

## Baixe o conteúdo completo em PDF

### Ferramentas indispensáveis

- Se virando no Linux
- Se virando no Git

### Java

- Introdução ao Java
- Orientação a Objetos

### Banco de dados

- Começando com o PostgreSQL
- Entendendo migrações de banco de dados
- Hibernate

### MVC, HTTP e Spring

- Como funciona o HTTP e a Web
- Introdução ao Spring

### Front-end

- HTML
- CSS
- JavaScript
- Thymeleaf
- Bulma

### Infraestrutura, Integração e entrega contínua

- Heroku
- Docker
- Circle CI

# Se virando no Linux

## Comandos Básicos

---

**cd [nome da pasta/]** Comando para entrar ou mudar de pasta.

**cd ..** Volta um nível.

**ls** Lista conteúdo da pasta onde o usuário se encontra.

**ls -l** Lista todos os itens da pasta atual com detalhes

**mkdir [nome da pasta]** Cria nova pasta onde o usuário se encontra.

**rm -rf [nome da pasta]** Remove pasta especificada.

**> [nome do arquivo]** Cria um arquivo onde o usuário se encontra.

**rm [nome do arquivo]** Remove o arquivo.

**mv item destino/caminho** Move o arquivo (ou pasta) ao destino escolhido (se o seu destino não existir, o arquivo será renomeado com tal).

**cp item copia-do-item** Cria uma cópia do arquivo (ou pasta).

**man [nome do comando]**

O comando "man" exhibe a função de determinado comando. Ele é muito útil quando não se sabe o que um comando faz, ou quando se pretende aprender mais sobre a sua utilização. Aconselha-se a leitura do manual sempre que houver dúvidas.

## Flags

---

Flags são diferentes funcionalidades do comando que podemos utilizar, ou seja, moldar o que o comando irá fazer.

Tomemos o comando **ls** como exemplo. Como mencionado anteriormente, este comando serve para listar o conteúdo de um diretório. Temos a seguinte estrutura de pastas:

sh

```
uma-pasta/  
├─ arquivo.txt  
└─ pasta-filha  
    └─ pasta-neta
```

Quando estamos dentro da pasta **uma-pasta** e executamos o comando **ls**, a saída será:

sh

```
$ ls  
arquivo.txt pasta-filha
```

Podemos usar as **flags** para modificar o comportamento do comando ls. Passando a flag **-l**, o comando mostrará a saída em formato de lista:

sh

```
$ ls -l  
  
total 12  
-rw-r--r-- 1 yrachid yrachid  0 jan 23 11:27 arquivo.txt  
drwxr-xr-x 3 yrachid yrachid 4096 jan 23 11:27 pasta-filha
```

A maioria dos comandos, possui a flag **-h** ou **--help**, que irá mostrar todas as flags que você pode usar para aquele comando.

## Formatos de flags

Geralmente, as flags dos comandos possuem dois formatos: **abreviado** e **expandido**.

Por exemplo, quando chamamos o comando como no exemplo **rm -rf** a parte em que usamos **-rf** estamos falando para o comando que queremos fazer um **remove** recursivo (flag **-r**) e forçado (flag **-f**). Neste caso, usamos o formato **abreviado** destas flags, mas também poderíamos utilizar seus formatos expandidos:

sh

```
rm --recursive --force
```

## Exemplos

**man ls** – Exibe o que faz o comando ls e quais são suas variações. Para sair do man pressione a tecla “q”.

## Exercícios de Fixação

Baseado no material desta apostila informe os comandos necessários para realizar cada uma das tarefas a seguir:

Imagine que estamos na seguinte estrutura de diretórios:

- Liste os arquivos do diretório **aceleradora** .
- Entre no diretório **pasta** .
- Dentro do diretório **pasta** crie um arquivo chamado **arquivo02.txt** .
- Volte ao diretório anterior.
- Sem sair do diretório atual, copie o arquivo **compactado.zip** para dentro do diretório **pasta** .
- Ainda no seu diretório atual, exclua o arquivo **compactado.zip** .
- Mova tudo o que há dentro do diretório **pasta** para o diretório **aceleradora** .
- Exclua o diretório **pasta** .
- Crie um diretório chamado **[seu-nome]** .
- Entre no diretório **[seu-nome]** .
- Crie três arquivos, tais com nomes de três características suas.
- Liste detalhadamente os arquivos do diretório **[seu-nome]** .

# Se virando no Git

## Git

---

Git é um sistema de controle de versão de arquivos. Através deles podemos desenvolver projetos na qual diversas pessoas podem contribuir simultaneamente no mesmo, permitindo que os mesmos possam existir sem o risco de suas alterações serem sobrescritas. Se não houvesse um sistema de versão, imagine o caos entre duas pessoas abrindo o mesmo arquivo ao mesmo tempo. Uma das aplicações do git é justamente essa, permitir que um arquivo possa ser editado ao mesmo tempo por pessoas diferentes.

## Instalação

Muitas das distribuições do Linux mais atuais já vêm com o git instalado. Verifique se o git já está instalado em sua máquina. No terminal, execute:

```
git --version
```

sh

A saída do comando deve ser algo similar ao seguinte:

```
git version 2.17.2
```

Caso você não tenha o git na máquina, instale-o com este comando (via terminal - Linux):

```
sudo apt-get install git
```

sh

Depois de instalar, a primeira coisa que você deve fazer é configurar o Git. Para isso, abra uma janela de terminal e digite os seguintes comandos:

```
git config --global user.name "Seu Nome"  
git config --global user.email "seu@email.com"
```

sh

A partir daí, o Git irá usar essas informações para registrar quem foi que fez as alterações nos arquivos.

Após isso já poderá realizar os comando do git pelo terminal.

## Nota

Esse exemplo configura o mesmo usuário para todos os projetos presentes no computador (isso se dá por conta da flag `--global`). Podemos também configurar usuários para cada projeto individualmente, basta acessar a pasta do projeto em questão e executar os mesmos comandos do exemplo sem a flag `--global`.

## Comando básicos

---

### Criando um repositório:

Criar um repositório no Git é muito simples, apenas siga esta sequência completa dos comandos:

**mkdir meu-projeto** (irá criar o diretório)

**cd meu-projeto** (irá entrar dentro do diretório)

**git init** (irá criar o repositório git)

Após isso, basta começar a trabalhar, criando, removendo e alterando arquivos.

### Outros Comandos:

**git clone [link do repositório]** Para clonar o projeto do repositório;

**git checkout -b [nome da branch]** Para criar uma nova branch;

**git checkout [nome da branch]** Para trocar de branch;

**git status** Para você visualizar os arquivos que modificou;

**git add .** Para você adicionar as modificações feitas;

**git commit -m ["mensagem"]** Para você comentar brevemente sobre as modificações feitas;

**git push origin [nome da branch]** Sobe as alterações feitas para a branch remota(para onde quero enviar, por isso usamos *origin*) do **GitHub**

**git pull origin master** Recebe as alterações feitas na branch remota origin master;

Mas se houver conflitos:

**git pull -r origin master** Facilita na hora de resolver conflitos.

Ex: Deram push em uma atualização de código da linha 11, e você localmente modificou esta linha e quer dar push, o git ficará sem saber o que fazer e resultará em conflitos, você terá que escolher entre uma das atualizações,ou em deixar as duas.

Após isso para ter certeza de que não tenha mais nada diferente execute:

**git rebase --continue**

**Nota:** Este comando só funcionará se usar a flag `-r` no **git pull**

Se não houver mais conflito pode dar seu push tranquilo, caso contrario terá que resolver todos os conflitos para dar push.

```
git reverse --hard HEAD~1
```

 Exclui commit local

Caso já tenha enviado ao seu repositório será necessário executar este comando também para excluí-lo:

```
git push origin HEAD --force
```

**Importante:** Não é muito recomendável usar estes últimos dois comandos exceto em casos muito extremos, eles podem causar grandes complicações.

## GitHub

---

O GitHub é uma plataforma de hospedagem de código-fonte com controle de versão usando Git. Nele criamos repositórios onde colocamos nossos projetos que vamos desenvolver. No Github o projeto é dividido em branches, elas são separações de código. Normalmente são utilizados para separar alterações ou novas funcionalidades do projeto.

## Exercícios de Fixação

---

Baseado no material desta apostila informe os comandos necessários para realizar cada uma das tarefas a seguir:

Imagine que criamos um repositório no GitHub:

- [ ] Clone este repositório para seu computador
- [ ] Crie uma branch
- [ ] Entre na branch criada
- [ ] Envie as modificações para o repositório
- [ ] Acesse o site do GitHub e crie um repositório chamado **exercicioGitHubAcelera**
- [ ] Agora, dentro de algum diretório de seu computador, inicie um repositório Git local
- [ ] Clone este repositório ( **exercicioGitHubAcelera** ) para seu computador
- [ ] Faça qualquer alteração neste diretório (crie arquivos novos, modifique algum existente e etc) e em seguida adicione estas alterações neste repositório Git local
- [ ] Realize um **commit** destas alterações ao seu repositório Git local, informando uma mensagem explicando o que esta sendo salvo neste **commit**
- [ ] Agora, envie as modificações para o repositório

# HTML

A estrutura básica de uma página HTML pode ser vista na Listagem 1, na qual podemos ver as principais tags que são necessárias para que o documento seja corretamente interpretado pelos browsers.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Document</title>
  </head>
  <body>
    <!-- Comentario -->
  </body>
</html>
```

html

**Linha 1:** A instrução DOCTYPE deve ser sempre a primeira a aparecer em uma página HTML e indica para o browser qual versão da linguagem está sendo usada. Nesse caso, estamos trabalhando com a HTML5, versão na qual a declaração do DOCTYPE é bastante simples, como podemos ver na listagem;

**Linhas 2 e 10:** Abertura e fechamento da tag html, que delimita o documento. Sendo assim, todas as demais tags da página devem estar nesse espaço;

**Linhas 3 e 6:** Abertura e fechamento da tag head, que define o cabeçalho do documento. O conteúdo neste espaço não é visível no browser, mas contém instruções sobre seu conteúdo e comportamento. Dentro dessa tag, por exemplo, podem ser inseridas folhas de estilo e scripts;

**Linha 4:** A tag meta, nesse caso, especifica qual conjunto de caracteres (character set ou charset) será usado para renderizar o texto da página. O **UTF-8** contém todos os caracteres dos padrões Unicode e ASCII, sendo, portanto, o mais utilizado em páginas web.

**Linha 5:** A tag title define o título da página, aquele que aparece na janela/aba do navegador;

**Linhas 7 e 9:** Abertura e fechamento da tag body, marcando o espaço no qual deve estar contido o conteúdo visual da página. As demais tags que representam texto, botões, etc. devem ser adicionadas nesse intervalo;

**Linha 8:** Nessa linha podemos observar a sintaxe para adição de comentários em HTML. Esse trecho não é renderizado pelo browser.

## Antes de seguir

---



Antes de continuar a leitura, recomendamos que você acesse a ferramenta [codepen.io](https://codepen.io) 

Esta ferramenta permite ver o resultado do código HTML, CSS e JavaScript que escrevemos.

Recomendamos que você reproduza os exemplos de cada seção abaixo para ver como os resultados serão exibidos.

Recomendamos fortemente que você não apenas copie e cole os exemplos de código, mas que os reproduza, escrevendo você mesma. Isto vai lhe ajudar a começar a fixar a ideia de como escrever HTML e lidar com erros de sintaxe e tudo o mais.

## Estruturas HTML

---

Um documento HTML é composto por **tags**, as quais possuem um nome e aparecem entre os sinais `<` e `>`, por exemplo, em `<html>` e `<head>`. Naquele exemplo também vimos que algumas tags precisam ser abertas e fechadas, como em `<body></body>`. Nesse caso, a tag de fechamento deve conter a barra `/` antes do nome. Outras, porém, não precisam ser fechadas, como a tag `<meta>`. Nesses casos, a adição da barra `/` no final da própria tag é opcional.

## Títulos

Títulos são normalmente utilizados para identificar páginas e seções, e possuem aparência diferenciada do restante do texto. No HTML há seis níveis de títulos que podem ser utilizados por meio das tags `h1`, `h2`, `h3`, `h4`, `h5` e `h6`, sendo `h1` o maior/mais relevante e `h6` o menor/menos relevante.

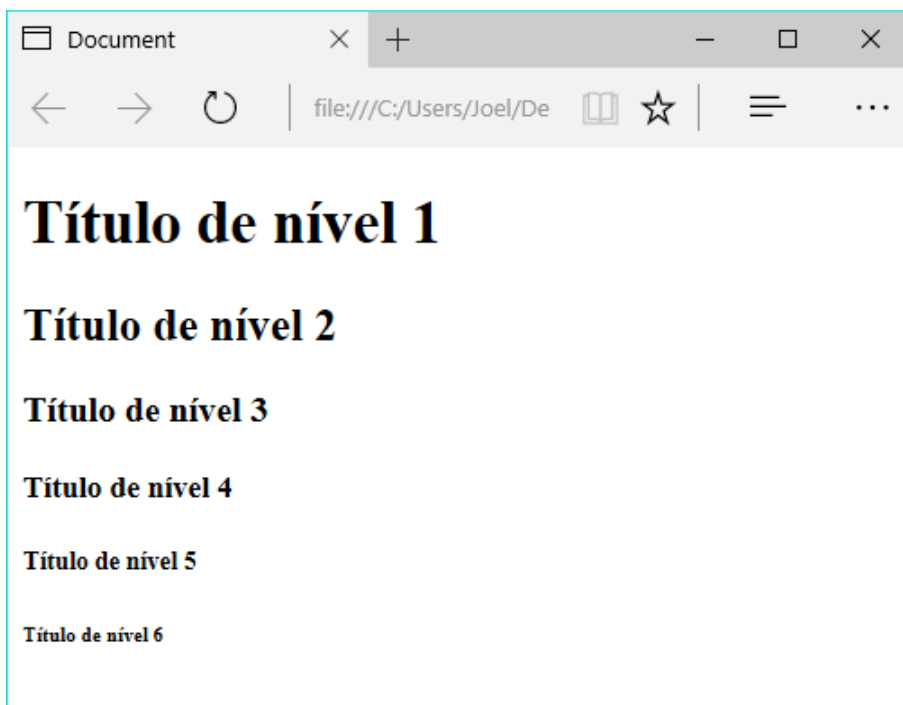
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Document</title>
  </head>

  <body>
    <h1>Título de nível 1</h1>
    <h2>Título de nível 2</h2>
    <h3>Título de nível 3</h3>
    <h4>Título de nível 4</h4>
    <h5>Título de nível 5</h5>
    <h6>Título de nível 6</h6>
  </body>
</html>
```

