

# Polimorfismo

# Apresentação



Fonte: <https://goo.gl/s8N3FV>

*"It'ssssss tiiiiimeeee..." É isso mesmo... empolgação total. Este conteúdo é um mito, uma lenda... Com todo respeito aos demais pilares, mas este é "o pilar".*

*Brincadeiras a parte, nesta aula você conhecerá o **polimorfismo**. Este paradigma completa com estilo os pilares da orientação a objetos. Este pilar permitirá abstrações com grande capacidade de manutenção e reuso. Diversos padrões de projetos são baseados neste pilar. Bons programadores devem ser mestres em construir abstrações polimórficas.*

*Isso só aumenta a sua responsabilidade em ler com bastante atenção o material, acompanhar os vídeos e, principalmente, fazer TODOS os exercícios. E não se esqueça de tirar suas dúvidas com seu professor. Está pronto? "Aree youuu ready?... Let's go...".*

*Só uma pergunta... Fez sua parte nos conteúdos anteriores? Não?!?! Então, economize esforço e finalize todas as suas demandas, isso ajudará você no conteúdo que está por vir. Se sua resposta for sim, não se esqueça do café para se inspirar, mas use uma xícara "top" porque esse conteúdo merece.*

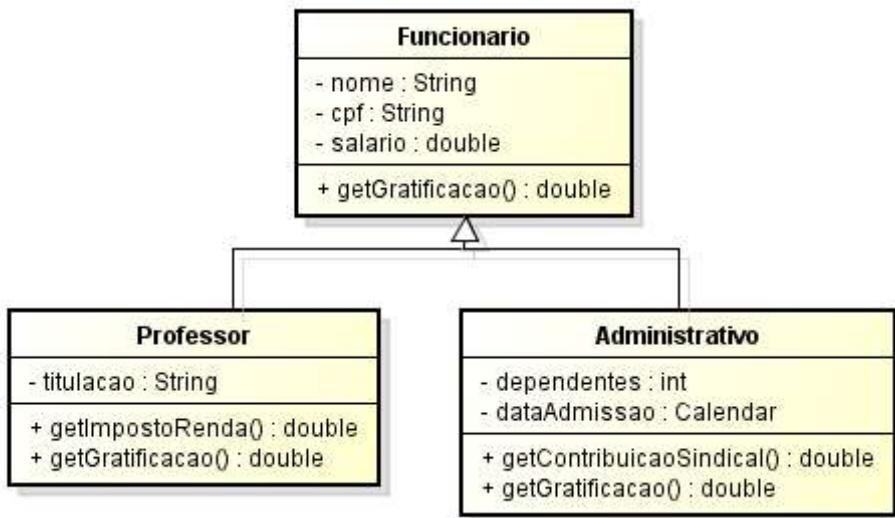
#partiu.

# Conteúdo

## O Conceito

A palavra polimorfismo vem do grego *poli*, que significa várias, e *morphos*, que significa formas. Logo, várias formas. Já um imaginou um objeto que tenha o comportamento de vários objetos a partir de uma forma genérica em dado momento? É para isso que o polimorfismo serve. Difícil de abstrair, não é? Calma... Para te explicar será utilizada a mesma abstração da aula passada entre funcionários de uma universidade, conforme a Figura 1:

**Figura 1 – Abstração de funcionários de uma universidade.**



Agora, imagine o seguinte: `Funcionario f = new Professor()` . Uma referência de **Funcionario** acabou de receber uma instância de um objeto **Professor**. Ou seja, *uma referência genérica recebeu uma instância especializada*. Esta atribuição só foi possível por conta da relação de herança que existe entre as duas entidades (viu como as aulas passadas são importantes?). A superclasse sempre poderá receber uma instância da subclasse, mas nunca o contrário. Se você tem alguma dúvida sobre herança, leia novamente o conteúdo da Aula 5. É muito importante para você avançar no estudo dessa aula.

Logo, Professor p = new Funcionario() não é possível.

Então, voltamos ao caso do funcionário recebendo uma instância de professor, Funcionario f = new Professor() , no caso deste trecho de código, você consegue imaginar o que é possível fazer com a referência f baseado na abstração da Figura 1? Melhorando a pergunta, quais métodos são possíveis acessar com a referência 'f' ? Neste contexto, a referência f só consegue acessar os métodos que existem em Funcionario. Ou seja, não é possível acessar, por exemplo, o método getTitulacao() . Você deve estar se perguntando: "Tá, mas qual é o benefício deste tipo de atribuição, já que só pode acessar os membros de funcionário?" Já imaginou o que acontece com os métodos comuns às duas entidades, ou seja, os métodos que foram sobrescritos ou, se preferir, especializados? O que está sendo perguntado é: o que aconteceria se fosse acionado o método f.getGratificacao() ?

