

Herança

Apresentação

É chegada a hora de tratarmos de mais um pilar importante do POO, que é a Herança. Até o momento, já foram abordados a abstração e o encapsulamento. Você observou a importância de se fazer uma abstração correta (captar estritamente o necessário para resolver dado problema) e ocultar os detalhes de implementação de um objeto (tanto os dados quanto à complexidade das ações). O relacionamento "TER" foi discutido e você percebeu que um objeto pode ter outros objetos e fazer várias operações por meio de seus métodos. Afinal, um objeto TEM características e ações.



Fonte: <https://goo.gl/VqCfDT>

Você interagiu com objetos do Java, como String e ArrayList, e pode perceber que ficam visíveis apenas métodos desses objetos e que os serviços são bem coesos e objetivos. Objetos desenvolvidos com essas características podem ser mais facilmente reutilizados em diversas soluções.

Nesta aula, você conhecerá os conceitos acerca de mais um pilar, a Herança. Serão discutidas as armadilhas que podem aparecer na abstração deste conceito. Conhecerá a diferença entre generalizar e especializar uma classe, bem como os termos que fazem parte deste conceito. O relacionamento SER será discutido. Por fim, perceberá a importância deste pilar para o reuso no paradigma orientado a objetos.

Conteúdo

O Conceito

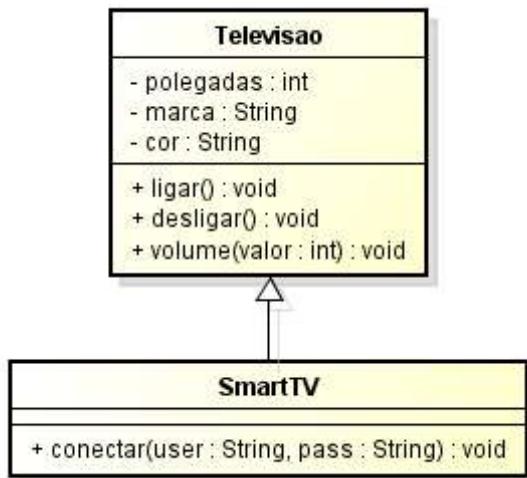
Qual é a primeira coisa que vem à sua cabeça quando você escuta a palavra herança? A pergunta foi feita em uma pesquisa rápida com dez alunos da Universidade que cursam outros cursos que não são na área de Tecnologia e, após análise, em linhas gerais, a resposta que prevaleceu foi a seguinte: "algo que se recebe de alguém". Concorda?

Segundo o Dicionário Aurélio (2017), herança é aquilo que se herda ou que se deve herdar. Mas como este conceito pode ser aplicado na Programação Orientada a Objetos?

Imagine uma classe que já possui características e ações bem definidas e com complexidade considerável e que outra classe possa herdar todas as implementações já realizadas e ainda poder acrescentar suas peculiaridades. Pense na seguinte abstração: Uma classe Televisão com suas características (polegadas, cor e marca) e suas ações (ligar, desligar, aumentar e abaixar o volume). Imagine que desejamos implementar uma Smart TV. Concorda que este tipo de televisão tem tudo que a abstração citada de Televisão tem? Além disso, ela tem algumas ações que são especificamente delas, como conectar à Internet, por exemplo. Percebeu a importância? Imagine reescrever toda a classe Smart TV, ao invés de herdar o que já existia como abstração da classe Televisão e apenas acrescentar a ação de conectar à internet. Seria um retrabalho, sim? Esse não é o foco, mas sim o **reuso**.

A Figura 1 representa o projeto da abstração da relação da classe Televisão e Smart TV, utilizando diagrama de classe da UML.

Figura 1 – Projeto da relação de herança entre Televisão e Smart TV.



A classe que serve de base para que outra classe herde as características e ações pode ser chamada de: **classe base, superclasse ou classe pai**.

A classe que herda, que recebe a implementação da classe base, pode ser chamada de: **classe derivada, subclasse ou classe filha**.

Sobre os diagramas da UML, neste momento, basta você saber que as entidades são representadas pelas caixas e que o triângulo vazado significa uma relação de Herança.