html

Valendo lembrar que essas tags servem apenas para atribuir a importância do título (título, subtítulo, etc) e não apenas para regular o tamanho. O tamanho (bem como outras características como fonte e cor, por exemplo) poderá ser regulado através de CSS.

O que resultaria em:



## Parágrafos

Parágrafos de texto são gerados na HTML por meio das tags `<p>` `</p>`. Esse é um exemplo de tag cuja disposição na tela se dá em forma de bloco, ou seja, um parágrafo é posto sempre abaixo do outro.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Document</title>
  </head>

  <body>
    <p>Sou um parágrafo</p>
    <p>Sou outro parágrafo</p>
  </body>
</html>
```

## Imagens

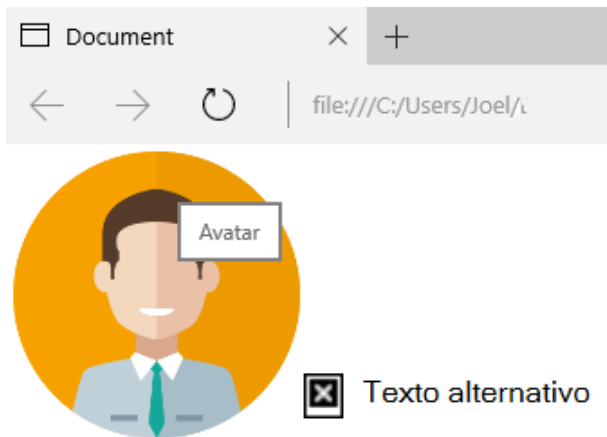
A inserção de imagens em uma página HTML pode ser feita por meio da tag `img`, que recebe no atributo `src` o endereço do arquivo a ser carregado. Além desse, outros dois atributos importantes são o `alt`, que indica um texto alternativo que será exibido caso o arquivo não possa ser carregado, e `title`, que indica o texto que aparecerá como tooltip ao passar o mouse sobre a figura.

```


```

O código a seguir insere uma imagem a partir do arquivo perfil.png, localizado na mesma pasta do arquivo HTML, e uma tag img apontando para um arquivo inexistente. Observe na Figura 3 que enquanto a primeira aparece corretamente, a segunda é exibida com um ícone de erro o texto alternativo que informamos. Note, ainda, que sobre a primeira figura está sendo exibido o tooltip definido no atributo title.

Resultado:



## Links

Links são normalmente utilizados para direcionar o usuário para outras páginas, ou para outras partes da mesma página. Nos dois casos, utilizamos a tag `a`, que possui o atributo `href` no qual indicamos o destino daquele link. O seguinte código mostra como adicionar um link para outra página, neste caso, indicada pelo arquivo `pagina2.html`.



Já o código abaixo mostra como adicionar um link para um elemento na mesma página. Nesse caso, ao clicar no link o browser mudará o foco para o elemento que possui o atributo `id` igual àquele indicado no `href`.



## Tabelas

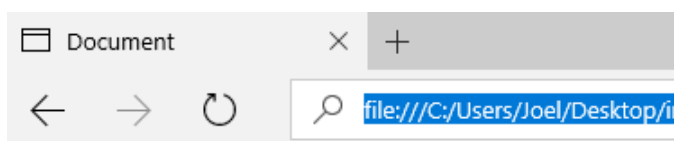
Tabelas são elementos utilizados com frequência para exibir dados de forma organizada em linhas e colunas. No HTML, elas são formadas por três tags básicas: `table`, para delimitar a tabela; `tr`, para indicar as linhas; e `td` para formar as colunas.

```

<table>
  <tr>
    <td>Linha 1, Coluna 1</td>
    <td>Linha 1, Coluna 2</td>
  </tr>
  <tr>
    <td>Linha 2, Coluna 1</td>
    <td>Linha 2, Coluna 2</td>
  </tr>
  <tr>
    <td>Linha 3, Coluna 1</td>
    <td>Linha 3, Coluna 2</td>
  </tr>
</table>

```

Este seria o resultado do código:



Linha 1, Coluna 1	Linha 1, Coluna 2
Linha 2, Coluna 1	Linha 2, Coluna 2
Linha 3, Coluna 1	Linha 3, Coluna 2

Existem ainda outras três tags utilizadas para delimitar, de forma mais organizada, as partes da tabela: `thead` para o cabeçalho; `tbody` para o corpo; e `tfoot` para o rodapé.

Exemplo de tabela mais complexa, utilizando todas as tags:



Observação: por padrão, as tabelas não possuem bordas, isto pode ser adicionado por meio de CSS. Então a tabela ficará assim:



## Formulários

Formulários são normalmente utilizados para integrar a página HTML a algum processamento no lado servidor. Nesses casos, a página envia dados para uma aplicação (Java, PHP, .NET, etc), que os recebe, trata e retorna algum resultado.

No HTML, geralmente usamos a tag `form` para delimitar a área na qual se encontram os campos a serem preenchidos pelo usuário, a fim de serem enviados para processamento no back-end (enquanto a página HTML é chamada de front-end da aplicação).

A figura mostra um exemplo de formulário com vários tipos de campos para entrada de dados e um botão para submetê-los ao servidor.



Como ficaria no browser:



Formatação de texto As tags de formatação de texto ajudam a destacar trechos da parte escrita da página, seja para fins de SEO ou por requisitos do conteúdo. Formatações como negrito e itálico podem ser aplicadas com facilidade utilizando as várias tags disponíveis para esse fim:

- **b** e **strong** para **negrito/texto** forte;
- **i** e **em** para **itálico/ênfase**;
- **small** para textos menores que o padrão;
- **mark** para texto destacado.

```
<p>Texto em negrito com <b>bold</b> e <strong>strong</strong>.</p>
<p>Texto em itálico com <i>italics</i> e <em>emphasis</em>.</p>
<p>Texto <sup>sobrescrito</sup> e <sub>subscrito</sub>.</p>
<p>Texto <ins>inserido</ins> e <del>excluído</del>.</p>
<p>Texto <small>pequeno</small> e <mark>destacado</mark>.</p>
```

```
<p>Texto <small>pequeno</small> e <mark>destacado</mark>.</p>
```

## Div e Span

As tags `div` e `span` são duas das mais utilizadas no HTML, com objetivos distintos, porém com grande importância para a composição do layout das páginas e formatação do texto.

As `div`s são normalmente utilizados para representarem containers para outros elementos, agrupando-os visualmente dentro de um bloco que pode conter dimensões e posição definidas. Por padrão, uma `div` não possui aparência características visuais definidas, isso precisa ser feito via CSS ao atribuir bordas, cores, etc. Sua principal característica, no entanto, é que essa tag representa um elemento do tipo bloco, ou seja, que quando adicionado na página, automaticamente gera uma nova linha no layout (semelhante a um parágrafo), ao invés de ser alocado lateralmente nos demais componentes.

O código a seguir demonstra um uso básico das `div`s:

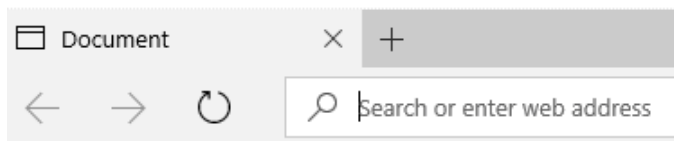
```
<input type="text" value="input 1">
<input type="text" value="input 2">
<div><input type="text" value="input 3"></div>
<div><input type="text" value="input 4"></div>
```

Perceba no resultado da imagem que, enquanto os dois primeiros `inputs` são dispostos lateralmente, os dois últimos aparecem um abaixo do outro, uma vez que estão dentro de `div`s diferentes.

Já a tag `span` é um elemento do tipo inline, ou seja, quando adicionado na página, ele é inserido lateralmente após os demais componentes, diferente das `divs` que são elementos do tipo bloco.

Elementos `span`, por padrão, também não possuem nenhuma característica visual definida, isso precisa ser feito via CSS para destacar ou aplicar uma formatação especial para um certo trecho do texto. Por exemplo, o código abaixo demonstra o uso do `span` em dois casos. No primeiro, a tag não conta com nenhum atributo adicional; no segundo, adicionamos a ela uma aparência diferenciada via CSS.

O resultado pode ser visto na imagem abaixo e, como esperado, no primeiro caso não conseguimos perceber nenhuma diferença visual devido ao uso do `span`.



Este é um texto com um trecho em destaque.

Este é um texto com um trecho em destaque.

## Exercícios De Fixação

Baseado no material desta apostila realize cada uma das tarefas a seguir:

Use as imagens em anexo nesta apostila como exemplo para recriar uma página que contenha:

- Título e subtítulo;
- Um parágrafo de texto;
- Uma imagem de sua preferência;
- Crie uma tabela com os seguintes dados:

Nome	E-mail	Idade
João	joao@gmail.com	19
Ana Júlia	anajulia@outlook.com	23
Cláudia	claudia@hotmail.com	29

- Crie um formulário com input para preencher nome e cidade;
- Crie um radio button com opções de gênero;
- Por fim, crie um botão do tipo "submit" no final do formulário.

Para um resultado melhor, poderá consultar a apostila de CSS e utilizar junto aos exercícios de HTML.



# CSS

O Cascading Style Sheets (CSS) é uma **"folha de estilo"** e é utilizada para definir a aparência em páginas da internet que adotam para o seu desenvolvimento linguagens de marcação (como XML, HTML e XHTML). O CSS define como serão exibidos os elementos contidos no código de uma página da internet.

Para criar um arquivo .css, basta abrir seu editor preferido e salvar o documento com a extensão **.css** . Nele basta colocar as regras CSS vinculadas aos elementos do documento HTML.

## Importando no HTML

---

O atributo rel define que o arquivo de destino é uma folha de estilo e href indica o endereço do arquivo .css.

```
<head>
  <meta charset="UTF-8">
  <title>Exemplo dos elementos apresentados</title>
  <link rel="stylesheet" href="estilo.css">
</head>
```

html

## Sintaxe CSS

---

A sintaxe CSS é formada por três regras fundamentais para definir um estilo:

- Seletor;
- Propriedade;
- Valor.

