**Programação Orientada à Objetos – (POO)**

A Programação Orientada à Objetos é atualmente um dos paradigmas mais utilizados por empresas e desenvolvedores no mundo. Esse modelo utilizado para o desenvolvimento de software revolucionou e facilitou a vida dos programadores em 101%.

A orientação à objetos é um paradigma que possui 4 pilares fundamentais, sendo eles:

* **Abstração**
* **Encapsulamento**
* **Herança**
* **Polimorfismo**

**Abstração**

A **abstração** na orientação à objetos é onde tudo começa, é o ponto chave para que a abstração do mundo real seja transformada em um sistema. No caso da orientação à objeto, as abstrações são adotadas no formato de “classes”.

Uma **classe** é uma abstração de um objeto do mundo real para o software, uma classe possuí atributos e métodos. O seguinte exemplo abaixo irá mostrar como funciona o processo de abstração:

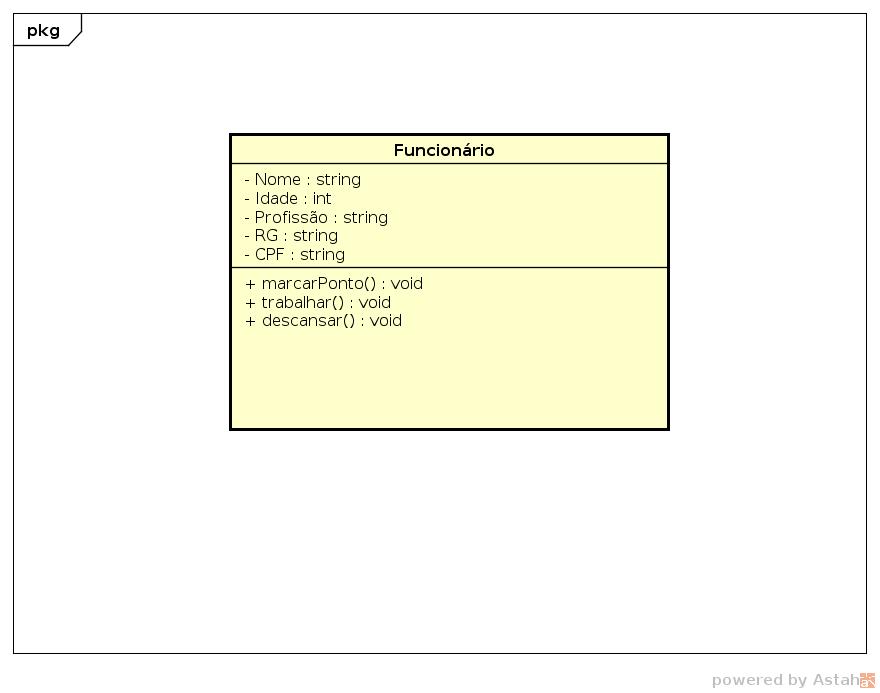
Vamos considerar o seguinte problema:

1. Preciso criar uma classe que represente a abstração de um funcionário de uma empresa.
2. Preciso definir quais são os atributos e métodos dessa classe que representará o meu funcionário.

Um **funcionário** possui as seguintes características, que na orientação à objetos conhecemos como ‘atributos’: nome, idade, profissão, rg, cpf e entre outros.

Um **funcionário** pode executar as seguintes funções: marcar ponto, trabalhar, descansar e etc.

Essa é abstração de um **funcionário** passada para a orientação à objetos.



**Encapsulamento**

O encapsulamento é uma das principais técnicas que define a programação orientada a objetos. Se trata de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta.

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados getters e setters, que irão pegar e alterar os valores de uma propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

**Exercícios**

1. Crie uma classe com o nome de “Funcionário” e implemente os atributos e métodos citados no diagrama acima.

2 – Após a criação da classe, no programa principal faça a instanciação de objetos da classe “Funcionário” e altere os seus atributos e faça a chamada dos seus métodos.

**Modificadores de Acesso (Visibilidade)**

Uma parte também muito importante da programação orientada à objetos são os modificadores de acesso, ou também conhecidos como: “visibilidade de métodos e atributos”. Os modificadores de acesso têm a responsabilidade de dar “permissões” às respectivas classes que poderão acessar métodos e atributos de uma outra.

