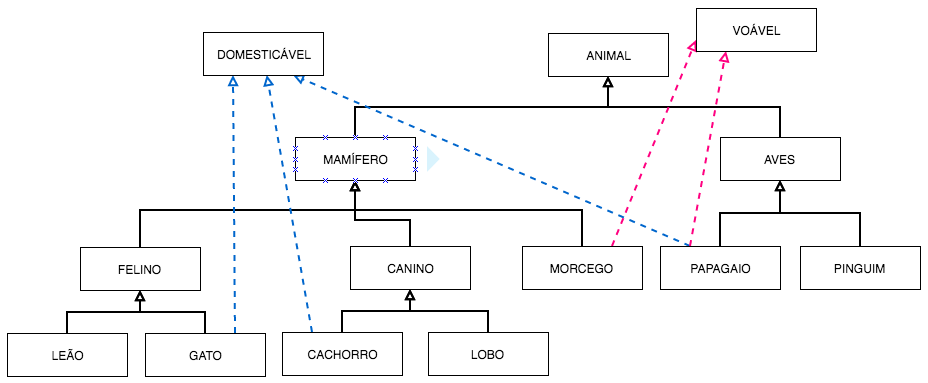


Figura 2. – Exemplo de boa prática de nomenclatura de interafaces

### 2.5 – Resumo

Nesta aula, aprendemos o que é herança em POO e que a classe Object é a "mãe de todos os objetos". Você viu que herança múltipla apresenta alguns problemas e é difícil manter uma boa semântica utilizando ela. Ao invés disso, utilizamos interfaces dadas as vantagens que elas apresentam em relação à herança múltipla.

### 2.6 – Exercícios

|  |  |
| --- | --- |
| 1. | O que é herança em POO? |
| 2. | Como especificar uma herança Java? |
| 3. | O que é classe Object? |
| 4. | Qual é a classe que todas as outras classes estendem? |
| 5. | O que é herança múltipla? Por que não é utilizada em Java? |
| 6. | O que é interface? |
| 7. | Dê um exemplo, diferente do livro, de herança. |
| 8. | Dê um exemplo, diferente do livro, de interface. |

### 2.7 – TDP

Nelson já voltou da licença médica e você voltou para o cargo de programador.

Seu desempenho como gerente de projetos foi ótimo. E ao final do mês receberá uma bonificação no salário.

#### 2.7.1 - Desenvolvimento

Você agora é oficialmente programador Java da empresa. Todos os outros programadores preferem usar linguagem procedural. Foi difícil convencê-los a usar Java.

Neste projeto você vai ajudá-los a firmar a idéia de que a POO é melhor do que a programação procedural.

#### Tarefa

Identifique e implemente heranças e interfaces no código, se necessário

Aula 3

## Polimorfismo

Na aula anterior aprendemos sobre os benefícios que as heranças e interfaces nos proporcionam. Nesta aula será apresentado o conceito de polimorfismo, que é um recurso que o Java nos proporciona quando fazemos uso de herança e interfaces. Explicaremos o conceito de polimorfismo e em seguida mostraremos em um exemplo prático, a aplicação de polimorfismo em um sistema real que gerência funcionários.

### 3.1 - Polimorfismo

Polimorfismo é o fenômeno ou poder de assumir diferentes formas. Em programação trata-se da *capacidade de um objeto ser referenciado de diversas formas*, mas não se confunda, polimorfismo não quer dizer que que o objeto poderá se transformar e mudar seu tipo, pelo contrário, um objeto nasce, cresce e morre sendo do mesmo tipo. O que muda é a maneira como nos referimos a ele. Vamos -ao exemplo prático.

O programa que iremos criar irá gerenciar o salário e a bonificação de diferentes funcionários de uma empresa. Um funcionário deverá ter um nome, salário, cargo e bonificação de salário. Existirão 3 cargos: Gerente, Supervisor e Desenvolvedor. Cada tipo de cargo receberá uma bonificação salarial diferente: 30% para Gerente, 25% para Supervisor e 20% para Desenvolvedor. Mãos na massa:

Vamos criar um novo projeto com o nome **Company** seguindo os mesmos passos das aulas anteriores. Crie o pacote **Classes** e em seguida crie a classe **Employee** no pacote Classes.