```
seletor {
  propriedade: valor;
}
```

CSS

O seletor vincula um elemento do documento HTML a declaração CSS. Declaração CSS é formada pela propriedade e o valor.

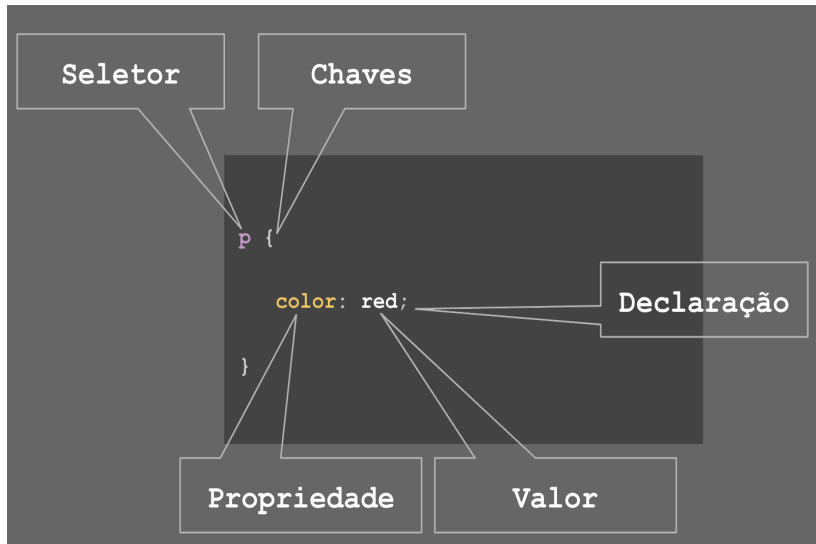
A propriedade define uma característica visual para o elemento HTML “selecionado” pelo seletor.



**Exemplo:** O texto de um parágrafo, marcado com elemento HTML `p`, possui uma propriedade de cor denominada `color`.

Já o valor define como isto vai ser atribuído à propriedade escolhida.

**Exemplo:** O valor da propriedade `color` para o elemento HTML `p` selecionado é `red` (vermelho). Ou seja, o texto do parágrafo terá uma cor vermelha.



Com esta regra qualquer `<p>` em um documento HTML, após vinculado ao arquivo CSS, receberá a cor vermelha.

**Observação:** Uma regra pode ter mais que uma declaração.

```
p {  
  font-size: 14px;  
  color: red;  
}
```

CSS

Neste caso, o parágrafo terá a fonte de tamanho 14 pixels e sua cor será vermelha.

## Seletor por classe

Este seletor possibilita o uso em mais de um elemento da mesma página. Indicado quando você precisa atribuir algumas propriedades iguais em elementos diferentes. Para construí-lo basta que você crie um nome precedido por um ponto e o chame no elemento HTML.

**Exemplo:**

No CSS:

```
.nome-da-classe {  
  color: blue;  
}
```

CSS

No HTML:

```
<title class="nome-da-classe">Document</title>
```

html

Nestes casos, o título ficará azul e o seletor pode ser usado em outros elementos que você também quer que fiquem azuis.

## Seletor por atributo

---

Este tipo de seletor associa a um atributo utilizado em um elemento HTML.

Código no HTML:

```
<input type="submit" value="Enviar">
```

html

Este é um botão para envio de dados de formulários. Podemos usar o atributo "type" com valor "submit" para estilizar o botão.

Código no CSS:

```
input [ type = "submit" ] {  
  font-weight: bold;  
}
```

CSS

## Seletor por tipo de elemento

---

O seletor "p" que usamos nos exemplos anteriores é um seletor de tipo de elemento. Esta espécie de seletor identifica e vincula um elemento do HTML, basta que para isso coloque o nome do elemento e depois ajuste suas propriedades.

**Exemplo:**

```
body {  
  propriedade: valor;  
}  
  
div {  
  propriedade: valor;  
}  
  
p, span, strong {
```

CSS

```
propriedade: valor;  
}
```

**Segue abaixo um resumo com as principais propriedades de estilo da linguagem CSS.**

- font-family: Define a família da fonte utilizada.
- font-style: Define a propriedades de estilos que podem ser: normal, italic ou oblique.
- font-size: Define o tamanho da fonte.
- line-height: Controla a altura entre as linhas do texto de um paragrafo.
- text-align: Controla o posicionamento horizontal do conteúdo de um elemento. Os valores possíveis são: left, right, center e justify.
- text-decoration: Define um efeito decorativo no texto. Podendo entre eles ser: none (sem decoração); underline (sublinhado), entre outros.
- color: Define a cor de um texto.
- width: Define o comprimento (largura) de um elemento.
- border: Define bordas para um elemento (espessura, cor).
- height: define a altura de um elemento.
- background: Define as propriedades relacionadas ao fundo de exibição.
- margin: Controla as margens de um elementos. Se forem indicados quatro valores, eles dizem respeito, respectivamente, às margens superior, direita, inferior e esquerda. Se for fornecido apenas um valor, ele é aplicado às quatro margens.
- padding: Controla os espaçamentos de um elemento. Se forem indicados quatro valores, eles dizem respeito, respectivamente, aos espaçamentos superior, direito, inferior e esquerdo. Se for fornecido apenas um valor, ele é aplicado aos quatro espaçamentos.

## Exercícios De Fixação

---

Baseado no material desta apostila realize cada uma das tarefas a seguir:

- Quais são os três estilos que o font-style tem?
- Como controlamos os espaçamentos de um elemento?
- Como definimos o posicionamento de um elemento e quais os quatro valores possíveis?
- Como definimos a altura de um elemento?
- Como definimos o tamanho da fonte?
- Como definimos a cor de uma texto?
- Como definimos bordas para um elemento?

- Como definimos o comprimento(largura) de um elemento?
- Como controlamos as margens de um elemento?
- O que usamos para definir a fonte?

# JavaScript

Além deste capítulo, você também pode visitar o tutorial em português da MDN sobre JavaScript:

<https://developer.mozilla.org/pt-BR/docs/Aprender/JavaScript>

O JavaScript (JS) é uma linguagem de programação utilizada principalmente em páginas web. Com o JS, você pode mostrar mensagens e outras informações interessantes, fazer verificações ou mudar dinamicamente a apresentação visual das páginas, conforme o comportamento que você deseja que sua página (ou aplicação) possua.

Apesar de o JavaScript ter nascido para ser utilizada nos browsers, ela mudou muito nos últimos anos, tornando-se uma linguagem muito versátil. Esta mudança se deve majoritariamente ao surgimento do Node.js, que é uma ferramenta que permite que o JavaScript seja executado fora dos browsers, da mesma maneira que outras linguagens interpretadas, como Python ou Ruby. Por conta desta versatilidade, ela pode hoje ser aplicada nos mais diversos contextos, desde a criação de front-ends modernos baseados em componentes e estados reativos, passando por aplicações desktop e IoT, até sistemas web altamente complexos que demandam diferentes níveis de disponibilidade e escalabilidade.

A sintaxe simplificada, tipagem flexível e a possibilidade de adotar conceitos de diversos paradigmas ao mesmo tempo são o que tornam JavaScript uma linguagem tão divertida e peculiar. Entretanto, sua peculiaridade abre margem para uma grande barreira de adoção, especialmente para aquelas pessoas que estão dando seus primeiros passos, seja somente em JS ou na programação como um todo. Assim como pode ser muito divertido programar usando JS e bolar soluções eficientes e com design incrível, também é muito fácil cair num temporal de confusão e código difícil de manter e entender.

## Exemplos de aplicação de JavaScript pós node

Citando apenas algumas ferramentas, se pode entender melhor como JavaScript tornou-se poderoso após o node:

### Programação em browsers

Ainda hoje se pode utilizar JavaScript da maneira como ele nasceu para ser utilizado: dentro de páginas web. Iremos focar nesta forma de uso nesta apostila, pois será a mais útil ao longo da Aceleradora. Veremos como integrar scripts com páginas HTML e algumas bibliotecas que ajudam a construir aplicações mais interativas.

### Transpiladores, bundles e bibliotecas de componentes

O código criado para ser usado em browsers não é mais o mesmo que o de cinco ou dez anos atrás. Muitas ferramentas surgiram em volta do JavaScript, dando-lhe muitos super poderes que permitem as mais diversas formas de uso. Hoje em dia podemos utilizar bibliotecas como React, por exemplo, que nos permite criar componentes de maneira mais eficiente. Podemos utilizar ferramentas de transpilação, que convertem um código com sintaxe avançada (que não é amplamente suportada pelos browsers) em um código compatível com browsers. Veja o apêndice deste capítulo para mais detalhes.

## **Ionic e React Native**

Dentre muitos outros, Ionic e React Native são ferramentas que possibilitam a criação de aplicações móveis para múltiplos sistemas operacionais (como Android e iOS, por exemplo) baseadas quase totalmente em código JavaScript.

## **Electron**

Electron possibilita a criação de aplicações Desktop baseadas em tecnologias web, tais como HTML, CSS e JavaScript. Algumas das aplicações que o utilizam são, por exemplo: Atom e Slack.

## **JhonnyFive e BENJA**

A biblioteca Jhonny Five permite o uso de JavaScript para programação de dispositivos IoT, como o Arduino. BENJA é uma distribuição Linux para nano computadores (como Raspberry Pi e Beagle Bone) com foco em aplicações do tipo Kiosk.

## **Express**

Express é a biblioteca mais popular para desenvolvimento de aplicações web com node.js. Com um design extremamente versátil e inteligente, Express permite o uso de diversos módulos para as mais diferentes necessidades de uma aplicação web, o que permite o desenvolvimento de aplicações web completas.

## **MongoDB**

MongoDB é um dos mais populares bancos de dados não relacionais e, embora não seja feito em JavaScript, ele ainda assim se baseia em vários conceitos da linguagem para estruturar informações e fornecer mecanismos de pesquisa.

# **A ECMAScript**

---

O ECMAScript é a especificação do JavaScript. Isso é dizer que ECMAScript é uma espécie de documentação onde estão as definições de o que é JavaScript e como cada browser ou plataforma devem implementar seus motores de interpretação JavaScript para que estes interpretem, bem, JavaScript, ou seja, que entenda JavaScript como a mesma coisa que todos os outros browsers

entendem. O ECMAScript é, portanto, o guia oficial seguido por cada empresa/equipe/pessoa que desenvolve um software que interprete JavaScript.

É dessa forma que os browsers possuem interpretadores de JavaScript que entendem JavaScript como sendo a mesma coisa. Na verdade, é quase isso. Alguns browsers implementam funcionalidades do JavaScript que ainda não foram implementadas em outros browsers (ou nem estão na especificação ECMAScript).

## Porque estamos falando de ECMAScript?

O ECMAScript define como o JavaScript será implementado, o que dita quais recursos as pessoas que programam em JavaScript terão de acordo com cada versão da linguagem. É importante saber que isto existe, pois boa parte do conteúdo na internet se refere a versões do ECMAScript para explicar funcionalidades do JavaScript. Por exemplo, é importante saber da existência do ECMAScript 2015 (também conhecido como ES6), que trouxe mudanças positivas drásticas à linguagem. As mudanças trazidas pelo ES6/2015 podem ser vistas aqui: <http://es6-features.org/>

Também é importante saber sobre isso para que se possa entender como escrever um código que seja compatível com os browsers disponíveis no mercado. Muitas vezes alguns browsers não entenderão a sintaxe do código JavaScript que escrevemos (pois eles podem ser muito antigos e podem não entender novos recursos da linguagem, por exemplo).

## Como eu sei o que posso ou não usar?

É preciso ler a documentação dos browsers. Uma das melhores fontes de informação à respeito dos padrões da Web é o [MDN](#), um portal criado pela Mozilla (empresa que desenvolve o navegador Firefox).

Outra ferramenta bastante útil é o Can I Use?, disponível aqui: <https://caniuse.com/>

## Como começar a usar JavaScript

---

Existem inúmeras maneiras de começar a usar o JavaScript. Vejamos como começar a usá-lo da maneira clássica, integrando-o com nossas páginas HTML diretamente.

## Inserindo o código JavaScript na página HTML

Podemos escrever código JavaScript dentro do HTML. Para isto, utilizamos a tag **script** :

Exemplo:

```
<script type="text/javascript">
  //código JavaScript
</script>
```

html

## Inserindo código JavaScript externo no HTML

A maneira anterior não é recomendada, pois os arquivos podem ficar muito extensos e difíceis de ler e manter. Sugere-se separar o código JavaScript em arquivos dependentes e linkar estes arquivos ao HTML. Exemplo:

Criamos o arquivo `meuArquivo.js` :

```
alert('Buenas');
```

js

E o linkamos ao nosso HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="meuArquivo.js"></script>
  </head>
  <body>
  </body>
</html>
```

html

## Variáveis

Em contraste ao Java, JavaScript possui tipagem dinâmica, ou seja, não é necessário definir o tipo das variáveis ao declará-las. Isto não significa que não existem tipos de dados no JavaScript, a diferença é que o JavaScript sempre tentará adivinhar os tipos de dados que estamos tentando utilizar (o que pode ser bom e ruim ao mesmo tempo).

## Tipos de dados

O JavaScript possui os seguintes tipos de dados:

### Booleano

Representam `true` (verdadeiro) ou `false` (falso).

### null

Representa a ausência de valor. É quando a variável está vazia.



## undefined

Representa a inexistência da variável. Diferentemente do **null**, **undefined** não significa que a variável está vazia, significa que ela nem sequer existe.

## Number

Representa números de qualquer espécie, sendo estes inteiros ou decimais (números quebrados).

## String

Representa texto.

## Objetos

Representa entidades mais complexas, como **Arrays**, **Expressões Regulares** ou **Objetos** propriamente ditos. Veremos um pouco mais sobre alguns destes tipos complexos mais adiante.

## Declarando variáveis

---

Como o JavaScript não possui tipos, basta declararmos nossas variáveis utilizando uma das palavras reservadas **var**, **let** ou **const**.