Abstraindo a Herança

Para abstrair a Herança em um determinado contexto, você deve pensar em uma generalização ou em uma especialização.

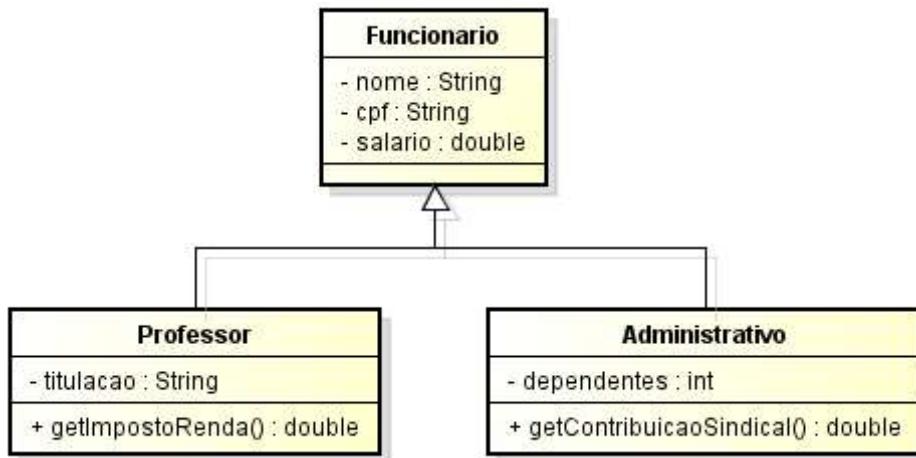
No caso da **generalização**, você generaliza entidades a partir de características e ações comuns. Por exemplo, em uma abstração específica, tem-se que em uma universidade trabalham dois tipos de funcionários, que são os **administrativos** e os **professores**. Estes possuem nome, cpf, titulação, salário como atributos e possui um método de cálculo do imposto de renda. Já os **administrativos** possuem nome, cpf, dependentes, salário como atributos e possuem um método de contribuição sindical baseado no salário. A figura 2 é uma representação de um projeto das entidades **Professor** e **Administrativo**.

Figura 2 - Representação das entidades Professor e Administrativo.

Professor	Administrativo
- nome : String - cpf : String - salario : double - titulacao : String	- nome : String - cpf : String - salario : double - dependentes : int
+ getImpostoRenda() : double	+ getContribuicaoSindical() : double

A partir desta abstração, você conseguiu visualizar o que as entidades têm em comum? Percebeu que todos são funcionários? Percebeu que ambas entidades possuem nome, cpf e salário como atributos? Pois bem, generalizar é buscar as características e ações comuns e atribuir esta responsabilidade a uma classe base, superclasse ou classe pai (escolha o termo que preferir) e fazer com que as entidades **Professor** e **Administrativo** herdem o que é comum **para fins de reuso**. Caso não existisse essa possibilidade, as implementações se replicariam em ambas as entidades. A Figura 3 é uma representação das entidades **Professor** e **Administrativo** em um contexto de herança.

Figura 3 – Representação das entidades professor e administrativo com herança.



Neste contexto, a classe **Funcionário** é uma generalização do cenário que existe entre **Professor** e **Administrativo**.

No caso da **especialização**, se já existir uma abstração que sirva como classe base, você pode apenas especializar a implementação. Foi justamente o que aconteceu no exemplo da Televisão e da Smart TV. Uma televisão é mais generalista frente à especialização da Smart TV.

Logo, quando uma entidade já existir e uma subclasse estender esta entidade você está especializando a entidade base e acrescentando novas características.

Ao pensar em generalização e especialização em um determinado contexto, você está em conformidade com a herança, que é um dos principais pilares da POO.

Relacionamento SER

Conforme abordado anteriormente, na aula 4 da Unidade I, dois verbos regem a organização do pensamento no paradigma orientado a objetos. O verbo "TER" e o "SER". O verbo "TER" você observou que é a possibilidade de **um objeto ter outros objetos** em forma de atributos.