Após criar a classe Employee (Funcionário), antes de tudo vamos criar as variáveis membro (mais um nome para atributos), não se esquecendo de deixá-las private.

1. **private** String name;    // Nome do funcionário
2. **private** Double salary;  // Salário do funcionário
3. **private** **int** position;   // Cargo do funcionário

Listagem 3.

Neste exemplo, não precisaremos de alterar os atributos. Uma vez inicializados, não alteraremos mais, então já que deixamos como private e não serão alterados, vamos criar um construtor. Na próxima linha clique com o botão direito do mouse e escolha a opção **Inserir Código > Construtor** e selecione todos os campos para serem acessados pelo construtor. Confira:

1. // Construtor, encapsulamento
2. **public** Employee(String name, Double salary, **int** position) {
3. **this**.name = name;
4. **this**.salary = salary;
5. **this**.position = position;
6. }

Listagem 3. 2

Agora que criamos a estrutura básica da classe precisamos calcular o bônus dos funcionários de acordo com a sua posição. Uma maneira fácil seria assim:

1. // Retorna a bonifcação salarial de acordo com o cargo.
2. **public** Double getBonus() {
4. **switch** (**this**.position) {
5. **case** 1:     // Gerente
6. **return** **this**.salary \* 0.3;   // 30%
7. **case** 2:     // Supervisor
8. **return** **this**.salary \* 0.2;  // 20%
9. **case** 3:     // Desenvolvedor
10. **return** **this**.salary \* 0.1;   // 10%
12. }
14. **return** 0.0; // Zero, se não for um dos três cargos
15. }

Listagem 3. 3

Agora nossa classe Employee está pronta (pelo menos por enquanto), vamos testá-la.

Na classe Main, localize o método main e crie três funcionários com diferentes salários e diferentes cargos. Exiba o bônus de cada um:

1. **public** **static** **void** main(String[] args) {
3. // João é gerente
4. Employee joao = **new** Employee("João", 2000.0, 1);
5. System.out.println("O bônus de João é " + joao.getBonus());
7. // José é supervisor
8. Employee jose = **new** Employee("José", 3000.0, 2);
9. System.out.println("O bônus de José é " + jose.getBonus());
11. // Jacó é desenvolvedor
12. Employee jaco = **new** Employee("Jacó", 1500.0, 3);
13. System.out.println("O bônus de Jacó é " + jaco.getBonus());
14. }

Listagem 3. 4

Execute o código e veja que funciona corretamente. Mas veja, está bastante confuso saber quem é Gerente, Desenvolvedor e Supervisor. Apenas um número não deixa intuitivo para o programador saber de que cargo se trata.

A Programação Orientada a Objetos foi feita exatamente para que isto seja evitado, o intuito é representar coisas reais através de objetos. Somos programadores OO e temos que deixar nosso código mais sofisticado. Agora que aprendemos herança, podemos utilizá-la a nosso favor.

Veja a seguinte árvore de herança:

*DICA: Estas árvores de herança chamam-se na verdade Diagrama de Classes UML – que você irá aprender melhor no capítulo 4.*

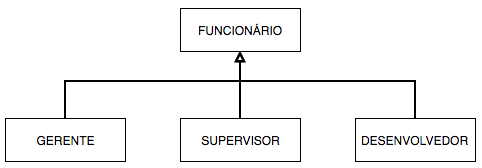


Figura 3. – Diagrama de classes simplificado da estrutura de uma Empresa

Vamos separar os cargos em classes. Desenvolvedor não é um funcionário? Gerente não é funcionário? Supervisor não é funcionário? Então vamos aproveitar as características em comum entre eles e usar a estrutura de herança mostrada anteriormente (Figura 3.1).

É hora de deixar nosso código mais sofisticado. Remova o atributo position (remova do construtor também) e altere o método getBonus(). Nossa classe Employee ficará mais abstrata, ou seja, simplificada.