### var

O **var** é o jeito clássico de declarar variáveis. Esta palavra reservada existe desde as primeiras versões da linguagem:

```
var nome = 'Silvia';
var outroNome = 'Outra Silvia';
var numero = 5;
var lista = [];
var objeto = {};

function umaFuncao() {
  var x = 1;
}
```

js

### let

O **let** surgiu à partir do ES6 e foi criado para corrigir alguns problemas causados pelo **var** . Se utiliza da mesma forma:

```
let nome = 'Silvia'
let outroNome = 'Outra Silvia'
let numero = 5
let lista = []
let objeto = {}

function umaFuncao() {
  let x = 1;
}
```

js

## Diferenças importantes em relação ao **var**

### Variáveis não podem ser redeclaradas

Ao utilizar **var** , podemos redeclarar variáveis com mesmo nome:

```
var x = 1;
var x = 2;
```

js

Isto é ruim pois é confuso. Ou seja, pode ser que tenhamos nos esquecido da existência da variável **x** e queiramos declarar uma variável com este nome em outro ponto do código. Com **var** , isto será permitido e o valor da variável original será sobrescrito, causando possíveis comportamentos inesperados. O ideal nesta situação, é que o JavaScript impeça que outra variável seja declarada com o mesmo nome. Para isto, devemos utilizar **let** .

O código abaixo:

```
let x = 1;
let x = 2;
```

js

Irá gerar o seguinte erro:

```
SyntaxError: Identifier 'x' has already been declared
```

Isto não significa que o valor de uma variável já declarada não possa ser alterado:

```
let x = 1;
x = 2;
```

O código acima funcionará sem problemas.

## Controle de escopo mais eficiente

Qual será a saída do código abaixo?

```
function umaFuncao() {  
  
  if (true) {  
    var x = 1;  
    let y = 2;  
  }  
  
  console.log(x)  
  console.log(y)  
}  
  
umaFuncao()
```

js

A saída será um erro:

```
1  
console.log(y)  
          ^  
ReferenceError: y is not defined
```

As variáveis foram declarados dentro do bloco `if`. Com `var`, não existe um controle muito rígido de escopos, e as variáveis podem ser usadas fora do bloco em que foram declaradas (veja que o valor `1` foi escrito na saída do programa). O mesmo não acontece com o `let`, que controla os escopos das variáveis de maneira mais rígida. Este controle mais rígido é algo positivo, pois ajuda a prevenir comportamentos inesperados.

## const

O `const` serve para declarar constantes, variáveis cujos valores não devem ser modificados depois de atribuídos:

```
const nome = 'Silvia'  
const outroNome = 'Outra Silvia'  
const numero = 5  
const lista = []  
const objeto = {}  
  
function umaFuncao() {  
  const x = 1;  
}
```

js

## Diferenças importantes em relação e ao `var` e ao `let`

### `const` são constantes

Como mencionado anteriormente, os valores de uma variável `const` não podem ser modificados. O código a seguir:

```
const x = 1;  
x = 2;
```

js

Irá causar o seguinte erro:

```
TypeError: Assignment to constant variable.
```

Ainda que seja contraditório, constantes não são totalmente constantes. Quando declaramos uma constante que guarda um objeto, por exemplo:

```
const pessoa = {  
  nome: 'Silvia',  
  idade: 30  
}
```

js

Podemos modificar o conteúdo do objeto sem problemas:

```
pessoa.nome = 'Lima'  
  
console.log(pessoa.nome) // Lima
```

js

Isto acontece porque não estamos modificando o valor da variável, mas sim do atributo do objeto atrelado à ela.

## Arrays (listas)

---

Os arrays do JavaScript se parecem mais com listas do que vetores, pois podem para armazenar múltiplos elementos e se expandem de forma dinâmica.

Arrays são representados por um par de colchetes ( `[]` ).

### Exemplos:

Podemos ter um array vazio:

```
const umArrayVazio = []
```

js

Um array com dados:

```
const umArrayComNumeros = [1, 2, 3, 4, 5]
```

js

Um array com arrays:

```
const umArrayDeArrays = [[1], [], [3], []]
```

js

Em Javascript, não é necessário que todos os elementos de um array sejam do mesmo tipo, o que nos permite fazer coisas como:

```
const saladaDeFrutas = [  
  1,  
  '2',  
  'tres',  
  false,  
  undefined,  
  null,  
  /expressao_regular/,  
  {}  
]
```

js

Os arrays em Javascript são objetos, o que significa que eles possuem atributos e "métodos". Ou seja, podemos fazer coisas como:

## Saber o tamanho de um array

```
const arraySimples = [1, 2, 4]  
console.log(arraySimples.length) // 3
```

js

## Adicionar novos elementos no final

```
const arraySimples = [1, 2, 4]  
arraySimples.push(5)  
console.log(arraySimples) // [1, 2, 4, 5]
```

js

## Percorrer os elementos

Podemos utilizar a função **forEach** :

```
const arraySimples = [1, 2, 4]

arraySimples.forEach(function(elemento) {
  console.log(elemento)
})
```

js

Mas também podemos usar loops **for** :

### For clássico

```
const arraySimples = [1, 2, 4]

for (let i = 0; i < arraySimples.length; i++) {
  console.log(arraySimples[i]);
}
```

js

### For of (conhecido como **for each** em outras linguagens)

```
const arraySimples = [1, 2, 4]

for (let numero of arraySimples) {
  console.log(numero);
}
```

js

## Acessando elementos de um array

Para acessar elementos de um array, utilizamos a lógica de índices:

```
const arraySimples = [1, 2, 4]
const primeiroElemento = arraySimples[0]
console.log(primeiroElemento) // 1
```

js

Quando tentamos acessar um elemento que não existe, recebemos **undefined** :

```
const arraySimples = [1, 2, 4]
const vigesimoElemento = arraySimples[20]
console.log(vigesimoElemento) // undefined
```

js

Mais informações sobre arrays [↗](#).

## Funções

---

Funções são blocos que definem um comportamento. Elas são formadas por três coisas:

- Entrada (parâmetros)
- Corpo (lógica, processamento)
- Saída (retorno)

Podemos ter funções que não recebem nenhum parâmetro e/ou que não retornam nada. Para declarar funções, utilizamos a palavra reservada `function`:

```
function minhaFuncao(p1, p2) {  
  return p1 + p2;  
  //a função retorna o produto p1 e p2  
}
```

js

A função acima recebe dois parâmetros (p1 e p2) e retorna a soma dos dois parâmetros.

## Funções em variáveis

O Javascript permite tratar funções como valores (assim como números, strings, arrays, etc.), o que nos permite guardá-las em variáveis, passá-las por parâmetros ou retorná-las:

```
let minhaFuncao = function(p1, p2) {  
  return p1 + p2;  
}
```

js

## Funções que retornam funções (que loucura!)

```
let dizAlgo = function(algo) {  
  return function() {  
    return 'Digo ' + algo  
  }  
}
```

js

```
let dizOi = dizAlgo('Oi') // lembre-se que dizAlgo retorna outra funcao  
  
console.log(dizOi()) // Imprime 'Digo Oi'
```

## Passar funções por parâmetro

Lembre-se do método `forEach` dos arrays, este método recebe uma função por parâmetro, onde esta função será executada uma vez para cada item do array:

```
let total = 0;

let numeros = [1, 2, 3];

numeros.forEach(function(item) {
  console.log(item);
  total += item;
})

console.log(total);
```

O código acima irá imprimir:

```
1
2
3
6
```

## Funções simplificadas

A versão ES6 (ou superior) permite que declaremos funções de maneira simplificada, são as chamadas **arrow functions** :

```
const minhaFuncao = (p1, p2) => {
  return p1 + p2;
}
```

Perceba que não é necessário utilizar a palavra **function** , podemos substituí-la pela sintaxe de seta ( **=>** ).

### Funções em linha

Quando o corpo de uma arrow function possui apenas uma linha, podemos declará-la sem chaves:

```
const minhaFuncao = (p1, p2) => p1 + p2

const resultado = minhaFuncao(1, 1) // resultado tera o valor 2
```

Neste caso, o resultado da expressão declarada será retornada pela função. No caso acima, é como se tivéssemos a palavra **return** antes de **p1 + p2** .

Esta simplificação deixa o código muito mais simples em algumas situações. Vamos ver como exemplo **forEach** novamente:



```
const numeros = [1, 2, 3]

numeros.forEach(n => console.log(n))
```

js

Repare que não colocamos parênteses entre o parâmetro `n`. Isto somente é válido quando a função espera somente um parâmetro. A saída do código acima será:

```
1
2
3
```

## Objetos

---

Objetos são estruturas mais complexas que servem para organizar dados. Eles são estruturas associativas, onde podemos associar um valor à um nome.

### Declarando objetos

Para declarar objetos, usamos as chaves ( `{}` ):

```
let umObjeto = {};
```

js

### Para que e como usar objetos?

Objetos são extremamente úteis para modelar dados, especialmente quando precisamos manipular estruturas de dados complexas, que possuem muitas variáveis e relações. Vejamos como representar dados de uma pessoa em JavaScript:

Podemos criar variáveis separadas:

```
let nome = 'Silvia'
let idade = 23

console.log(nome) // Silvia
console.log(idade) // 23
```

js

Ou podemos agrupar estes valores de uma maneira que eles façam mais sentido e sejam mais fáceis de entender no meio do código:

```
const pessoa = {
  nome: 'Silvia',
```

js

```
    idade: 23
  }

  console.log(pessoa.nome) // Silvia
  console.log(pessoa.idade) // 23
```

De uma maneira muito simplificada, podemos pensar em objetos como sendo grupos de variáveis. Pense em como isto pode ser útil.

## Objetos com comportamento

Assim como podemos armazenar funções em variáveis ou arrays, também podemos armazená-las em um atributo de um objeto:

```
const pessoa = {
  nome: 'Silvia',
  idade: 23,
  dizOi: () => console.log('Oi')
}

pessoa.dizOi() // Ira imprimir 'Oi' na tela
```

js

Isto é extremamente útil, pois podemos pensar de uma maneira mais orientada a objetos, o que muitas vezes nos ajuda a escrever código mais organizado.

## Exercícios De Fixação

---

- Faça um programa que leia três notas de um aluno e diga se ele está aprovado ou reprovado.

Aprovado: se a média das notas for maior ou igual a 5.

Reprovado: se a média das notas for menor que 5.

A média deve ser calculada somando as três notas e dividindo o resultado por 3.

Faça o mesmo programa mostrar qual foi a maior nota do aluno. Faça o programa também mostrar qual foi a sua menor nota.

- Crie um programa que leia a idade e diga se o voto da pessoa é facultativo, obrigatório ou proibido.

Lembrando que:

Facultativo para adolescentes entre 16 e 17 anos. Obrigatório para adultos de 18 até 70 anos de idade. Após os 70 o voto se torna facultativo de novo. Menores de 16 anos: proibido.



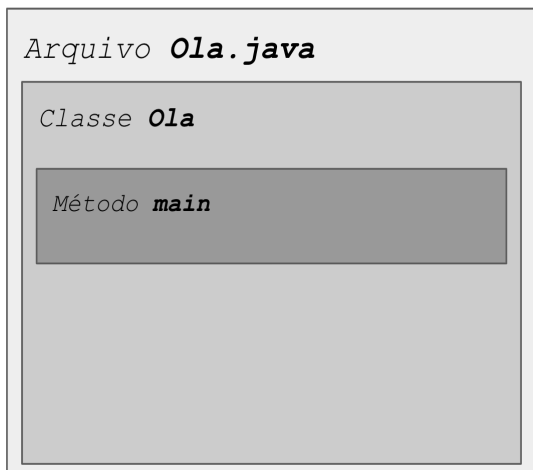
# Introdução ao Java

Neste capítulo, veremos alguns conceitos fundamentais do Java para que possamos começar a utilizar a linguagem.

## Olá, mundo

---

No Java o seu código sempre será escrito dentro de classes e métodos. O código de uma classe, por sua vez, deve ser escrito dentro de um arquivo com extensão **.java** que tenha o mesmo nome da classe que estará contida nele. Ou seja, se quero escrever uma classe **ola**, esta deve ser escrita dentro do arquivo **ola.java**:



Uma classe é apenas um bloco de código que contém atributos (variáveis) e métodos (funções).

Os atributos irão guardar dados e os métodos irão executar lógica/comportamento. Para este capítulo, isto é tudo que precisamos saber sobre classes e métodos, veremos mais sobre eles no capítulo de orientação a objetos.

Sabendo disto, vejamos então como escrever um olá, mundo em Java:

```
public class Ola {  
    public static void main(String [] args) {  
        System.out.println("Ola, mundo");  
    }  
}
```

java

No exemplo acima, criamos a classe **ola**, que possui o método **main**, o qual irá escrever uma mensagem na tela utilizando o método **System.out.println** do Java.

Existem algumas palavras chave neste exemplo que podem parecer bastante confusas ( **public** , **static** , **void** , **String[]** ). Por enquanto, não precisamos nos preocupar com elas, e iremos entender o que cada uma significa em outros capítulos.