Isso é polimorfismo. Será acionado o método getGratificacao() de Professor e não de Funcionario. Isso é fantástico!

A referência do objeto Funcionario conseguiu acionar um método do objeto Professor que está em uma posição mais especializada na árvore de herança.

O benefício deste pilar é que poderia ser atribuída à referência f de Funcionario uma instância de Administrativo e após acionar o método f.getGratificacao() seria acionado o método do objeto instanciado, ou seja, administrativo. Você deve ter compreendido que uma referência do tipo genérico conseguiu "se passar" por outros objetos, por isso se chama polimorfismo.

Acompanhe a Tabela 1 a seguir e observe um exemplo de aplicação deste conceito a partir de uma abstração que você já conhece.

### Tabela 1 – Abstração exemplo de polimorfismo.

#### Programa em Java:Programa.java

```
import java.util.Calendar;
public class Programa{
    public static void main(String args[]){

        //objeto professor
        Professor prof = new Professor();
        prof.setNome("João da Silva");
        prof.setCpf("165.812.493-60");
        prof.setSalario(8000);
        prof.setTitulacao("MESTRE");

        //objeto administrativo
        Administrativo adm = new Administrativo();
        // admitido em 1 de março de 2000
```

```
Calendar dtAdmissao = Calendar.getInstance();
dtAdmissao.set(Calendar.YEAR, 2000); //ano 2000
dtAdmissao.set(Calendar.MONTH,Calendar.MARCH);//mês de março
dtAdmissao.set(Calendar.DAY_OF_MONTH, 1);// dia 1
adm.setDataAdmissao(dtAdmissao);
adm.setNome("José da Sivva");
adm.setCpf("877.388.152-05");
adm.setSalario(1200);
adm.setDependentes(2);

//vamos falar de polimorfismo
Funcionario f;
f = prof; //atribuindo professor a funcionario
System.out.println("Funcionário(a) "+f.getNome()+" tem direito de: R$ "+f.getGratificacao());
f = adm; //atribuindo administrativo a funcionario
System.out.println("Funcionário(a) "+f.getNome()+" tem direito de: R$ "+f.getGratificacao());
}

}

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;
}
public double getGratificacao(){
    return 0;//Já que depende do tipo fr funcionários para receber é 0
}
}

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
}

public double getGratificacao(){
    //testando String
    if(getTitulacao().equalsIgnoreCase("ESPECIALISTA"))
        return getSalario() * 0.1;//10% para especialista
    else if(getTitulacao().equalsIgnoreCase("MESTRE"))
        return getSalario() * 0.12;//12% para mestre
    else if(getTitulacao().equalsIgnoreCase("DOUTOR"))
        return getSalario() * 0.15;//15% para doutor
    else
        return 0;//caso não seja identificado - 0
}
}

class Administrativo extends Funcionario{//vínculo de herança
private int dependentes;
private Calendar dataAdmissao;
public void setDependentes(int dependentes){
    this.dependentes = dependentes;
}
public int getDependentes(){
    return dependentes;
}
public double getContribuicaoSindical(){
    return getSalario() * 0.01; //taxa fixa de 1% do salario
}
public double getGratificacao(){
    return getDiasTrabalhados() * .30;//30 centavos por cada dia trabalhado
}
public Calendar getDataAdmissao() {
    return dataAdmissao;
}
public void setDataAdmissao(Calendar dataAdmissao) {
    this.dataAdmissao = dataAdmissao;
}
//retorna a quantidade de dias trabalhado
public int getDiasTrabalhados(){
    Calendar dtAtual = Calendar.getInstance();
    int MILLIS_IN_DAY = 86400000;//milisegundos de um dia
    return (int) ((dtAtual.getTimeInMillis() -
getDataAdmissao().getTimeInMillis()) / MILLIS_IN_DAY);
}
```

## Para Refletir ⚡

Qual foi o objetivo de criar o método `getDiasTrabalhados`? Qual foi o grande benefício alcançado?