1. **public** **class** Employee {
2. **protected** String name;    // Nome do funcionário
3. **protected** Double salary;  // Salário do funcionário
5. // Construtor, encapsulamento
6. **public** Employee(String name, Double salary) {
7. **this**.name = name;
8. **this**.salary = salary;
9. }
11. // Retorna a bonificação salarial de acordo com o cargo.
12. **public** Double getBonus() {
13. **return** 0.0; // Zero, se não for um dos três cargos acima
14. }
15. }

Listagem 3. 5

Veja que o método getBonus() agora retorna o valor 0.0 pois as subclasses de Employee farão sobreposição do método.

### 3.1.1 – Sobreposição e sobrecarga

Sobreposição e sobrecarga de método são alguns dos recursos que o polimorfismo nos oferece. Quando estendemos uma classe, temos a possibilidade de reescrever os métodos da superclasse para adequar às condições da subclasse. Isto é chamado de sobreposição. A sobrecarga permite que existam vários métodos com o mesmo nome, desde que eles possuam assinaturas (quantidade e tipos de parâmetros) diferentes. Por exemplo, se quisermos criar um método construtor recebendo apenas o salário como parâmetro, precisamos apenas adicionar mais um método com esta assinatura. Exemplo:

1. // Construtor, encapsulamento
2. **public** Employee(String name, Double salary) {
3. **this**.name = name;
4. **this**.salary = salary;
5. }
7. // Sobrecarga do construtor
8. **public** Employee(Double salary) {
9. **this**.salary = salary;
10. }

Listagem 3. 6

Agora crie as classes filhas de Employee e sobreponha o método getBonus() de acordo com a condição da subclasse.

Obs.: Crie os arquivos para a classe Manager, Developer e Supervisor no pacote Classes.

1. /\*\*
2. \* Classe Gerente
3. \* @author pearson
4. \*/
5. **public** **class** Manager **extends** Employee{  // Gerente é um Funcionário
7. **public** Manager(String name, Double salary) {
8. **super**(name, salary);
9. }
11. @Override       // Sobrepõe o método getBonus()
12. **public** Double getBonus() {
13. **return** **this**.salary \* 0.3;
14. }
16. }

Listagem 3. 7

1. /\*\*
2. \* Classe Supervisor
3. \* @author pearson
4. \*/
5. **public** **class** Supervisor **extends** Employee{  // Supervisor é um Funcionário
7. **public** Supervisor(String name, Double salary) {
8. **super**(name, salary);
9. }
11. @Override       // Sobrepõe o método getBonus()
12. **public** Double getBonus() {
13. **return** **this**.salary \* 0.2;
14. }
16. }

Listagem 3. 8

1. /\*\*
2. \* Classe Desenvolvedor
3. \* @author pearson
4. \*/
5. **public** **class** Developer **extends** Employee{  // Desenvolvedor é um Funcionário
7. **public** Developer(String name, Double salary) {
8. **super**(name, salary);
9. }
11. @Override       // Sobrepõe o método getBonus()
12. **public** Double getBonus() {
13. **return** **this**.salary \* 0.1;
14. }
16. }

Listagem 3. 9

Nitidamente temos mais linhas de código, mas também temos o código mais organizado e legível. Vamos ver como o método main da classe Main ficaria. Apenas troque os construtores.

1. **public** **static** **void** main(String[] args) {
3. Manager joao = **new** Manager("João", 2000.0);
4. System.out.println("O bônus de João é " + joao.getBonus());
6. Supervisor jose = **new** Supervisor("José", 3000.0);
7. System.out.println("O bônus de José é " + jose.getBonus());
9. Developer jaco = **new** Developer("Jacó", 1500.0);
10. System.out.println("O bônus de Jacó é " + jaco.getBonus());
11. }

Listagem 3. 10

Execute o código e veja que o resultado final será o mesmo, mas agora sabemos dizer e diferenciar cada cargo utilizando a herança e polimorfismo.

**Parabéns, se você chegou até aqui você já sabe o fundamental de Java**.

Mas não é o suficiente! Vamos deixar mais sofisticado.