## Comentários no código

Em alguns exemplos de código você verá o que se chamam comentários. Comentários são pedaços de texto no código que são ignorados pela linguagem de programação e suas ferramentas. Ou seja, podemos escrever qualquer coisa nestes comentários e o programa seguirá funcionando, pois eles não serão interpretados como um comando. Eles servem para documentar o código.

```
/*  
Sou um comentario de multiplas linhas.  
Posso escrever varias coisas entre as barras e asteriscos.  
Nao serei interpretado como codigo  
  
*/  
public void umMetodo() {  
  
    // Sou um comentario de uma linha, eu tambem nao serei interpretado como codigo  
  
}
```

java

## Tipos de dados

---

A linguagem Java oferece diversos tipos de dados com os quais podemos trabalhar. Há basicamente duas categorias em que se encaixam estes tipos de dados: **tipos primitivos** e **tipos de referência**. Veremos primeiro os tipos primitivos e, depois, lá no final do capítulo, um pouco sobre os tipos de referência.

### Tipos primitivos

Os tipos primitivos correspondem a dados simples escalares (que possuem um tamanho fixo na memória). No java, existem oito tipos primitivos, mas nem todos são comumente utilizados. Os tipos que você utilizará com mais frequência serão:

- **boolean**: Assume os valores booleanos **true** (verdadeiro) ou **false** (falso).
- **int**: Armazena números inteiros entre  $-2^{31}$  e  $2^{31}-1$
- **double**: Armazena números decimais (quebrados, ou com vírgula).
- **char**: O char é um tipo de variável que aceita a inserção de um caractere apenas.

```
boolean verdade = true;  
boolean mentira = false;  
  
int numero = 5;
```

java

```
double numeroQuebrado = 5.00000001;
```

```
char umCaractere = 'a';
```

## Tipos primitivos menos comuns

Além dos tipos mais comuns, ainda temos alguns outros tipos primitivos para guardar números:

- **float:** Armazena números decimais (quebrados, ou com vírgula) com uma precisão menor (menos números depois da vírgula) que o **double**.
- **short:** Armazena números inteiros entre **-32768** e **32767**
- **long:** Armazena números inteiros entre  $-2^{63}$  e  $2^{63}-1$
- **byte:** Armazena números inteiros entre **-128** e **127**

Estes tipos são muito similares aos tipos mais comuns. No entanto, eles existem para casos muito específicos, especialmente para quando precisamos economizar memória ou precisamos utilizar valores inteiros muito grandes (declarando-os como **long** em vez de **int**, por exemplo). Não estamos preocupados com estas situações neste momento.

## Links da documentação

Alguns destes são tópicos bastante avançados, mas caso queira entender um pouco mais sobre alguns detalhes, aqui estão alguns links da documentação oficial do Java:

- Sobre tipos primitivos [↗](#)
- Explicação sobre a precisão de números decimais [↗](#)
- O padrão Unicode (utilizado pelo Java para representar variáveis do tipo char) [↗](#)

## Exercícios sobre tipos

### Exercício 1:

Substitua as lacunas com os tipos que você achar mais adequados para guardar os valores das variáveis:

java

```
// Veja este exemplo:
```

```
boolean v0 = false;
```

```
// Agora preencha as lacunas para as variáveis restantes:
```

```
_____ v1 = 1;
```

```
_____ v2 = 5000;
```

```
_____ v3 = 1.00;
```

```
_____ v4 = 42000;  
  
_____ v5 = 'a';  
  
_____ v6 = '5';  
  
_____ v7 = 2147483648;  
  
_____ v8 = true;
```

## Operadores

Como o próprio nome diz, os operadores permitem executar operações sobre um ou dois **valores primitivos**.

Alguns links da documentação oficial do Java:

- [Introdução a operadores Java](#)🔗
- [Resumo sobre operadores](#)🔗

### Operador de atribuição

Pode ser que isto passe despercebido, mas ao atribuir um valor à uma variável, estamos utilizando um operador, o operador de atribuição ( = ):

```
int cinco = 5;
```

java

### Operadores de Igualdade

Os operadores de igualdade são utilizados para fazer a comparação de dois valores, ou seja, utilizamos estes operadores quando precisamos saber se um valor é **igual** , **diferente** , **maior** ou **menor** do que outro:

Nome	Sintaxe	Exemplo	Significado
Igual	==	x == y	x é igual a y
Diferente	!=	x != y	x é diferente de y
Maior que	>	x > y	x é maior que y
Menor que	<	x < y	x é menor que y
Maior ou igual	>=	x >= y	x é maior ou igual a y
Menor ou igual	<=	x <= y	x é menor ou igual a y



O uso de operadores de igualdade resulta em um valor booleano, o que permite utilizar estes operadores de diferentes maneiras:

Podemos utilizá-los diretamente dentro de estruturas condicionais:

```
if (5 > 2) {  
    System.out.println("5 eh maior que 2");  
} else {  
    System.out.println("5 nao eh maior que 2");  
}
```

java

Ou podemos guardar o resultado em uma variável, o que nos ajuda escrever código de uma maneira um pouco mais legível em algumas situações:

```
boolean cincoEhMaiorQueDois = 5 > 2;  
  
if (cincoEhMaiorQueDois) {  
    System.out.println("5 eh maior que 2");  
} else {  
    System.out.println("5 nao eh maior que 2");  
}
```

java

## Operadores Condicionais

Operadores condicionais são utilizados em valores booleanos. Eles são úteis quando precisamos verificar mais de uma condição ou precisamos inverter o valor de um booleano (trocar de **true** para **false** ou vice-versa):

### E/And (&&)

Em inglês a palavra "and" é equivalente ao "e" do português (como na frase Maria **e** João), logo, este operador verifica duas condições e resulta em verdadeiro somente se as duas forem verdadeiras, caso contrário, resulta em falso:

```
if (vaiChover == true && ehSexta == true) {  
    System.out.println("Hoje irei embora mais cedo, pois eh sexta E esta chovendo.");  
} else {  
    System.out.println("Hoje ficarei até mais tarde.");  
}
```

java

No código acima, a pessoa só iria para casa somente se fosse sexta e fosse chover.

### Ou/Or (||)

Em inglês a palavra "or" significa "ou", logo, este operador verifica duas condições e resulta em verdadeiro se pelo menos uma das duas for verdadeira, e, somente caso as duas sejam falsas, resulta em falso:

```
if (vaiChover == true || ehSexta == true) {  
    System.out.println("Hoje irei embora mais cedo, pois eh sexta ou esta chovendo.");  
} else {  
    System.out.println("Hoje ficarei até mais tarde.");  
}
```

java

No código acima, a pessoa iria para casa se fosse chover, independentemente do dia da semana. Ou, caso fosse sexta mas não estivesse chovendo, ela também iria para casa.

## Negação/Not (!)

Em inglês, "not" significa "não" ou negação. Este operador inverte o valor booleano de uma expressão ou variável:

Expressão:

```
boolean naoEhCincoNemMaiorQueDez = !(numero == 5 || numero > 10)
```

java

Variável:

```
boolean verdade = true;  
boolean mentira = !verdade;
```

java

## Exemplo de uso:

Temos que escrever um programa que valida o embarque de passageiros em um avião. O programa só deve permitir pessoas maiores de idade e que possuam passaporte. Caso a pessoa seja maior de idade mas não possua passaporte o sistema deve notificá-la. Caso a pessoa seja menor de idade, o programa deve notificá-la para estar acompanhada dos pais:

```
public void verificaEmbarque(int idade, boolean possuiPassaporte) {  
    boolean ehMaiorDeIdade = idade >= 18;  
  
    if (ehMaiorDeIdade && possuiPassaporte) {  
        System.out.println("Pode embarcar");  
    } else if (ehMaiorDeIdade && !possuiPassaporte) {  
        System.out.println("Nao pode embarcar. Apresente o passaporte.");  
    } else if (!ehMaiorDeIdade) {  
        System.out.println("Nao pode embarcar. Venha com seus pais.");  
    }  
}
```

java

## Operadores Numéricos

Os operadores numéricos servem para executar operações com números. Temos dois tipos de operadores numéricos:

### Binários

São os operadores que executam operações entre **dois** números:

Nome	Sintaxe	Exemplo	Resultado
Soma	+	1 + 1	2
Subtração	-	2 - 2	0
Multiplicação	*	2 * 2	4
Divisão	/	4 / 2	2
Módulo	%	4 % 2	0

### Exemplos de uso

Podemos utilizá-los para criar uma calculadora em Java:

```
public class Calculadora {  
    public int soma(int a, int b) {  
        return a + b;  
    }  
  
    public int subtrai(int a, int b) {  
        return a - b;  
    }  
  
    public int multiplica(int a , int b) {  
        return a * b;  
    }  
  
    public int divide(int a, int b) {  
        return a / b;  
    }  
}
```

java

### Unários

São operadores que executam operações com apenas **um** número. Estes operadores não funcionam diretamente em números literais, apenas variáveis (veja os exemplos para entender isto melhor):

Nome	Sintaxe
Incrementa	<b>++</b>
Decrementa	<b>--</b>
Acumula soma	<b>+=</b>
Acumula multiplicação	<b>*=</b>
Acumula subtração	<b>-=</b>
Acumula divisão	<b>/=</b>

### Exemplo:

```
5++; // nao funciona
```

java

```
int numero = 4;
numero++; // numero agora tem o valor 5
numero--; // numero agora tem o valor 4
numero += 2; // numero agora tem o valor 6
numero -= 2; // numero agora tem o valor 4
numero *= 2; // numero agora tem o valor 8
numero /= 2; // numero agora tem o valor 4
```

## Exercícios sobre Operadores

### Exercício 4:

## Estruturas condicionais

As estruturas condicionais permitem com que tomemos decisões nos nossos programas. Como o próprio nome indica, elas nos permitem validar se alguma condição é verdadeira ou falsa e nos permitem decidir o que fazer em cada situação (o que fazer caso uma condição seja verdadeira e o que fazer caso uma situação seja falsa).

Estas estruturas foram criadas para imitar a maneira como o idioma inglês (ou português, neste caso) funciona. Ou seja, no nosso dia a dia, expressamos condições e decisões o tempo todo de maneira natural.

### Se (if) senão se (else if) e senão (else)

Vejamos um exemplo de como expressamos condições e decisões usando o bom e charmoso português:

**Se estiver chovendo, ficarei em casa.**

**Senão, irei até a lancheria do Zico comprar um dogão.**

Na frase acima, expressamos uma condição:

- **Se estiver chovendo**

E duas decisões:

- **Ficarei em casa**
- **irei até a lancheria do Zico comprar um dogão**

Vamos mudar um pouco a estrutura da frase acima para tentar deixar um pouco mais parecido com a maneira do Java expressar condições e decisões:

```
se (estiver chovendo) {  
    ficarei em casa  
} senao {  
    irei até a lancheria do Zico comprar um dogão  
}
```

## Exemplo de código

Pensando em código, vejamos como utilizaríamos as estruturas de **se** e **senão** (as quais se chamam **if** e **else** no Java):

```
if (estaChovendo == true) {  
    System.out.println("ficarei em casa");  
} else {  
    System.out.println("irei até a lancheria do Zico comprar um dogão")  
}
```

java

## O Senão se (else if)

O Java possui a estrutura **else if**, para quando precisamos encadear mais de uma verificação de condição:

**Se for dia de pagamento, comprarei um dogão**

**Senão se for fim de semana, descansarei em casa**

**Senão, seguirei trabalhando para poder comprar um dogão**

## Exemplo de código

java

```
if (ehDiaDePagamento == true) {  
    System.out.println("Eh dia de dogao");  
} else if (ehFimDeSemana == true) {  
    System.out.println("Nao eh dia de dogao, eh dia de descanso");  
} else {  
    System.out.println("Nao eh dia de dogao, muito menos de descanso");  
}
```

Caso a condição anterior seja falsa, a condição do else if será verificada e caso, seja verdadeira, será executado. Caso contrário, a ação contida no else será executada.

É possível encadear inúmeros **else ifs** em um bloco:

java

```
if (condicao) {  
    // Acao  
} else if (condicao2) {  
    // Outra Acao  
} else if (condicao3) {  
    // Mais Outra Acao  
} else if (condicao4) {  
    // Ainda Mais Outra Acao  
} else {  
    // Nada deu certo, outra acao  
}
```

## Switch

O switch é muito útil para quando precisamos tomar muitas decisões com base em muitas condições. Se pode utilizar um monte de ifs encadeados para fazer a mesma coisa que o switch, mas ele é muito mais legível.

Vamos pensar em condições da vida cotidiana:

**Caso seja Segunda, levarei o lixo para fora.**

**Caso seja Terça, lavarei a roupa.**

**Caso seja Quarta, passarei pano no chão.**

**Caso seja Quinta, dobrarei roupas.**

Caso seja Sexta, farei comida.