No caso da Herança, o verbo que rege esta relação é o verbo "SER". Antes de criar uma generalização ou uma especialização, você deve ser perguntar se a classe derivada, subclasse ou classe filha **É UM** tipo da classe base, superclasse ou classe pai. Uma Smart TV **é uma** Televisão. Um Professor **é um** Funcionário.

Não é pelo fato de uma abstração ter dados em comum que você deve partir para criação de uma relação de Herança. Por exemplo, uma escola possui nome e endereço. Já um aluno possui nome, endereço e matrícula. Baseado no exposto, equivocadamente, você poderia generalizar uma das entidades e criar uma relação de Herança entre aluno e escola.

Por exemplo, você poderia ser tentado a fazer aluno herdar de escola ou ainda tentar criar uma entidade genérica entre as duas. Qualquer uma das estratégias estaria errada, pois não passam no teste **é um**. Um **Aluno** não pode herdar uma **Escola**, pois **um aluno não é uma escola**. E, ainda, tente definir uma classe genérica entre escola e aluno neste contexto, qual seria o nome? Percebe a dificuldade? Sempre que sentir dificuldade em dar nome para uma generalização, provavelmente estará equivocado. Sempre verifique, ao implementar a herança, se as classes derivadas passam no teste **É UM** em relação às classes base.

Para Refletir ⚡

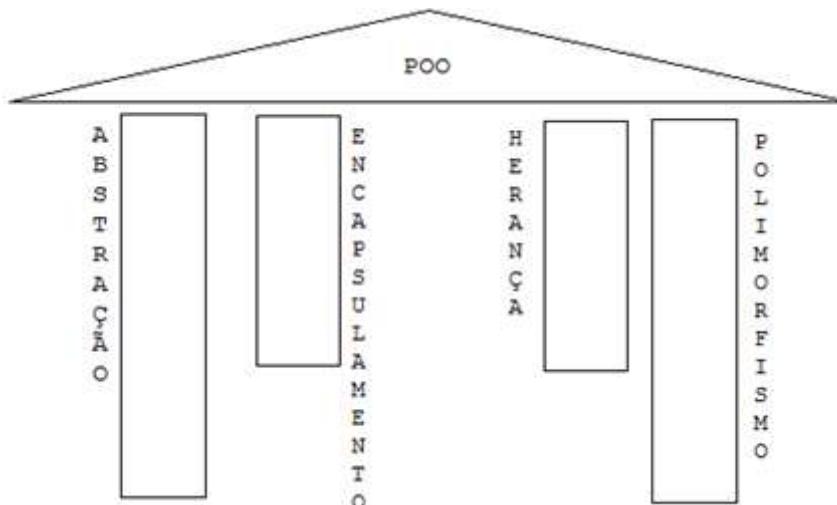
Com todas as informações sobre Herança apresentadas até o momento, você consegue pensar em uma definição para Herança na POO?

Uma definição minimalista seria... Herança é um dos pilares da orientação a objetos que provê a possibilidade de reaproveitamento (reuso) de uma entidade previamente criada.

Herança no Java

Até agora, você teve contato com a parte conceitual do pilar da Herança. Ao programar, esses conceitos devem fazer parte de suas decisões, para que seus programas estejam coerentes e em conformidade com todos os pilares. Esqueceu quais são elas? A Figura 4 o ajudará a lembrar.

Figura 4 - Representação dos pilares da orientação a objetos.



Agora que relembrou quais são os pilares e conheceu os conceitos da Herança, você precisa aprender como representar a herança utilizando a linguagem Java. Vamos lá!

A herança no Java é materializada pelo uso do extends. Uma classe derivada cria um vínculo de herança após estender uma classe base. O código apresentado na Tabela 1 é uma implementação da Figura 3, que representa a relação de herança entre **Professor** e **Administrativo** de uma universidade.

Tabela 1 – Abstração da relação de herança entre as classes professor e administrativo.