Os modificadores de acesso mais usados e as suas funcionalidades são os seguintes:

**Public -** Os métodos e atributos são acessíveis em qualquer classe.

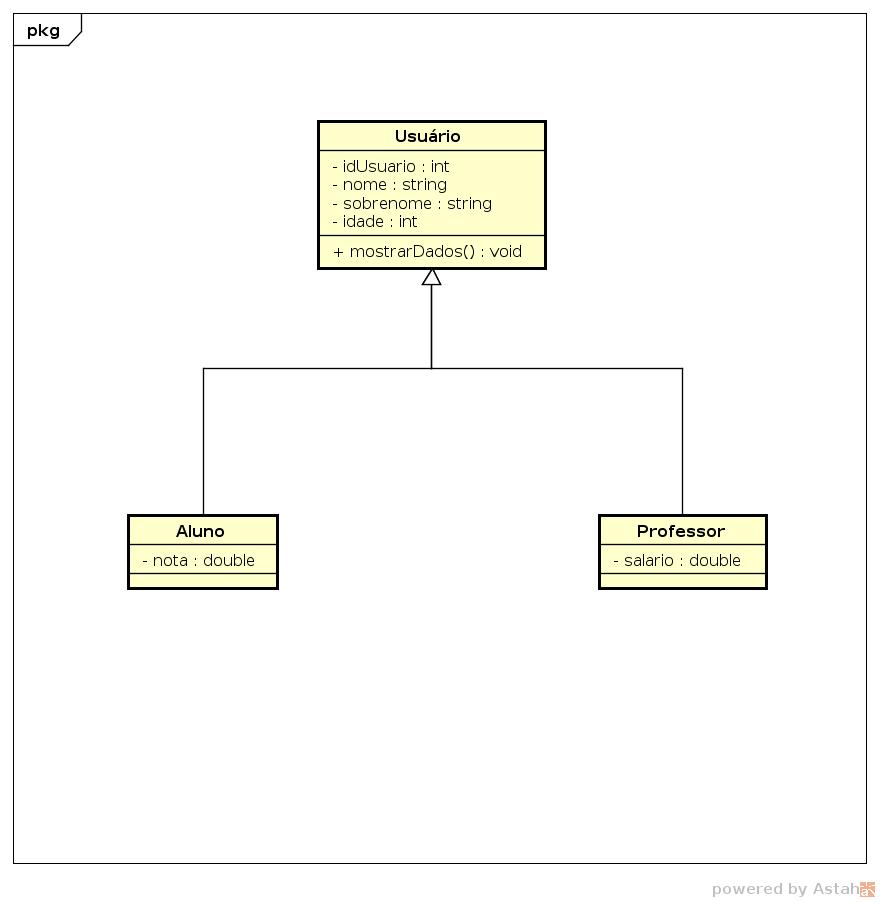
**Private -** Os métodos e atributos são acessíveis somente dentro da própria classe.

**Protected -** Os métodos e atributos são acessíveis somente dentro da própria classe ou por uma subclasse. Ou seja, por uma classe que herdou da mesma.

**Herança**

O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como herança. Essa característica otimiza a produção da aplicação em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus “ancestrais”. A herança a partir das características do objeto mais acima é considerada herança direta, enquanto as demais são consideradas heranças indiretas. Por exemplo, na família, a criança herda diretamente do pai e indiretamente do avô e do bisavô.



A imagem acima mostra um ciclo de **herança** representado por um diagrama de classes, onde três classes são definidas. Sendo elas: Usuário, Aluno e Professor. Onde a classe “Usuário” é a classe pai e “Aluno” e “Professor” são as classes filhas, também conhecidas como “*subclasses”*. Estas classes herdam métodos e atributos de sua classe pai, e também pode ter métodos e atributos próprios definidos dentro dela mesma.

**Exercícios**

1. Crie uma classe abstrata com o nome de “Eletrônico”, uma classe com o nome de “Rádio”, uma classe com o nome de “Televisão” e implemente os atributos e métodos base na classe “Eletrônico”. Após, herde os mesmos nas classes “Rádio” e “Televisão” e crie atributos e métodos específicos para cada uma destes. Finalizada a construção dessas classes, no programa principal, instancie objetos dessas classes e faça a alteração dos atributos e a chamada dos métodos.