Se não for nenhum dos dias anteriores, comerei um dogão, pois ninguém é de ferro.

Para expressar isto em forma de if encadeados, teríamos mais ou menos o seguinte:

java

```
if (dia == SEGUNDA) {
    System.out.println("Levarei o lixo para fora");
} else if (dia == TERCA) {
    System.out.println("Lavarei a roupa");
} else if (dia == QUARTA) {
    System.out.println("Passarei pano no chao");
} else if (dia == QUINTA) {
    System.out.println("Dobrarei roupas");
} else if (dia == SEXTA) {
    System.out.println("Farei comida");
} else {
    System.out.println("Comerei um dogão");
}
```

O mesmo código, representado em um switch seria o seguinte:

java

```
switch(dia) {
    case SEGUNDA:
        System.out.println("Levarei o lixo para fora");
        break;

    case TERCA:
        System.out.println("Lavarei a roupa");
        break;

    case QUARTA:
        System.out.println("Passarei pano no chao");
        break;

    case QUINTA:
        System.out.println("Dobrarei roupas");
        break;

    case SEXTA:
        System.out.println("Farei comida");
        break;

    default:
        System.out.println("Comerei um dogão");
        break;
}
```

# Estruturas de repetição

---

## Enquanto (while)

Assim como as estruturas de condição, o enquanto (ou **while** ) também funciona verificando uma condição. A diferença é que o enquanto é uma estrutura de repetição, ou seja, ele repetirá o código enquanto aquela condição for verdadeira.

Vamos pensar em mais um exemplo da vida cotidiana:

Um dogão triplo custa 10 reais.

Eu tenho 0 reais na minha poupança.

Enquanto a quantia da poupança for menor que o preço do dogão, eu guardo um real

Depois de ter guardado dinheiro suficiente, eu comprarei o dogão

Vamos reformular para o exemplo acima do jeito Java:

```
int precoDoDogao = 10;

int valorNaPoupanca = 0;

while (valorNaPoupanca < precoDoDogao) {
    valorNaPoupanca++;
    System.out.println("Valor na poupanca: " + valorNaPoupanca);
}

System.out.println("Tenho dinheiro para um dogao triplo");
```

java

Ao executar este código, a saída é a seguinte:

```
Valor na poupanca: 1
Valor na poupanca: 2
Valor na poupanca: 3
Valor na poupanca: 4
Valor na poupanca: 5
Valor na poupanca: 6
Valor na poupanca: 7
Valor na poupanca: 8
Valor na poupanca: 9
```



**Valor na poupança: 10**  
**Tenho dinheiro para um dogão triplo**

Ou seja, o código dentro do **while** foi repetido até a condição ( **valorNaPoupanca < precoDoDogao** ) deixar de ser verdadeira.

Cada vez que o código se repetia, ele fazia duas coisas:

- Incrementa **1** ao **valorNaPoupanca**
- Imprime o valor atual de **valorNaPoupanca**

Após dez repetições (e consequentemente dez incrementos ao valor de **valorNaPoupanca** ), o valor da variável **valorNaPoupanca** passou a ser **10** , que é igual ao valor de **precoDoDogao** . A partir deste momento, a condição **valorNaPoupanca < precoDoDogao** é falsa, pois os dois valores são iguais.

## Para (for)

A estrutura **for** é similar ao **while** , mas se organiza de uma maneira um pouco diferente.

Embora possa funcionar de outras maneiras (que não nos interessam agora), o loop **for** foi pensado para funcionar junto com um **incrementador** :

uma variável inteira, que geralmente começa com valor 0 e tem seu valor aumentado (geralmente de 1 em 1, mas pode ser de outras formas).

Um laço for convencional é construído à partir de três coisas:

- Declaração do incrementador
- Condição de execução
- Incremento

### Declaração do incrementador

É quando declaramos a variável que servirá de incrementadora. Nada mais do que isso.

### Condição de execução

Definimos uma condição que será verificada pela estrutura a cada repetição (ou iteração). Assim como no loop **while** , enquanto esta condição for verdadeira, a repetição seguirá acontecendo, quando ela tornar-se falsa, a repetição para de acontecer.

### Incremento

Definimos como a variável incrementadora terá seu valor aumentado após cada repetição. Podemos dizer, por exemplo, que o valor aumentará de 1 em 1 ( **incrementadora++** ) ou de 2 em 2 ( **incrementadora += 2** ). Este incremento acontece sempre ao final de cada repetição.

### Sintaxe do for

java

```
for (incrementador; condicao; incremento) {  
    // Codigo a ser repetido  
}
```

## Exemplos de código

Vejamos como calcular a tabuada do 5 usando o for:

java

```
System.out.println("Tabuada do 5:");  
  
for (int i = 0; i <= 10; i++) {  
    System.out.println(5 * i);  
}
```

Saída:

```
Tabuada do 5:  
0  
5  
10  
15  
20  
25  
30  
35  
40  
45  
50
```

## O dogão

Podemos resolver o problema que resolvemos com o `while` (de guardar a grana do dogão) também com o `for` :

java

```
int valorDoDogao = 10;  
  
for (int poupanca = 0; poupanca < valorDoDogao; poupanca++) {  
    System.out.println("Valor na poupanca: " + valorNaPoupanca);  
}  
  
System.out.println("Tenho dinheiro para um dogao triplo");
```

## Meio confuso?

O **for** pode parecer meio confuso agora, mas ele será extremamente útil para resolver vários problemas, especialmente quando começarmos a lidar com vetores e listas (situações para as quais o **for** foi especialmente desenhado). Não se preocupe, tudo ficará mais concreto nos próximos capítulos.

## Tipos de referência

---

Lá no começo do capítulo havíamos comentado sobre tipos de variáveis: **primitivas** e **por referência**. Deixamos as de referência por último pois era importante visitar alguns conceitos antes. Vamos tentar entender um pouco mais sobre as variáveis de referência.

Os tipos de referência armazenam objetos. Neste momento, não faz sentido tentarmos entender a fundo o que isto significa. Sugerimos que depois de você dar uma lida no capítulo de orientação a objetos, revise esta parte da apostila para entender um pouco melhor.

Nas primeiras interações com a linguagem, você irá utilizar quase que constantemente dois tipos de referências:

- Arrays (vetores)
- Strings

Portanto, neste momento, vamos nos concentrar em entender estes tipos primeiro.

## Arrays (vetores)

---

Arrays (ou vetores) são uma estrutura utilizada para quando precisamos armazenar um conjunto de valores em uma variável, como uma lista.

### Por que utilizamos vetores?

Pensemos em um caso de uso. Estamos escrevendo um programa que armazena 10 valores aleatórios inteiros na memória, multiplica cada um por 2 e exibe os resultados na tela. Podemos resolver isto de duas maneiras:

### Criando 10 variáveis

```
int valor0 = 5;
int valor1 = 11;
int valor2 = 8;
int valor3 = 13;
int valor4 = 18;
int valor5 = 20;
int valor6 = 30;
int valor7 = 35;
int valor8 = 2;
```

java

```
int valor9 = 4;
```

```
System.out.println(valor0 * 2);  
System.out.println(valor1 * 2);  
System.out.println(valor2 * 2);  
System.out.println(valor3 * 2);  
System.out.println(valor4 * 2);  
System.out.println(valor5 * 2);  
System.out.println(valor6 * 2);  
System.out.println(valor7 * 2);  
System.out.println(valor8 * 2);  
System.out.println(valor9 * 2);
```

Esta alternativa não é um bom caminho, pois nosso código ficaria imenso e confuso. Imagine ter que fazer isto com 100 variáveis?

Vejamos a outra alternativa:

## Criando um vetor de 10 posições

Antes de mais nada, para criar um vetor de valores inteiros, fazemos o seguinte:

```
int[] valores = new int[10];
```

java

Utilizamos os colchetes `[]` para indicar que a variável será um array de inteiros. Para criar um array, precisamos dizer qual será seu tamanho, ou em outras palavras, quantos elementos ele poderá guardar. Neste caso, estamos dizendo que o array poderá guardar **10** elementos.

### Precisamos falar sobre índices

Como mencionando anteriormente, arrays são uma lista de valores e, para guardar ou acessar elementos desta lista, utilizamos um índice, que indica em qual posição da lista está o elemento que queremos acessar.

Ao criar um array de inteiros de tamanho 10, inicialmente, todas as suas 10 posições conterão o valor **0** :

```
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

java

Vamos adicionar o valor **15** à segunda posição do array. Os índices começam em **0** , portanto, para acessar a segunda posição, utilizamos o índice **1** :

```
valores[1] = 15;
```

java

Ao executar o código acima, o valor do nosso array agora será:

```
{ 0, 15, 0, 0, 0, 0, 0, 0, 0, 0 }
```

java

### De volta ao problema

Para resolver o nosso problema utilizando arrays, basta adicionar os números em cada posição do array:

```
valores[0] = 5;  
valores[1] = 11;
```

java

```
valores[2] = 8;
valores[3] = 13;
valores[4] = 18;
valores[5] = 20;
valores[6] = 30;
valores[7] = 35;
valores[8] = 2;
valores[9] = 4;
```

## Segura o cavaco!

Parando para pensar, até agora não ganhamos muita coisa ao utilizar um array em vez de 10 variáveis, nosso código está ficando muito parecido e até um pouco mais complexo que a primeira alternativa (de criarmos dez variáveis).

## Qual é a moral de usar esse negócio então?

Uma das maiores vantagens de utilizar um array em vez de várias variáveis é que eles nos permitem utilizar **estruturas de repetição!**

Vamos resolver nosso problema utilizando um loop **for** :

```
valores[0] = 5;
valores[1] = 11;
valores[2] = 8;
valores[3] = 13;
valores[4] = 18;
valores[5] = 20;
valores[6] = 30;
valores[7] = 35;
valores[8] = 2;
valores[9] = 4;

for (int i = 0; i < valores.length; i++) {
    System.out.println(valores[i] * 2);
}
```

java

*"Arrays existem por bons motivos, fique de olho em onde você poderá utilizá-los."*

Regina Casé

## Bônus: Definindo valores direto na declaração

No caso do nosso problema, os arrays nos dão uma outra vantagem bacana. Podemos criar o array já preenchido com os valores que precisamos, o que simplifica bastante nosso código:

```
int [] valores = { 5, 11, 8, 13, 18, 20, 30, 35, 2, 4 };

for (int i = 0; i < valores.length; i++) {
```

java

```
System.out.println(valores[i] * 2);  
}
```

## Exercícios sobre vetores

### Exercício 2

Complete o código abaixo para criar um novo vetor vazio de numeros inteiros e tamanho 10:

```
int [] vetor = _____;
```

java

Complete o código abaixo criando um vetor de inteiros com os valores pre-determinados **1, 3, 5 e 7** :

```
int [] vetor = _____;
```

java

No código abaixo:

```
int [] vetor = {2, 4, 6, 8};
```

java

```
int a = vetor[1];
```

Qual será o valor da variável **a** ?

## Strings

---

String é uma das classes mais importantes do Java, sendo vastamente utilizada. Strings servem para representar e manipular texto.

Para declarar uma String, basta fazer o seguinte:

```
String dia = "Sexta";
```

java

Ao começar no Java, muitas pessoas pensam que String é um tipo primitivo, o que não é verdade, pois ela é uma classe e valores String armazenados em variáveis são do tipo referência. Esta confusão geralmente acontece pois ela é a única classe na linguagem que possui uma **representação literal**, ou seja, é possível criar novas Strings utilizando aspas duplas.

## Manipulando Strings

Strings são úteis para resolver incontáveis tipos de problemas, por isso, é interessante revisar como manipulamos valores String utilizando seus métodos. Vejamos alguns métodos úteis:

## Saber o tamanho de uma String

Usa-se o método **length**. Nome em português:

length: tamanho

Retorna o tamanho da String. O tamanho é a quantidade de caracteres que a String possui:

```
String texto = "Oi";

int tamanho = texto.length(); // tamanho sera 2;
```

java

## Comparar duas Strings

Usa-se o método **equals**. Nome em português:

equals: igual ou é igual a

Compara duas strings e retorna verdadeiro ( **true** ) caso elas sejam iguais:

```
String texto = "Oi";
String outroTexto = "Oi";

boolean saoIguais = texto.equals(outroTexto); //saoIguais sera true
```

java

ou falso ( **false** ) caso elas sejam diferentes:

```
String texto = "Oi";
String outroTexto = "Opa";

boolean saoIguais = texto.equals(outroTexto); //saoIguais sera false
```

java

## Substituir parte do conteúdo de uma String

Usa-se o método **replace**. Nome em português:

replace: substituir

Este método recebe dois argumentos:

- **conteudoAntigo**
- **conteudoNovo**



Ele retorna uma nova String substituindo todas as ocorrências do valor **conteudoAntigo** encontradas na String por **conteudoNovo** :

```
String bomDia = "Bom Dia!";

String boaNoite = bomDia.replace("Dia", "Noite"); // boaNoite sera "Boa Noite!"
```

java

## Obter um pedaço da string

Usa-se o método **substring**.

Este método recebe dois argumentos:

- **indiceInicial**
- **indiceFinal**

Retorna uma nova String contendo a porção que está entre as posições **indiceInicial** e **indiceFinal** :

```
String texto = "Aceleradora";

String pedaco = texto.substring(0, 3); // pedaco sera "Ace"
```

java

## Separar uma String em vários pedaços

Usa-se o método **split**. Nome em português:

split: separar

Este método recebe um argumento:

- **token**

Retorna um array de Strings, formado pela divisão da String original. Este método irá dividir a String cada vez que encontrar o **token** no conteúdo da String:

```
String texto = "A,B,C";

String[] pedacos = texto.split(","); // pedacos sera um array contendo {"A", "B", "C"}
```

java

Dica, caso queira transformar uma String em um array de Strings, utilize este método passando uma String vazia:

```
String texto = "dica";

String[] pedacos = texto.split(""); //pedacos sera um array contendo {"d", "i", "c", "a"}
```

## Verificar se a String contém um conteúdo específico

Usa-se o método **contains**. Nome em português:

contains: contém

Este método recebe um argumento:

- busca

Retorna verdadeiro caso a String contenha o valor especificado na **busca** ou falso caso contrário.