Perceba a importância de criar métodos coesos e reutilizáveis. No programa apresentado na Tabela 1, o método `getDiasTrabalhados` poderá ser utilizado em outras operações, como o cálculo de aposentadoria do funcionário.

Uma definição plausível e eficiente sobre o polimorfismo é: "uma referência genérica que é capaz de realizar uma chamada especializada em tempo de execução (*ligação tardia*)". Somente no momento que a aplicação executa que a chamada "desce" na árvore de herança para realizar a chamada mais especializada do método sobrescrito. Dentro deste contexto, o Polimorfismo é completamente dependente da herança e da sobrescrita.

## As Divergências Conceituais

Alguns autores divergem quanto ao polimorfismo. A divergência aparece se a sobrescrita e a sobrecarga fazem parte do conceito de polimorfismo ou não.

Vamos a algumas considerações. Erich et al. (2000) definem polimorfismo como a capacidade de substituir objetos com interfaces coincidentes por um outro objeto em tempo de execução. Este conceito também é conhecido por ligação dinâmica (*dynamic binding*) ou se preferir ligação tardia (*late binding*), este último termo é utilizado por (HORSTMANN; CORNELL, 2010).

Já, para Booch, Rumbaugh e Jacobson (2006), em uma estrutura de herança, poderá haver muitos métodos para a mesma operação (sobrescrita) e o polimorfismo seleciona qual método existente na hierarquia é usado em tempo de execução.

Horstmann e Cornell (2010) afirmam que o âmago do polimorfismo é a ligação tardia. O mecanismo de chamada de método tradicional é chamado de ligação estática (*static binding* ou *early binding*), pois o método a ser executado é totalmente determinado em tempo de compilação. Para eles, a ligação estática depende apenas do tipo de variável; já, a ligação dinâmica depende do tipo do objeto real em tempo de execução. Os autores concluem que "o polimorfismo em uma hierarquia de heranças é algumas vezes chamado de **polimorfismo verdadeiro**. A ideia é distingui-lo do tipo mais limitado de sobrecarga, que não é resolvido dinamicamente, mas estaticamente em tempo de compilação".

Deitel e Deitel (2006) defendem que o polimorfismo provê a extensibilidade do software e associam o polimorfismo à ligação dinâmica de método. Sierra e Bates (2006) defendem que o polimorfismo se aplica somente à sobrescrita e não à sobrecarga. Essa referência é da certificação Java.

Pelo exposto, pode-se concluir que o polimorfismo só existe quando há sobreposição (sobrescrita ou *override*) de métodos. No entanto, a divergência aparece ao encontrar autores que definem dois tipos de polimorfismo: o estático (*ad-hoc* ou ocasional) e o dinâmico (verdadeiro ou universal). Nestas definições, sobrecarga também é considerado polimorfismo.

Niemeyer e Knudsen (2000) afirmam que todos os métodos em Java podem ser sobrecarregados (*overload*); esse é um aspecto do princípio de polimorfismo da programação orientada a objetos. Os autores definem a sobrecarga como "polimorfismo ocasional".

Já, Stuckey e Sulzmann (2002) definem sobrecarga como "polimorfismo *ad-hoc*". Finalmente, Cardelli e Wegner (1985) fazem uma profunda análise sobre os tipos de polimorfismo na qual a sobrecarga é considerada "polimorfismo *ad-hoc*" e a sobreposição é considerada "polimorfismo universal" ou "**verdadeiro polimorfismo**".

Particularmente, é estranho definir duas coisas e chamar uma delas de "verdadeiro". Enfraquece a outra... não é verdade?

Baseado no que foi apresentado, conclui-se que o objetivo principal do polimorfismo é facilitar a manutenção evolutiva do software e que só a sobreposição/sobrescrita é capaz de proporcionar tal característica. No entanto, é possível encontrar em bibliografias renomadas o conceito de polimorfismo *ad-hoc*, estático ou ocasional, no qual se enquadra a sobrecarga.

Nesta disciplina, não se considera a sobrecarga como um conceito relacionado ao polimorfismo, **somente a sobrescrita**.

Logo, uma definição que atenda à definição do polimorfismo sugerida seria a seguinte: **polimorfismo é uma referência genérica capaz de realizar uma chamada especializada (método sobrescrito) em tempo de execução (ligação tardia ou dinâmica)**.