Programa em Java:Programa.java

```
//programa atendeu aos pilares da abstração, encapsulamento e herança.
public class Programa{
    public static void main(String args[]){
        Professor prof = new Professor();
        prof.setNome("João da Silva");//método acessor herdado
        prof.setCpf("165.812.493-60");//método acessor herdado
        prof.setSalario(8000);          //método acessor herdado
        prof.setTitulacao("MESTRE");   //método acessor de professor
        double ir = prof.getImpostoRenda();
        System.out.println("Sr(a) "+prof.getNome()+" irá pagar de imposto o valor:");
    }
}
```

Programa em Java:Programa.java

```
R$ "+prof.getImpostoRenda());
    }
}
class Funcionario {
    private String nome;
    private String cpf;
    private double salario;
    public void setNome(String nome){
        this.nome = nome;
    }
    public String getNome(){
        return nome;
    }
    public void setCpf(String cpf){
        this.cpf = cpf;
    }
    public String getCpf(){
        return cpf;
    }
    public void setSalario(double salario){
        this.salario = salario;
    }
    public double getSalario(){
        return salario;
    }
}
class Professor extends Funcionario{//vinculo de herança
    private String titulacao;
    public void setTitulacao(String titulacao){
        this.titulacao = titulacao;
    }
    public String getTitulacao(){
        return titulacao;
    }
    public double getImpostoRenda(){
        return getSalario() * 0.15; //taxa fixa de 15% do salário
    }
}
class Administrativo extends Funcionario{//vínculo de herança
    private String dependentes;
    public void setDependentes(String dependentes){
        this.dependentes = dependentes;
    }
    public String getDependentes(){
        return dependentes;
    }
    public double getContribuicaoSindical(){
        return getSalario() * 0.01; //taxa fixa de 1% do salario
    }
}
```

A palavra extends, conforme o exemplo de codificação da Tabela 1, criou a relação de Herança entre as classes **Professor** e **Administrativo** com **Funcionário**.

Dessa forma, os métodos acessores foram todos herdados pelas classes derivadas.

Perceba que, por meio da referência do tipo Professor chamada prof foi possível acessar todos os métodos como se pertencessem a Professor. Veja que a Herança proporcionou o reuso de uma entidade que foi **generalizada** chamada Funcionario. Após esta generalização, qualquer entidade que surgir no escopo do sistema, se for um tipo Funcionario, basta herdar e definir suas especialidades. Suponha que seja necessário que o sistema também administre os estagiários. Basta o programador criar uma entidade Estagiario herdar de Funcionario e na sequência definir seus atributos e ações específicas.

Para Refletir ⚡

Como a Herança impacta no desenvolvimento de aplicações em um ambiente de fábrica de software?

As fábricas de software, geralmente, faturam de acordo com a quantidade de funcionalidades entregues de um determinado sistema para seus usuários. Baseado nisto, quanto mais rápido e facilitado for o desenvolvimento, maior será a produtividade. A reutilização aparece como ponto forte neste contexto. Lembra-se da definição da Herança? Foco no reuso.