```
String texto = "Aceleradora";

texto.contains("A"); // sera verdadeiro
texto.contains("B"); // sera falso
texto.contains("Ace"); // sera verdadeiro
texto.contains("radora"); // sera verdadeiro
```

## Converter tudo para minúsculas

Usa-se o método **toLowerCase**. Nome em português:

toLowerCase: para minusculas

Retorna uma nova String com todas as letras maiúsculas trocadas por minúsculas:

```
String texto = "BOM DIA! 123";

String textoMinusculo = texto.toLowerCase(); // textoMinusculo sera "bom dia! 123"

String outroTexto = "Bom Dia";

String outroTextoMinusculo = texto.toLowerCase(); // outroTextoMinusculo sera "bom dia! 123"
```

## Converter tudo para maiúsculas

Usa-se o método **toUpperCase**. Nome em português:

toUpperCase: para maiusculas

Retorna uma nova String com todas as letras minúsculas trocadas por maiúsculas:

java

```
String texto = "bom dia";

String textoMaiusculo = texto.toUpperCase(); // textoMaiusculo sera "BOM DIA"

String outroTexto = "Bom Dia";

String outroTextoMaiusculo = texto.toUpperCase(); // outroTextoMaiusculo sera "BOM DIA"
```

Documentação Java:

- [Lista completa dos métodos da classe string](#) 

## Fim do capítulo

---

E isso é quase tudo que você deve saber para começar a se aventurar no Java! Este capítulo serve como um pontapé inicial, mas ainda temos muita coisa para ver! Caso você tenha interesse, dê uma lida nos apêndices, mais abaixo, que tentam explicar um pouco mais sobre algumas coisas que foram comentadas neste capítulo, mas que podem ser meio confusas neste momento.

Recomendamos revisitar o apêndice depois da leitura do capítulo de orientação a objetos e da realização de alguns exercícios.

## Exercícios de fixação

---

Olhando para o código abaixo, preencha os comentários indicando o que está acontecendo em cada uma das linhas. Logo abaixo você encontrará o gabarito, recomendamos tentar resolver o exercício antes de olhar para o gabarito:

```
//
public class IntrducaoJava {

    //
    String umAtributo;

    //
    public void executar() {

        //
        int valor = 10;

        //
        Cachorro cachorro = new Cachorro();

        //
        cachorro.brincar();

        //
        int i = 1;

        //
        while (i < valor) {
            //
            System.out.println("Iteracao: " + i);

            //
            if (i % 2 == 0) {
                //
                System.out.println("Iteracao par")
            }

            //
            i++;
        }
    }
}
```

# Exercícios práticos

---

No repositório da trilha de exercícios, você encontrará alguns desafios de lógica de programação que lhe ajudarão a fixar os conceitos apresentados nesta introdução.

Acesse o repositório aqui: <https://github.com/aceleradora-TW/trilha-de-exercicios> 

## Gabaritos dos exercícios

---

### Exercício 1:

java

```
// Veja este exemplo:

boolean v0 = false;

// Agora preencha as lacunas para as variaveis restantes:

byte v1 = 1; // int, short ou long tambem sao respostas corretas. Se utiliza byte neste ca

short v2 = 5000; // int ou long tambem seriam corretoa. Short nao poderia ser utilizado ac

double v3 = 1.00; // float tambem seria correto.

int v4 = 42000; // long tambem serviria.

char v5 = 'a'; // caracteres sao representados utilizando aspas simples (')

char v6 = '5'; // apesar de cinco ser um numero, aqui ele esta representado como um caracte

long v7 = 2147483648; // Este numero nao cabe no tipo int. O compilador geraria um erro co

boolean v8 = true; // O tipo booleano representa apenas os valores logicos true (verdadeir
```

### Exercício 2

Complete o código abaixo para criar um novo vetor vazio de numeros inteiros e tamanho 10:

java

```
int [] vetor = new int [10];
```

Complete o código abaixo criando um vetor de inteiros com os valores pre-determinados **1, 3, 5 e 7** :

java

```
int [] vetor = {1, 3, 5, 7};
```

Outra forma de resolver esta questão, seria:

```
int [] vetor = new int[4];
vetor[0] = 1;
vetor[1] = 3;
vetor[2] = 5;
vetor[3] = 7;
```

java

No código abaixo:

```
int [] vetor = {2, 4, 6, 8};

int a = vetor[1];
```

java

Qual será o valor da variável `a` ? 4

java

```
// Declara a classe IntroducaoJava
public class IntrducaoJava {

    // declara um atributo chamado "umAtributo" do tipo String
    String umAtributo;

    // declara um metodo chamado executar que nao retorna nenhum valor (tipo de retorno void)
    public void executar() {

        // declara uma variavel inteira "valor" com valor 10
        int valor = 10;

        // Declara a variavel cachorro do tipo Cachorro e atribui a ela numa nova instância de Cachorro
        Cachorro cachorro = new Cachorro();

        // Chama o metodo brincar do objeto cachorro
        cachorro.brincar();

        // declara uma variavel i do tipo int com valor 1
        int i = 1;

        // repete enquanto i for menor que valor
        while (i < valor) {
            // imprime o valor de i na tela
            System.out.println("Iteracao: " + i);

            // Verifica se i é par
            if (i % 2 == 0) {
                // Se i for par, imprime mensagem na tela
                System.out.println("Iteracao par")
            }
        }
    }
}
```

```
        // Soma 1 ao valor de i (o que se chama de incrementar)
        i++;
    }
}
```

## Apêndice

---

### Strings

#### Representação literal

Quando falamos *representação literal*, estamos nos referindo às *aspas duplas*. No Java, quando queremos criar um objeto de alguma classe, sempre temos que utilizar a palavra **new**. Vamos supor que nosso programa tem as classes **Carro** e **Papagaio**. Para criar objetos destas classes e guardá-los em variáveis, teríamos que utilizar o **new**:

```
Carro carro = new Carro();
Papagaio passaro = new Papagaio();
```

java

Seguindo esta lógica, teríamos que fazer o mesmo com a **String**, certo? Afinal, ela é uma classe! Teríamos que fazer algo como:

```
String dia = new String();
```

java

Ainda que isto seja possível, não é necessário, pois **String** é uma classe tão comumente utilizada, que o Java nos dá a facilidade de utilizar as aspas duplas em vez de **new**:

```
String dia = "Sexta";
```

java

#### Imutabilidade

Quando dizemos que as **Strings** são imutáveis, basicamente significa que o valor de uma variável **String** não pode ser alterado em algumas situações.

Isto gera bastante confusão, mas vejamos o que isto significa na prática.

Quando o código a seguir é executado:

```
String saudacao = "Bom dia";

saudacao.substring(0, 3);
```

java

Qual será o valor da variável saudação? Continuará sendo **Bom dia** , pois os métodos da String retornam novas Strings e mantêm os valores originais inalterados, basicamente isto é que significa imutabilidade das Strings. Se quiséssemos utilizar o valores resultante de **substring** , deveríamos guardá-lo em uma outra variável:

```
String saudacao = "Bom dia";

String bom = saudacao.substring(0, 3); // tera o valor "Bom"
```

java

## Qual a diferença entre tipos primitivos e tipos de referência?

Existem grandes diferenças entre estes tipos, no entanto, para nós esta diferença ainda não é clara, pois não exploramos os conceitos de orientação a objetos. Basicamente, tipos primitivos guardam valores, enquanto tipos de referência guardam a referência para um objeto na memória. Esta ideia pode soar bastante estranha por enquanto, pois ainda não sabemos o que é um objeto.

Para saber quando uma variável é primitiva e quando ela é referência, podemos observar o uso da palavra chave **new** (exceto com as Strings). Esta palavra é responsável por criar uma instância de objeto. Em outras palavras, ela colocará os dados do objeto em memória e adicionará na variável uma referência para a posição de memória onde estão estes dados para que possamos manipulá-los (daí o nome tipo de referência).

Não se preocupe se nada disto fizer sentido agora, recapitularemos estas ideias posteriormente com mais detalhes.

Com tudo isto em mente, vejamos uns exemplos:

```
// Um tipo primitivo:
int numero = 5;

// Um tipo de referencia
Carro carro = new Carro();

/*
Strings sao a unica excessao a regra da palavra new.

Elas tambem sao um tipo por referencia, mas nao precisamos da palavra new, em vez disto,
podemos utilizar as aspas duplas para declarar uma nova String e o Java vai entender numa
*/
String dia = "Sexta";
```

java

Uma diferença muito importante entre tipos primitivos e tipos de referência é que tipos de referência, por serem objetos, possuem atributos e métodos. Ou seja, em um tipo por referência, eu posso fazer o seguinte:



```
String dia = "SEXTA";

// Chamar um metodo da String
String diaMinusculo = dia.toLowerCase();

int [] vetor = new int [5];

// Acessar um atributo do vetor
int tamanhoVetor = vetor.length;
```

Já nos tipos, primitivos, nada disto é possível, pois eles não possuem atributos nem métodos, pois variáveis primitivas apenas guardam um valor bruto.

Depois de ler o capítulo de orientação a objetos, recomendamos que vocês revise esta parte, prometemos que tudo fará um pouco mais de sentido.

# Orientação a Objetos

## Classes

No Java o seu código sempre será escrito dentro de classes e métodos. Uma classe é um elemento do código Java que utilizamos para representar objetos do mundo real. Na orientação a objetos, sempre tentamos pensar em como abstrair conceitos do mundo real dentro do código.

Vejamos por exemplo uma classe que representa um carro:

```
public class Carro {  
  
    String marca;  
    int quantidadeDePneus;  
    //Tipos De Variáveis e atributos  
  
    public Carro() {  
        //construtor sem parâmetros  
    }  
  
    public void andar() {  
        //método  
    }  
}
```

java

Na orientação a objetos (ou a até mesmo na programação em geral), sempre teremos dois elementos:

- (Representação de) Dados
- Comportamento

## Atributos

Quando se está estudando e utilizando orientação a objetos, muito ouve-se falar dos tais atributos. Estes nada mais são que variáveis que pertencem a uma classe. No caso da nossa classe carro, temos dois atributos:

- Marca
- Quantidade de pneus

Podemos informar a visibilidade da classe, que pode ser **public**, **private** ou **default**. Utilizamos a palavra reservada **class** seguida pelo nome da classe. Logo após, entre chaves, definimos os elementos a ela relacionados: atributos, construtores e métodos.

## Construtores

### *Para que servem?*

Métodos construtores servem para construir um objeto da classe. Ao contrário de outros métodos, um construtor não pode ser chamado diretamente. Para isso usamos a palavra **new** para criar o objeto e então atribuí-lo a uma variável de mesmo tipo.

### *Exemplo de instanciação de classe:*

Chama-se instância de uma classe, a criação um objeto (através do método construtor) cujo comportamento e estado são definidos pela classe.