Para iniciar, não confunda abstração com classe abstrata. Abstração é um pilar da orientação a objetos, que diz: ao abstrair (pensar) em uma entidade você deve utilizar somente o necessário, e tão somente o necessário para materializá-la. Classes abstratas servem para representar entidades abstratas. Entendeu? Veja só... Se um amigo diz que possui um animal de estimação, sem falar o tipo, você não conseguirá concluir se ele tem um cachorro, um gato, um cavalo, uma cobra, um peixe, etc. Vamos ao exemplo: na abstração apresentada anteriormente tínhamos uma universidade com dois tipos de funcionários: Professores e Administrativos. Se alguém diz ser funcionário sem dizer o tipo, nunca saberemos se é um professor ou administrativo. Isso acontece porque, nos exemplos citados, tanto animal quanto funcionário são conceitualmente abstratos. Para representar essas situações, temos a opção de definir uma classe abstrata. Nem sempre uma generalização será uma classe abstrata, não tome isso como regra.

Isto é tão evidente, que no exemplo anterior tivemos que colocar o método `getGratificacao()`, retornando zero, por não saber qual é o tipo do funcionário, torcendo que as subclasses "sobrescrevessem" o método para determinar o comportamento correto. As classes abstratas podem ser utilizadas para representar classes como estas, que não há necessidade de instanciar, que representam entidades conceitualmente abstratas, e, além disso, podem obrigar as subclasses a "sobrescreverem" **métodos abstratos** que foram definidos na classe pai abstrata.

### Para Refletir ⚡

Mesmo sem poder instanciar uma classe abstrata, faz sentido sua existência?

A representação conceitual das classes abstratas é importante. Elas não podem ser instanciadas, mas podem ser estendidas. Logo, faz todo sentido definir classes abstratas, mesmo sem poder instanciá-las.

Uma classe abstrata, assim como os métodos, é declarada utilizando a palavra reservada `abstract`, assim como os métodos abstratos.

Vamos para a implementação! A Tabela 2 representa a declaração de uma classe funcionário como `abstract`.

### Tabela 2 – Classe funcionário abstrata.

#### Programa em Java:`Funcionario.java`

## Importante

Assista ao **vídeo 1** para visualizar na prática esse importante recurso.

### Vídeo 1

Prática Profissional - Program...



## Patrocínio ao Polimorfismo

Dois recursos bastante interessantes trazem de forma indireta um patrocínio ao polimorfismo: as classes abstratas e as interfaces.

Conforme abordado anteriormente, o polimorfismo depende da sobrescrita para o seu funcionamento. Concorda que seria interessante uma forma de obrigar as classes derivadas, filhas ou subclasses a reescreverem alguns métodos? Pois bem... Esses recursos (classes abstratas e interfaces) têm um papel interessante neste contexto.

## Classes Abstratas

## Programa em Java:Funcionario.java

```
public abstract 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;  
    }  
    //metodo abstrato. Somente a assinatura não tem corpo, ou seja, {}.  
    abstract public double getGratificacao();  
}
```

Com a implementação desta classe, duas coisas devem ser enfatizadas. A primeira, esta classe **não poderá ser instanciada**. A outra é que todas as subclasses concretas (qualquer classe que não seja abstrata) que herdarem esta classe abstrata serão **obrigadas a definir o comportamento** do método abstrato herdado. Somente classes abstratas podem ter métodos abstratos.

Perceba que, com este conceito presente na abstração da entidade Funcionario, ela se aproxima ainda mais da perspectiva do mundo real.

Concluindo e repetindo, uma classe abstrata impede instanciação equivocada de uma entidade que só existe conceitualmente, fornece às subclasses uma implementação inicial com alguns atributos e métodos e, por fim, ainda pode obrigar as classes filhas completar ou fornecer comportamento para a superclasse, por meio de sobreescrita.

## Interfaces

Uma *Interface* tem um papel análogo ao das classes abstratas, entretanto, ela não fornece implementação inicial alguma. Em uma interface, pode ter apenas métodos e, logicamente, se não fornece implementação, pois todos os métodos de uma interface são abstratos. Sempre que alguém falar em interface para você, lembre-se de um contrato, o qual deve ser todo materializado na classe concreta que a IMPLEMENTA. Implementa? Sim, implementa. Uma interface não é herdada, pois não há nada para herdar, apenas para implementar. A palavra reservada utilizada para interface é o implements. Em uma interface, haverá apenas assinatura de métodos a serem implementados. O exemplo na Tabela 3 mostra a utilização de uma abstração de uma solução fazendo uso de interface.