**Polimorfismo**

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

**Exemplo:**

Suponha que temos uma classe chamada “Aparelho Eletrônico”, nós sabemos existem vários tipos de aparelhos eletrônicos, certo? Um aparelho eletrônico pode ser um rádio, televisão, celular, câmera fotográfica e entre outros.

Agora, voltando um pouco atrás onde falamos que uma classe possui métodos e atributos, vamos considerar que a classe “Aparelho Eletrônico” seja a nossa classe genérica, ou seja: um aparelho eletrônico pode assumir diversas formas, das quais foram mencionadas anteriormente.

Agora, vamos definir que a nossa classe “Aparelho Eletrônico” possui um método chamado “**ligar**”. Como foi analisado antes, um aparelho eletrônico pode assumir diversas formas diferentes, e todos eles necessitam ser **ligados** para funcionar, certo? Mas nem todos eles são ligados da mesma forma. Exemplo: a forma de ligar um rádio pode ser a de ligar uma câmera fotográfica e entre outros, mas todos eles executam o método ligar.

Graças a esse tipo de abordagem, podemos reutilizar o método **ligar** da nossa classe genérica “Aparelho Eletrônico” em classes que assumem a forma de um aparelho eletrônico.

**Exercícios**

1. Na classe “Eletrônico” implementada anteriormente, crie um método abstrato chamado “EmitirSom” e o sobrescreva nas classes “Rádio” e “Televisão”.

**ASP NET MVC**

O padrão arquitetônico MVC (Model-View-Controller) separa um aplicativo em três componentes principais: modelo, exibição e controlador. A estrutura ASP.NET MVC oferece uma alternativa ao padrão Web Forms do ASP.NET para criar aplicativos Web. A estrutura ASP.NET MVC é uma estrutura de apresentação leve e altamente testável que (à semelhança dos aplicativos baseados em Web Forms) é integrada aos recursos ASP.NET existentes, como páginas mestras e autenticação baseada em associação. A estrutura MVC é definida no assembly **System.Web.Mvc**.

O MVC é um padrão de design padrão que muitos desenvolvedores conhecem. Alguns tipos de aplicativos Web irão se beneficiar da estrutura MVC. Outros vão continuar usando o padrão de aplicativo ASP.NET tradicional que é baseado em Web Forms e postbacks. Outros tipos de aplicativos Web irão combinar as duas abordagens; uma abordagem não exclui a outra.

O padrão MVC ajuda a criar aplicativos que separam os diferentes aspectos do aplicativo (lógica de entrada, lógica de negócio e lógica da IU), enquanto fornece um acoplamento flexível entre esses elementos. O padrão especifica onde cada tipo de lógica deve ficar localizado no aplicativo. A lógica da IU fica na exibição. A lógica de entrada fica no controlador. A lógica de negócios fica no modelo. Essa separação ajuda a administrar a complexidade quando você cria um aplicativo, porque ela permite que você se concentre em um aspecto da implementação por vez. Por exemplo, você pode se concentrar na exibição sem depender da lógica de negócios.

O acoplamento flexível entre os três componentes principais de um aplicativo MVC também promove o desenvolvimento paralelo. Por exemplo, um desenvolvedor pode trabalhar na exibição, um segundo desenvolvedor pode trabalhar na lógica do controlador e um terceiro desenvolvedor pode se concentrar na lógica de negócios do modelo.

**Por que utilizar MVC?**

Com o aumento da complexidade dos sistemas/sites desenvolvidos hoje, essa arquitetura tem como foco dividir um grande problema em vários problemas menores e de menor complexidade. Dessa forma, qualquer tipo de alterações em uma das camadas não interfere nas demais, facilitando a atualização de layouts, alteração nas regras de negócio e adição de novos recursos. Em caso de grandes projetos, o MVC facilita muito a divisão de tarefas entre a equipe.  
  