Eu quero fazer uma calculadora de bonificação do salário. Ela simplesmente vai somar o bônus de cada funcionário. Para isto, criaremos a classe Calculator (Calculadora).

A classe Calculator irá armazenar o total de bônus que a empresa pagará aos funcionários. Ela conterá um método addBonus() que receberá o bônus do funcionário e o somará no atributo total.

1. **public** **class** Calculator {
3. **private** Double total = 0.0; // Inicializamos com zero.
5. **public** **void** addBonus(Double bonus) {
6. **this**.total += bonus;
7. }
9. // Getter
10. **public** Double getTotal() {
11. **return** **this**.total;
12. }
14. }

Listagem 3. 1

No método main da classe Main vamos testar a classe Calculator. Adicione o seguinte código:

1. // Calculando o bônus
2. Calculator calculator = **new** Calculator();
4. calculator.addBonus(joao.getBonus());
5. calculator.addBonus(jose.getBonus());
6. calculator.addBonus(jaco.getBonus());
8. System.out.println("O total de bônus a pagar é: " + calculator.getTotal());

Listagem 3. 2

No código da Listagem 3.12, estamos adicionando o bônus de cada funcionário no objeto calculator. Mas e se resolvermos adicionar -1.000.0 de bônus? Observe a seguir:

calculator.addBonus(-1000.0);

A calculadora aceitará e adicionará nas suas contas. Como evitar isto? Queremos garantir que a calculadora só calcule o bônus estipulado para cada funcionário. Para isto usaremos mais polimorfismo. O método getBonus() está presente em funcionário, não é? Então ao invés do método addBonus(), da classe Calculator , receber um Double de parâmetro, vamos fazer com que ele (o método addBonus()) receba o funcionário no qual terá seu bônus somado. O método addBonus() ficará da seguinte maneira:

1. **public** **void** addBonus(Employee employee) {
2. **this**.total += employee.getBonus();
3. }

Listagem 3. 3

E ajustando o seu uso no método main obteremos o seguinte código:

1. **public** **static** **void** main(String[] args) {
3. Manager joao = **new** Manager("João", 2000.0);
4. System.out.println("O bônus de João é " + joao.getBonus());
6. Supervisor jose = **new** Supervisor("José", 3000.0);
7. System.out.println("O bônus de José é " + jose.getBonus());
9. Developer jaco = **new** Developer("Jacó", 1500.0);
10. System.out.println("O bônus de Jacó é " + jaco.getBonus());
12. // Calculando o bônus
13. Calculator calculator = **new** Calculator();
15. calculator.addBonus(joao);
16. calculator.addBonus(jose);
17. calculator.addBonus(jaco);
19. System.out.println("O total de bônus a pagar é: " + calculator.getTotal());
20. }

Listagem 3.4

Execute o código e veja que obtivemos o mesmo resultado, porém agora está mais encapsulado e seguro de falhas.

Como vimos no exemplo, o polimorfismo permite que um objeto assuma diversas formas. Um objeto **Gerente** pode também ser um **Funcionário**, o mesmo com o objeto **Supervisor**. Ambos têm características em comum mas podem se comportar de maneira diferente em cada característica. Com o polimorfismo, pudemos deixar o sistema de herança flexível, com a sobrecarga e sobreposição, ou seja, demos comportamentos diferentes aos objetos utilizando um mesmo método. Assim um objeto pode assumir diversas formas e flexibilizar seu tratamento.

Agora sabemos os fundamentos iniciais de Java e os conceitos básicos de POO, aprenderemos conceitos mais aprofundados.

### 3.2 – Métodos estáticos

Nem tudo em POO deverá ser necessariamente um objeto. Métodos estáticos independem das variáveis de instância (atributos) de uma classe, e executam uma função sem dependência do conteúdo do objeto. Digamos que quando um método não altera o estado do objeto, ele pode ser transformado para um método estático. Exemplo: Classe Math.

A classe Math nos proporciona uma série de operações e constantes matemáticas que são facilmente acessadas estaticamente, ou seja, não precisamos instanciar um objeto para podermos usar seus métodos.