**Tabela 3 – Exemplo de uso de interface.**

### Programa em Java: Programa.java

```
public class Programa {  
    public static void main(String[] args) {  
        Calculator c = new Soma();  
        double res = c.operacao(10, 20); //Polimorfismo  
        System.out.println("O Resultado da soma é: "+res);  
  
        c = new Subtracao();  
        res = c.operacao(10, 20); //Polimorfismo  
        System.out.println("O Resultado da subtração é: "+res);  
    }  
}  
interface Calculator{  
    //somente métodos com a assinatura  
    public double operacao(double a, double b);  
}  
class Soma implements Calculator {  
    @Override  
    public double operacao(double a, double b) {  
        return a + b;  
    }  
}  
class Subtracao implements Calculator{  
    @Override  
    public double operacao(double a, double b) {  
        return a - b;  
    }  
}
```

Perceba que, com a utilização de interface, cria-se um contrato de implementação que as subclasses devem atender. Com isso, as chamadas polimórficas podem acontecer na utilização deste contexto, conforme apresentado no Programa.java.

O grande benefício é a extensibilidade que é proporcionada ao software. A manutenção fica muito facilitada. Imagine agora que outras operações sejam obrigatórias no programa? Simples. É só definir o comportamento implementando a interface e usar.

Em java, é permitido que uma classe "implemente" (implements) mais de uma interface, ao contrário do que acontece com a "extensão" (extends).

## Finalizando... C

*Esta aula foi muito importante! Englobou conceitos de todos os pilares e lhe dá base para utilização e aplicação de diversos padrões de projetos. O conteúdo evoluiu e você visitou todos os pilares da orientação a objetos. A extensibilidade da sua codificação depende muito da aplicação dos conceitos abordados nesta aula e dos demais pilares. A preocupação com a aplicação de todos os conceitos é fundamental. A qualidade do seu código é diretamente proporcional a sua preocupação e aplicação dos pilares da orientação a objetos. Passar um tempo pensando no melhor projeto não é perda de tempo, pois, com esses pilares aplicados, a manutenabilidade do software desenvolvido por você cresce de forma escalar. Sempre se preocupe com o reuso, já que outros programadores utilizarão os objetos desenvolvidos por você. "O que se faz em vida ecoa pela eternidade..." Seja lembrado por outros colegas como um grande implementador, e não como um gerador de gambiarra.*

*Somente a prática fará com que sua abstração a partir dos pilares aconteça de forma mais natural. Portanto, faça todos os exercícios e siga em frente.*

# 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

^

Um matemático trabalha com dois tipos de figuras os triângulos retângulos e os retângulos. Crie um programa Java (classes) que representem 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 * altura / 2`, já do retângulo é apenas `base * altura`. Um cálculo diferente somente para o triângulo retângulo deve ser realizado que é a hipotenusa.

Crie um método em todas as entidades com a seguinte assinatura:

```
public static double getArea();
```

Um método construtor que receba a base e altura deve ser criado em Figura.

```
public Figura (double base, double altura){ ... }
```

Ao final do programa, faça uma classe main e aplique o conceito do polimorfismo. Faça com que uma referência de figura consiga calcular a área de um triângulo retângulo e na sequência faça para o retângulo.

## Atividade 02



Em uma universidade tem dois tipos de Funcionários: Diretores ou Professores. Um professor tem nome, matrícula, cpf, salário, ano de admissão. Em professor deve ter um método `getImpostoDeRenda`, que deverá retornar o imposto de renda do professor (20% do seu salário). Um diretor tem nome, matrícula, cpf, salário e tempo de casa. Em Diretor também tem o método para obter o imposto de renda. (23% do seu salário).

Pegue tudo que for comum às entidades e coloque em uma classe Funcionário, inclusive o método `getImpostoDeRenda`, para que ele possa ser sobreescrito pelas entidades filhas.

Não é aceito funcionário repetido nesta universidade. Use sobreescrita do `equals` para auxiliar nesta empreitada e para apresentação use o `toString`.

Para criar um funcionário, deve ser passada a matrícula no construtor obrigatoriamente.

Trabalhe com um ÚNICO ArrayList de Funcionário.

Receba os dados de vários funcionários e apresente a média do imposto de renda e o total do imposto de renda.