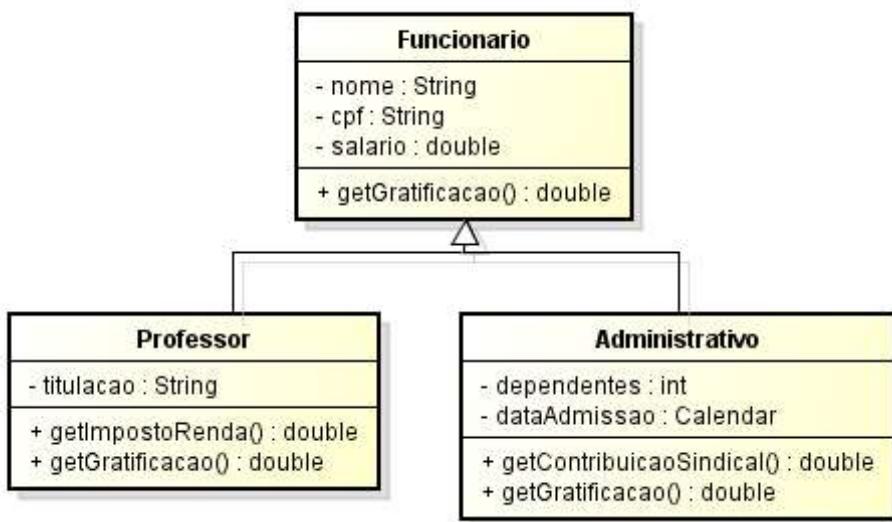
Sobrescrita

Atrelado ao conceito de Herança, existe a **Sobrescrita**. A sobrescrita é a especialização de uma ação específica, onde se mantém a mesma assinatura (nome do método e argumentos) do método, mas com comportamento diferente. Este importante recurso está presente na abstração de vários objetos da vida real. Todos os animais comem, mas cada animal come de uma maneira peculiar. Todos os equipamentos eletrônicos ligam, mas cada um faz isso a sua maneira. Perceba que as ações citadas são comuns em uma relação de herança, mas são especializadas em cada subclasse.

Pois bem, voltando ao exemplo da universidade que possui como funcionários, professores e administrativos, abstraia o seguinte: tanto professores quanto os funcionários recebem uma gratificação, os professores de acordo com a titulação, os administrativos de acordo com o tempo de serviço. Percebe que ambos possuem uma

ação de cálculo de gratificação? Baseado no pilar da Herança e pensando na generalização, você deve criar um método genérico em funcionário chamado `getGratificacao()`. Entretanto, você deve lembrar que este cálculo de gratificação é feito de forma diferente para cada subclasse. É exatamente aí que entra a sobrescrita do método de gratificação herdado da classe base `Funcionario`, pois cada um (`Professor` e `Administrativo`) possuem regras de cálculo diferentes. A classe filha herdará todas as características e ações da classe pai, mas deverá sobreescrivê-las para que este funcione de forma peculiar nas classes filhas. Observe a figura 5 para visualizar esta abstração.

Figura 5 – Representação de sobreescrita.



Observe que o método `getGratificacao` aparecerá em todas as entidades, já que conceitualmente todos os funcionários têm a gratificação. Contudo, ela é calculada de forma distinta nas entidades especializadas; consequentemente, este método, que é responsável por este cálculo, deve ser sobreescrito em cada uma das entidades filhas e cada uma terá uma lógica de implementação distinta.

Vários termos e conceitos foram contemplados pelo material até o momento. A Figura 6, a seguir, traz um esquema dos termos utilizados na herança, para você ter um modelo sobre o tema.

Figura 6 – Resumo dos termos que envolvem o conceito da Herança.

Para Refletir ⚡

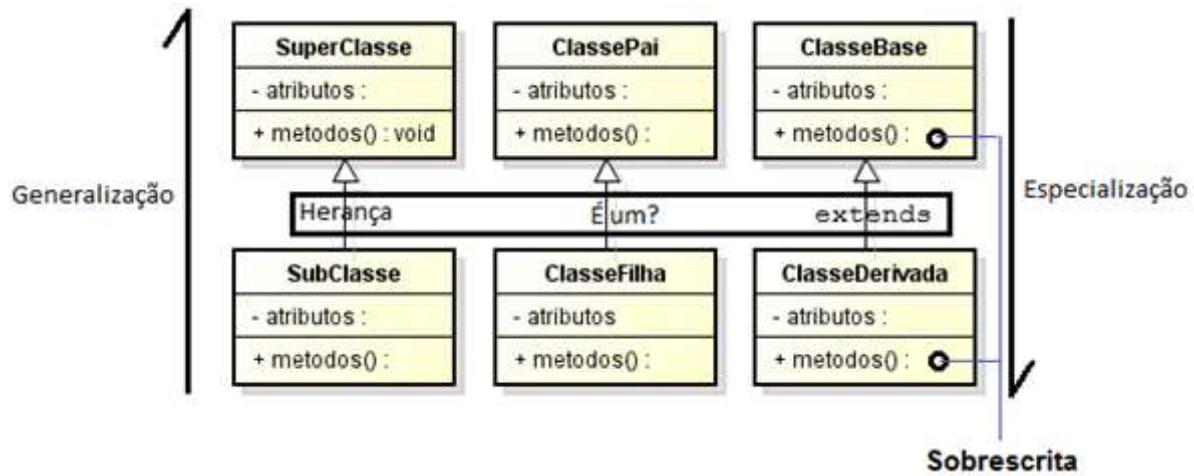
Já imaginou uma classe herdar duas classes? Por exemplo, um hovercraft é um veículo que consegue andar no solo e na água... Concorda que ele tem características de veículo terrestre e aquático? Em uma abstração, é como se ele herdasse das duas fontes de informação. É possível? Quais são as desvantagens? Existe vantagem?