Abaixo serão listadas algumas das vantagens em utilizar MVC em seus projetos:

* Facilita o reaproveitamento de código;
* Facilidade na manutenção e adição de recursos;
* Maior integração da equipe e/ou divisão de tarefas;
* Diversas tecnologias estão adotando essa arquitetura;
* Facilidade em manter o seu código sempre limpo;

**Definição das camadas**

A estrutura MVC inclui os seguintes componentes:

* **Modelos(Models)**. Os objetos de modelo são as partes do aplicativo que implementam a lógica para o domínio de dados do aplicativo. Muitas vezes, os objetos de modelo recuperam e armazenam o estado do modelo em um banco de dados. Por exemplo, um objeto Product pode recuperar informações de um banco de dados, operar nele e, em seguida, gravar informações atualizadas de volta em uma tabela de Produtos em um banco de dados do SQL Server.

Em aplicativos pequenos, o modelo, muitas vezes, é uma separação conceitual em vez de física. Por exemplo, se o aplicativo apenas ler um conjunto de dados e enviá-lo para exibição, o aplicativo não terá uma camada de modelo físico nem classes associadas. Nesse caso, o conjunto de dados assume a função de um objeto de modelo.

O modelo (Model) é utilizado para manipular informações de forma mais detalhada, sendo recomendado que, sempre que possível, se utilize dos modelos para realizar consultas, cálculos e todas as regras de negócio do nosso site ou sistema.

* **Exibições(Views)**. As exibições são os componentes que exibem a interface do usuário (IU) do aplicativo, ou seja, as telas. Normalmente, esta IU é criada a partir dos dados do modelo. Um exemplo seria uma exibição de edição de uma tabela de Produtos que mostra caixas de texto, listas suspensas e caixas de seleção com base no estado atual de um objeto Product. A visão (View) é responsável por tudo que o usuário final visualiza, toda a interface, informação, não importando sua fonte de origem.
* **Controladores(Controllers)**. Os controladores são os componentes que lidam com a interação do usuário, trabalham com o modelo e, finalmente, selecionam uma exibição de renderização que mostra essa IU(View). Em um aplicativo MVC, a exibição só mostra informações; o controlador manipula e responde à entrada e à interação do usuário. Por exemplo, o controlador manipula valores da cadeia de consulta e passa esses valores ao modelo, que por sua vez pode usar estes valores para consultar o banco de dados.

**Exemplo do funcionamento do MVC**

Com a teoria já compreendida, vamos imaginar a seguinte situação: Você desenvolveu um site, e esse site possui uma tela de login, onde o usuário digita seu login e sua senha, após a autenticação, caso ocorra tudo certo, o usuário acessa a área restrita do site, caso contrário é redirecionado novamente para a página de login repassando uma mensagem que a combinação de usuário e senha é inválida.  
  
Conseguiu imaginar essa situação? Beleza... Agora veja como isso acontece caso você ainda não tenha adotado a arquitetura MVC em seu site: Primeiramente, o usuário preenche o formulário com seu login e sua senha e pressiona o botão “Logar”.   
  
Depois disso, o formulário envia essas informações para um arquivo onde, no mesmo arquivo, você executa as seguintes etapas:  
  
1. Armazena em variáveis os dados digitados pelo usuário;  
2. Montam um comando SQL para selecionar o usuário;  
3. Verifica se retornou alguma informação;

* Se retornar alguma informação, armazena o usuário em uma sessão e redireciona para a área restrita;
* Se não retornar nenhuma informação, redireciona para a página de login com uma mensagem notificando que a combinação digitada é inválida;

Aparentemente está tudo ok, tudo funcionado. Mas veremos agora como funcionaria se o seu site estivesse utilizando a arquitetura MVC: Os passos seguem os mesmos, primeiramente, o usuário preenche o formulário com seu login e sua senha e pressiona o botão “Logar”. Agora veremos algumas mudanças.   
  