## Atividade 03



O Datacenter corporativo do Governo está passando por uma adaptação no que tange à administração dos seus sistemas. Basicamente, existem dois tipos de sistema para o governo: Administrativo e Corporativo. O Administrador do Datacenter sobre os sistemas administrativos deseja saber quantos usuários simultâneos utilizam esse tipo de sistema, se deve ficar no ar full time ou não e o nome do sistema. Para os Corporativos, se deve ficar no ar full time ou não, nome do sistema e quantos acessos por minuto ele deve suportar.

Todo sistema tem uma pessoa responsável, que deve ser recebido o nome e telefone. Essa administração está acontecendo para que seja possível determinar quantos funcionários por plantão o governo irá pagar hora extra.

Sistemas Administrativos – Se for *full time* sempre deverá ter um funcionário e mais um adicional de 2 caso o sistema tenha mais 200 usuários simultâneos.

Sistemas Corporativos – Se for *full time* sempre deverá ter dois funcionários e caso entre 3000 e 5000 acessos inclusive, um adicional de 2, caso tenha mais de 5000 mil acessos, um adicional de 3.

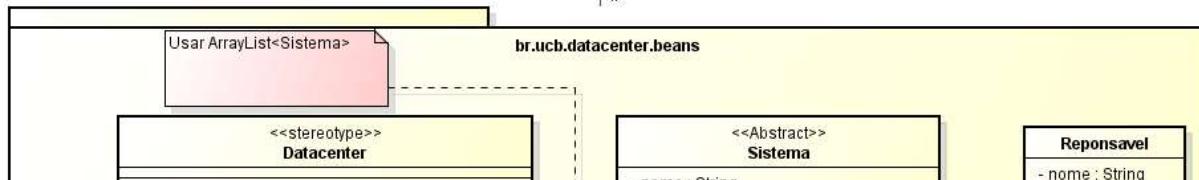
O gerente, enquanto desejar, informará os dados dos sistemas.

Sabe-se que a classe Sistema (a mais genérica deve ser abstrata, assim como o método de saber a quantidade de funcionários).

Ao término do cadastro, ele deve ter acesso às seguintes informações:

- Apresente todos os sistemas.
- O sistema que demanda mais funcionários.
- Os sistemas que o responsável tenha "Steve" em seu nome.

Uma proposta de projeto foi elaborada para este exercício a fim de auxiliá-lo a organizar suas entidades de negócio. Faça sua implementação a partir do projeto abaixo e complemente com as classes de entradas de dados e fluxo do programa.



## Atividade 04

Hoje pela manhã, a França iniciou a caça ao mentor dos ataques terroristas em Paris. Entre os envolvidos, 2 cometem suicídio e outros foram capturados. O governo francês declarou que não vai parar por aí. Entretanto, o governo precisa de um sistema para controlar os Terroristas capturados pela polícia especializada, que cometem atos terroristas ou que morreram em consequência disto.

A polícia afirma que existem dois tipos de terroristas: os suicidas e não suicidas. Sobre os suicidas, a polícia deseja catalogar o nome, a religião, a quantidade de explosivos encontrados. Já, para os não suicidas, deseja saber o nome, a quantidade de explosivos e o País de origem.

A polícia afirma também que existe um grau de periculosidade dos terroristas que é calculado de forma distinta para cada um dos casos, conforme abaixo:

**Suicida** – é o produto do dobro da quantidade de explosivos com a constante de Religião. Se for do Islamismo, 5. Caso contrário, 2.

**Não suicida** – é o triplo da quantidade de explosivos adicionado com a constante do País. Se for da Arábia, 7. Senão, 5.

Após receber todos os dados necessários de todos os terroristas, faça o seguinte:

- Apresente todas as informações de todos os terroristas informados pelo usuário.
- Apresente todos os terroristas que possuem grau de periculosidade maior que 50.
- Mostre todos os terroristas que possuem MOHAMED no nome, em qualquer formato.
- A quantidade de terroristas encontrados com mais de 10 explosivos.
- O percentual de terroristas que não são do 'Egito'.

# Saiba Mais

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

- Na postagem "[Ordenando coleções com Comparable e Comparator ↗](#)", você poderá acessar outras implementações. Confira!

# 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](https://pt.wikipedia.org/wiki/Ambiente_de_desenvolvimento_integrado)>. Acesso em: 5 mar. 2017.