Exemplo: Crie um programa que calcule e exiba a raiz quadrada de 9.

System.out.println("A raiz quadrada de 9 é " + Math.sqrt(9));

Repare também que desde o início do curso estávamos usando um método estático: o System.out.println(), pois não é necessário que um objeto System seja instanciado.

### 3.3 – Classes e métodos abstratos

#### 3.3.1 Classes abstratas

Lembra da classe funcionário?

Se pensarmos logicamente não existe um cargo Funcionário. A classe funcionário só tem serventia para agregar as características em comum entre os cargos citados e possibilitar o uso de polimorfismo na sua manipulação. Tendo isto em vista, como poderemos evitar com que haja um funcionário sem cargo (ou do cargo Funcionário)? As classes abstratas estão aí para isto.

As classes abstratas são classes não instanciáveis. Somente as suas subclasses poderão ser instanciadas (isto se a subclasse não for abstrata também). Para evitarmos que a classe funcionário seja instanciada, basta inserir a palavra reservada abstract em sua declaração.

1. /\*\*
2. \* Classe Funcionário
3. \* @author pearson
4. \*/
5. **public** **abstract** **class** Employee {
6. **protected** String name;    // Nome do funcionário
7. **protected** Double salary;  // Salário do funcionário
9. // Construtor, encapsulamento
10. **public** Employee(String name, Double salary) {
11. **this**.name = name;
12. **this**.salary = salary;
13. }
15. // Sobrecarga do construtor
16. **public** Employee(Double salary) {
17. **this**.salary = salary;
18. }
20. // Retorna a bonificação salarial de acordo com o cargo.
21. **public** Double getBonus() {
22. **return** 0.0; // Zero, se não for um dos três cargos acima
23. }
24. }

Listagem 3.15

Veja que nada além da inserção do modificador abstract foi alterado.

Tente instanciar uma classe Employee.

Employee employee = **new** Employee();

Um erro será disparado e o código não executará.

#### 3.3.2 Métodos abstratos

Um método abstrato é similar à classe abstrata, ele não pode ser executado, a não ser que for sobreposto em uma subclasse. O método abstrato poderá estar em uma classe não abstrata, porém poderá ser executado apenas se a subclasse o implementar.

Veja no exemplo funcionário. Agora que a classe Employee é abstrata, o método getBonus() nunca será executado por si só, ou seja, apenas as suas subclasses poderão executá-lo e o valor de retorno 0,0 nunca será obtido. Então nada melhor do que deixá-lo abstrato. Para isso use o modificador de acesso na declaração do método e remova todo o seu conteúdo da seguinte maneira:

**public** **abstract** Double getBonus();

Desta forma não conseguiríamos executar o método getBonus() com um objeto Employee se o mesmo não fosse abstrato.

### 3.4 – Resumo

Nesta aula, aprendemos o que é polimorfismo e alguns dos recursos que ele nos oferece como sobreposição e sobrecarga. Os métodos estáticos são recursos que executam uma função independente de um objeto instanciado. Também foram abordados Classes e Métodos abstratos, que são formas de manter o objeto polimórfico de forma que um mesmo método pode gerar comportamentos diferentes para as classes que o sobreescrevem.

### 3.5 – Exercícios

|  |  |
| --- | --- |
| 1. | O que é polimorfismo em programação? |
| 2. | O que é sobreposição e sobrecarga? |
| 3. | O que são métodos estáticos? |
| 4. | O que são classes abstratas? |
| 5. | O que é um método abstrato? |
| 6. | Qual a relação entre Polimorfismo e Sobrecarga? |
| 7. | O que acontece se instanciarmos um objeto de uma classe abstrata? |
| 8. | O que acontece se tentarmos usar um método abstrato? |

### 3.6 – TDP

Os desenvolvedores estão gostando da ideia de utilizar POO. Para convencê-los de vez e deixá-los de queixo caído, mostre o poder do Polimorfismo.

#### Tarefa

Abstraia mais e tente identificar possíveis usos de polimorfismo e classes estáticas. Implemente.