**Depois disso, o formulário envia essas informações para uma controladora, e essa controladora realizará as seguintes etapas:**  
  
1. A controladora (controller) carrega um modelo (model), e executa um método que realiza a validação;

2. No modelo (model) são executadas as seguintes tarefas:

* Armazena as informações digitadas pelo usuário;
* Realiza a consulta no banco retornando verdadeiro (true) em caso de sucesso (usuário existe e a senha bate), ou falso (false) no caso de a combinação das informações digitadas serem inválidas;

3. A controladora (controller) verifica o que o modelo retornou;

* Se retornar verdadeiro (true) armazena as informações em uma sessão e redireciona o usuário para visão (view) da área restrita;
* Se retornar falso (false) redireciona o usuário de volta para a tela (view) de login repassando a mensagem que a combinação digitada é inválida;

Agora você pode ficar se perguntando, mas do modo que eu faço também funciona? Pode ser que sim, mas imagine ter que alterar a regra de negócio. Antes de utilizar MVC você precisaria abrir o arquivo que realiza todas as tarefas e localizar a sua regra, para depois alterar. No caso do MVC, você já sabe onde se encontra as suas regras de negócio, então você vai direto ao arquivo. Por isso que a maioria dos frameworks já vem com sua estrutura de diretórios pronta, facilitando a localização dos arquivos.

**ViewData, ViewBag e TempData**

* Similaridades entre ViewData e ViewBag

ViewData e ViewBag são similares nas seguintes características:

* São utilizadas para persistir dados entre a Controller e a View correspondente.
* A duração “tempo de vida” é apenas entre o envio através da Controller e a exibição na View, depois disso tornam-se nulas novamente.
* No caso de um redirect se tornam nulas.

## Diferenças entre ViewData e ViewBag

|  |  |
| --- | --- |
| **ViewData** | **ViewBag** |
| É um dicionário de objetos derivado de ViewDataDictionary e é acessível utilizando strings como chaves. | É uma propriedade dinâmica baseada na funcionalidade “dynamic” do C# 4.0 |
| Requer typecasting (conversão) quando associada a tipos complexos. | Não necessita de conversão para tipos complexos. |

**Exemplos de aplicação**

Controller

Uma imagem contendo texto

Descrição gerada com alta confiança

View

Uma imagem contendo interior

Descrição gerada com alta confiança

Resumindo, ViewData e ViewBag possuem a mesma proposta, porém o ViewBag está disponível a partir do ASP.Net MVC 3, enquanto o ViewData existe desde a primeira versão.

O ViewData é um wrapper, uma implementação do ViewBag, pois utiliza o ViewBag internamente, portanto:

Por este motivo ViewData é mais rápido que o ViewBag, porém essa diferença de velocidade é mínima, não é necessário deixar de usar o ViewBag por este motivo.

**TempData**

* TempData assemelha-se mais a uma sessão de servidor, porém de curta duração.
* Possui um tempo de vida maior que o ViewBag e ViewData, o TempData perdura desde sua criação até que seja chamado, ou seja, quando houver um request da informação do TempData, ele se tornará nulo novamente.
* Uma informação em TempData criada em um Controller persiste após um redirect entre actions (apenas um) e pode ser exibido em sequência em uma View (muito usado em tratamento de erros).
* Caso não seja chamado o TempData pode manter o estado de seus dados até que a sessão do usuário se encerre.
* É utilizado para compartilhar informações entre Controllers.
* O TempData salva suas informações no SessionState do servidor.
* Após a leitura os dados do TempData são marcados para deleção, ou seja, no final do request todos os dados marcados serão deletados.
* É um benefício quando necessário transmitir um volume de informações entre as Controllers sem se preocupar em zerar os valores, pois o TempData automaticamente faz isso.

**Exemplo de aplicação**

* **Controller**

**Uma imagem contendo captura de tela, texto

Descrição gerada com alta confiança**

Neste exemplo pudermos entender que o propósito do TempData é compartilhar dados entre Controllers, portanto sua duração persiste até que a informação seja lida.  
Outro detalhe é sempre checar se o TempData não está nulo.

Caso você queira manter o dado de um TempData mesmo após a leitura, basta chamar o método Keep(), assim o dado será persistido novamente até a próxima requisição.

* **View**

Uma imagem contendo parede, interior, monitor, tela

Descrição gerada com alta confiança

Recomenda-se utilizar sempre ViewBag e ViewData para transferência de dados entre Controller e View. O TempData em Views é recomendado no caso de um dado necessitar ser redirecionado entre Actions e posteriormente ser exibido numa View (ViewBag e ViewData são anulados em redirects).