Em Java, não existe herança múltipla. Este recurso está presente na linguagem C++, entretanto, os engenheiros da SUN resolveram retirá-lo do Java. Já imaginou uma classe com métodos com mesmo nome, mas implementações diferentes sendo Herdados, qual seria levado em consideração?

Finalizando... C

Herança é um conceito muito importante no contexto do paradigma orientado a objetos. Uma boa abstração de herança provê excelente reutilização e facilidade de manutenção por meio da sobrescrita. Perceba que esta é uma disciplina prática, entretanto, a aplicação dos conceitos é essencial para garantia da qualidade do seu código. Lembre-se: não adianta você conhecer toda a sintaxe Java se o seu código não estiver em conformidade com os pilares da orientação a objetivos. "A arma deve ser utilizada da forma correta". Espero que você tenha entendido com certo conforto o conceito da herança e o da sobrescrita, pois estes conceitos são importantes para o entendimento do último pilar: o Polimorfismo.

Agora você precisa praticar. Faça todos os exercícios que foram deixados para trás, poste suas dúvidas, assista aos vídeos, discuta as lacunas com seus colegas e professor. Você tem de se esforçar para não deixar pendências para a próxima aula, que reforçará alguns conceitos estudados nesta aula. Até a próxima!



Vamos colocar em prática? Imagine que a gratificação do professor seja calculada da seguinte forma: 10% do salário para o especialista, 12% do salário para o mestre, e 15% do salário para o doutor. Já, do administrativo, é R\$ 0,30 por dia de trabalho. Assista ao [vídeo 1](#) para acompanhar esta implementação.

Vídeo 1

Prática Profissional - Programação orientada à objeto - Unidad...



Na Prática

"Prezado(a) estudante,

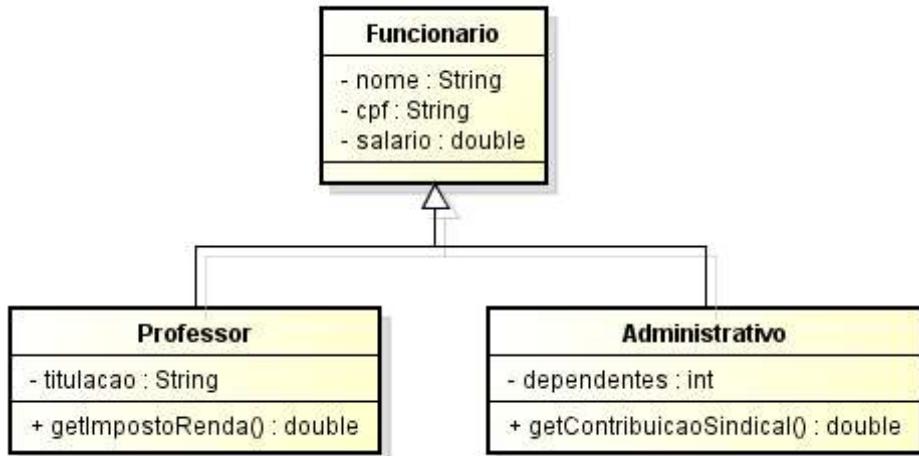
Esta seção é composta por atividades que objetivam consolidar a sua aprendizagem quanto aos conteúdos estudados e discutidos. **Caso alguma dessas atividades seja avaliativa, seu (sua) professor (a) indicará no Plano de Ensino e lhe orientará quanto aos critérios e formas de apresentação e de envio.**"

Bom Trabalho!

Atividade 01



Em uma universidade, há dois tipos de funcionários: professor e administrativo. Conforme a figura a seguir.



Crie um programa Java que implemente o projeto da figura acima. Crie uma classe chamada **Funcionario** com os atributos (nome, cpf e salário). Crie uma classe **Professor**, que é um Funcionário e, além dos dados de funcionário, tem a titulação e o cálculo do seu imposto de renda por intermédio do método `getImpostoRenda()` que é 23 % do salário. Crie também a classe **Administrativo**, que também é Funcionário e tem um dado que representa a quantidade de dependentes. Não se esqueça de criar o método `getContribuicaoSindical()`, que retornará 1% do salário.

Seu programa deve perguntar qual tipo de Funcionário deseja informar P - Professor e A - Administrativo. Quando o usuário escolher a opção receba os dados do respectivo objeto e apresente todas as suas informações.

Atividade 02



Um matemático trabalha com dois tipos de figuras: os triângulos retângulos e os retângulos. Crie um programa Java (classes) que represente a modelagem para abstrair este problema. Ambos têm dois dados comuns: Base e Altura. Cada uma tem um cálculo de forma distinta de área. O triângulo retângulo é base X altura / 2, já do retângulo é apenas base X altura. Um cálculo diferente somente para o triângulo retângulo deve ser realizado, que é a hipotenusa. Crie um programa de teste, que instancie objetos, atribua valores e execute para apresentação.

Saiba Mais

Para ampliar seu conhecimento a respeito desse assunto, veja abaixo a(s) sugestão(ões) do professor:

- Leia o artigo sobre diagrama de classes e conceitos correlatos, inclusive a herança, publicado no [Portal IBM](#).
- Leia a postagem "[Conheça a nova API de datas do Java 8](#)".
- Leia também o Capítulo 5 do livro *Core Java – Fundamentos*, por Cay S. Horstman e Gary Cornell.

Referências

- BOOCH, G. **Object-Oriented Design with Applications**, Benjamin-Cummings, 1991.
- BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML – guia do usuário**. 2. ed. Rio de Janeiro: Campus. 2006.
- CARDELLI, L.; WEGNER, P. **On Understanding Types, Data Abstraction, and Polymorphism**. ACM Computing Surveys (CSUR). vol. 17, pp. 471-523. 1985.
- DEITEL H. M.; DEITEL, P. J. **Java, como programar**. 6. ed. Porto Alegre: Bookman. 2006.
- DICIONÁRIO AURELIO. 2017. Disponível em:
<https://contas.tcu.gov.br/dicionario/home.asp> Acesso em: 19 mar. 2017.
- ERICH, G. et al. **Padrões de projeto**: soluções reutilizáveis de software rientado a objetos. Porto Alegre: Bookman. 2000.
- HORSTMANN, C. S.; CORNELL, G. **Corejava 2 – Volume I – Fundamentals**. São Paulo: Makron Books. 2010.
- NEWRELIC. **The Most Popular Programming Languages of 2016**. 2016. Disponível em:
<https://blog.newrelic.com/2016/08/18/popular-programming-languages-2016-go>. Acesso em: 9 Mar 2017.
- NIEMEYER, P.; KNUDSEN, J. **Aprendendo Java**. Rio de Janeiro: Campus. 2000.
- ORACLE. **Java Licensing Logo**. 2017. Disponível em:
<http://www.oracle.com/us/technologies/java/java-licensing-logo-guidelines-1908204.pdf>. Acesso em: 13 mar. 2017.
- SIERRA, K.; BATES, B. **Certificação Sun para Programador JAVA 5 Guia de Estudo**. Rio de Janeiro: Alta Books, 2006.
- SILBERSCHATZ, A; GALVIN, P. B.; GAGNE, G. **Sistemas operacionais com Java**. 6. ed. Rio de Janeiro. Editora Campus, 2004.

- STUCKEY, P. J.; SULZMANN, M. **A theory of overloading**. International Conference on Functional Programming. Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. Pittsburgh, PA, USA, 2002. pp. 167-178.
- WIKIPÉDIA. Desenvolvido pela Wikimedia Foundation. Conteúdo sobre o **Ambiente de Desenvolvimento Integrado**. 2017. Disponível em: <https://pt.wikipedia.org/wiki/Ambiente_de_desenvolvimento_integrado>. Acesso em: 5 mar. 2017.