```
Carro carrinho = new Carro();
```

java

## Extends

Quando uma classe precisa herdar características de outra, fazemos uso de herança. Em Java, é representado pela palavra-chave **extends**. Todos os atributos e métodos não-privados serão herdados pela outra classe. Por isso, é comum dizer que a classe herdada é pai da classe que herdou seus elementos.

**Nota:** Em Java não existe herança múltipla. Assim, uma classe pode herdar apenas de outra.

### *Exemplo:*

```
public class Produto {  
  
    public double valorCompra;  
    public double valorVenda;  
  
    public class Computador extends Produto {  
        private String processador;  
  
    }  
}
```

java

A palavra-chave **extends** foi utilizada na declaração da classe Computador. Assim, além do atributo processador, devido à herança, a classe Computador também terá os atributos valorCompra e valorVenda, sem que seja necessário declará-los novamente, sem repetir código.

## Implements

Quando uma classe precisa implementar os métodos de uma interface, utiliza-se a palavra reservada **implements** :

### **Exemplo:**

Considerando a interface **IProduto** :

```
public interface IProduto {  
  
    double calculaFrete();  
  
}
```

java

Podemos ter a classe **Televisao** implementando-a:

```
public class Televisao implements IProduto {  
    private double peso;  
    private double altura;  
  
    @Override  
    public double calculaFrete() {  
        //código para cálculo do frete  
    }  
}
```

java

A anotação **@Override** explicita os métodos que foram codificados/sobrescritos.

**Nota:** Podemos implementar várias interfaces. Para isso, basta separá-las por vírgula.

Também é possível utilizar **extends** conjuntamente com **implements** . Trata-se de um recurso útil quando deseja-se tornar uma classe mais específica e implementar novos comportamentos definidos em interfaces.

### **Exemplo:**

```
public class ClasseFilha extends ClassePai implements NomeInterface {  
    // Atributos, construtores e métodos da ClasseFilha  
    //Métodos implementados da interface  
}
```

java

### **Regras para nomeação de classes:**

- Manter o nome simples e descritivo;
- Usar palavras inteiras, isto é, sem siglas e abreviações;
- A primeira letra de cada palavra devem ser maiúsculas. (camel casing)

## Constantes

Uma constante é declarada quando precisamos lidar com dados que não devem ser alterados durante a execução do programa. Para isso, utilizamos conjuntamente as palavras reservadas **final** e **static**.

### Exemplo:

```
public static final float PI = 3.14;  
public static final String MEU_NOME = "Cassia";
```

java

- A palavra **final** indica que a variável não pode ter seu valor modificado.
- A palavra **static** indica que todos os objetos de uma classe compartilharão o mesmo valor.

**Nota:** Por convenção, usamos letras maiúsculas e underscores ( \_ ) para declarar constantes e assim distingui-las das variáveis.

## Enums

Em Java, uma enum é um tipo especial de classe no qual declaramos um conjunto de valores constantes pré-definidos. usamos a palavra chave **enum** que antecede seu nome.

### Exemplo:

```
public enum Turno {  
    MANHA, TARDE, NOITE;  
}
```

java

Por serem os campos de uma enum constantes, seus nomes são escritos em letras maiúsculas.

Para atribuir um desses valores a uma variável podemos fazer como no código abaixo:

### Exemplo:

```
Turno turno = Turno.MANHA;
```

java

## Por que usar enums?

Enums são extremamente úteis quando precisamos representar um conjunto restrito de valores de uma maneira mais segura em vez de usar apenas Strings. Eles nos garantem que o compilador irá aceitar somente o conjunto de possibilidades que nós definimos, o que deixa o código menos propenso a erros. Vejamos um exemplo à respeito disso.

Precisamos fazer um programa para gerenciar as turmas de uma escola. Para isso, criamos primeiro uma classe para representar as turmas:

```
public class Turma {  
    private String nome;  
    private String turno;  
  
    public String getTurno() {  
        return turno;  
    }  
}
```

java

Uma das funcionalidades do programa é informar o horário de início das turmas de acordo com seus turnos. Para isso, implementamos o seguinte:

```
public class GestaoDeTurmas {  
  
    public void mostraHorarioDaTurma(Turma turma) {  
        if (turma.getTurno().equals("manha")) {  
            System.out.println("As aulas comecam as 7h30min");  
        } else if (turma.getTurno().equals("tarde")) {  
            System.out.println("As aulas comecam as 13h30min");  
        } else {  
            System.out.println("As aulas comecam as 18h30min");  
        }  
    }  
}
```

java

Não temos como garantir a integridade dos valores que receberemos, ou seja, pode ser que as nossas usuárias escrevam `manha` , `manhã` , `Manhã` , `De tardezinha` para representar o turno, o que causará comportamentos estranhos.

Vejamos como os enums podem ajudar a garantir um comportamento mais previsível:

Primeiro, criamos um enum para representar os turnos que o programa suporta:

```
public enum Turno {  
    MANHA, TARDE, NOITE;  
}
```

java

Depois, mudamos a nossa classe `Turma` para que ela utilize o enum:

```
public class Turma {  
    private String nome;  
    private Turno turno;  
}
```

java

```

    public Turno getTurno() {
        return turno;
    }
}

```

Agora, a **GestaoDeTurmas** pode ser um pouco mais precisa, utilizando somente os tipos que o programa aceita e informando o usuário caso a informação recebida seja inválida:

```

public class GestaoDeTurmas {

    public void mostraHorarioDaTurma(Turma turma) {
        if (turma.getTurno() == Turno.MANHÃ) {
            System.out.println("As aulas começam as 7h30min");
        } else if (turma.getTurno() == Turno.TARDE) {
            System.out.println("As aulas começam as 13h30min");
        } else {
            System.out.println("As aulas começam as 18h30min");
        }
    }
}

```

java

## Os 4 pilares da Programação Orientada a Objetos

### Abstração

É utilizada para a definição de entidades do mundo real. Sendo onde são criadas as classes. Essas entidades são consideradas tudo que é real, tendo como consideração as suas características e ações.

Entidade	Características	Ações
Carro, Moto	tamanho, cor, peso, altura	acelerar, parar, ligar, desligar
Elevador	tamanho, peso máximo	subir, descer, escolher andar
Conta Banco	saldo, limite, número	depositar, sacar, ver extrato

### Encapsulamento

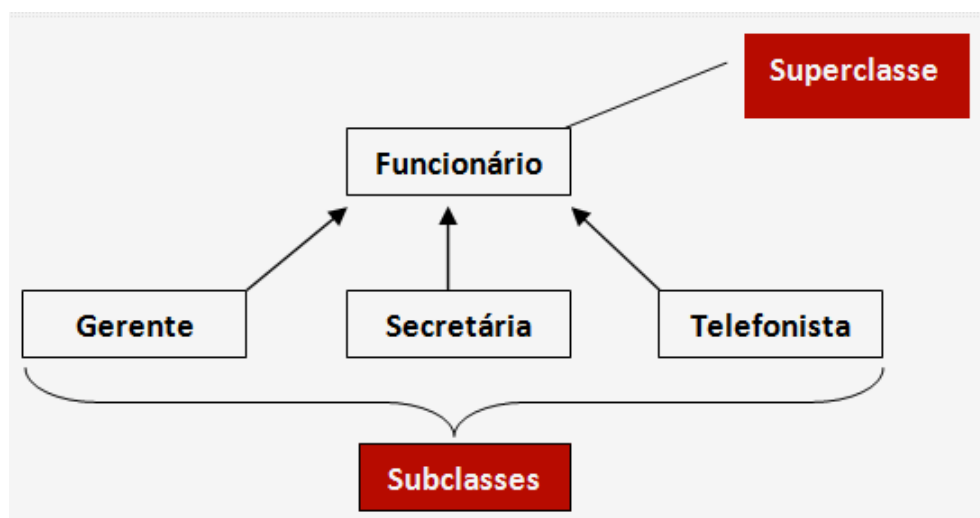
É a técnica utilizada para esconder uma ideia, ou seja, não expor detalhes internos para o usuário, tornando partes do sistema mais independentes possível. Por exemplo, quando um controle remoto estraga apenas é trocado ou consertado o controle e não a televisão inteira. Nesse exemplo do controle remoto, acontece a forma clássica de encapsulamento, pois quando o usuário muda de canal não se sabe que programação acontece entre a televisão e o controle para efetuar tal ação.

 Explicação

### Herança

Na Programação Orientada a Objetos o significado de herança tem o mesmo significado para o mundo real. Assim como um filho pode herdar alguma característica do pai, na Orientação a Objetos é permitido

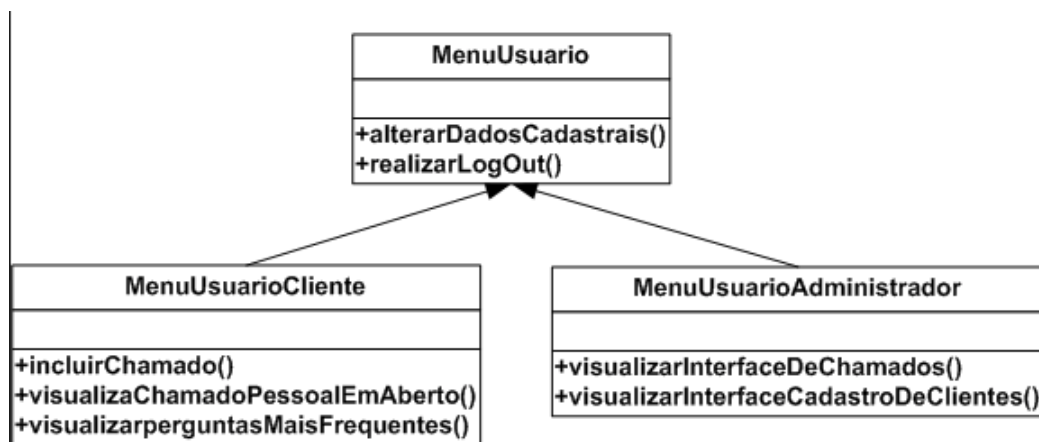
que uma classe possa herdar atributos e métodos da outra, tendo apenas uma restrição para a herança. Os modificadores de acessos das classes, métodos e atributos só podem estar com visibilidade **public** e **protected** para que sejam herdados.



## Polimorfismo

O polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

*Exemplo:*



## EXERCÍCIOS DE FIXAÇÃO

Clone o projeto do GitHub link: <https://github.com/aceleradora-TW/laboratorio-oo-java>

Faça os exercícios e deixe os testes passar

**Divirta-se!**



# Começando com o PostgreSQL

São um conjunto de arquivos relacionados entre si com registros sobre pessoas, lugares ou coisas, são coleções organizadas de dados. Sempre que for possível agrupar informações que se relacionam e tratam de um mesmo assunto, posso dizer que tenho um banco de dados.

Já um sistema de gerenciamento de banco de dados (SGBD) é um software que possui recursos capazes de manipular as informações do banco de dados e interagir com o usuário. Um exemplo de SGBD é:



É um Sistema Gerenciador de Bancos de dados Relacional estendido e de código aberto(SGBDR- o R é porque ele é relacional). Existem vários Modelos de Base de Dados, alguns exemplos são: Modelo em Rede, Modelo Hierárquico, Modelo Relacional, Orientado a Objetos.

Para criar a base de dados o SGBD utiliza uma linguagem. A mais utilizada atualmente é o SQL, (Structured Query Language). Para armazenar um dado em um banco de dados, é necessário criar tabelas e dentro delas são criadas colunas, onde as informações são armazenadas.

```
sudo apt update
```

sh

Quando vamos instalar algum programa via terminal, é sempre uma boa ideia executar antes **sudo apt update** , pois ele irá atualizar os links para os repositórios de onde o Linux faz download dos programas (como se fossem links para o Baixaki que o Linux usa internamente para encontrar os instaladores dos programas).

```
sudo apt install postgresql postgresql-contrib
```

sh

Esse comando instala o pacote Postgres junto com o pacote contrib, que adiciona alguns utilitários e funcionalidades adicionais.

```
sudo -i -u postgres
```

sh

Alternando para a conta postgres.

# Comandos SQL (utilizando PostgreSQL)

---

Antes de executar qualquer comando **SQL** , precisamos acessar o Postgres. Para isso, executamos:

```
psql
```

sh

Isso fará você entrar no prompt do PostgreSQL e, a partir daqui, você estará livre para interagir com o sistema de gerenciamento de banco de dados imediatamente (consultando e alterando dados de tabelas e bancos).

Para sair do prompt do Postgres e voltar ao terminal comum, digite:

```
\q
```

sh

## Agora sim, SQL

```
CREATE TABLE pessoa (  
  id serial primary key,  
  nome varchar(255));
```

O comando CREATE TABLE cria uma tabela, pessoa é um exemplo de nome para sua tabela e dentro dos () vai as suas colunas, como exemplos temos id e nome.

```
INSERT INTO TABLE pessoa VALUES (1, "Ingrid");
```

O comando INSERT INTO TABLE insere na tabela pessoa respectivamente os seguintes valores: id=1 e nome=Ingrid.

```
SELECT * FROM pessoa;
```

Seleciona toda a tabela pessoa.

```
UPDATE pessoa SET nome = 'Brenda' WHERE id = 1;
```

Atualiza o nome do cliente para Brenda se o Id for igual a 1

```
DELETE FROM pessoa WHERE id = 1;
```

Exclui as linhas onde o id é igual a 1 na tabela especificada. Se o id não for especificado, o efeito é excluir todas as linhas da tabela.

## Ferramentas Adicionais

---

### PgAdmin 3

Para uma interface gráfica de usuário do PostgreSQL, use o seguinte comando:

```
sudo apt-get install pgadmin3
```

sh

```
sudo su postgres -c psql postgres
```

sh

```
ALTER USER postgres WITH PASSWORD 'postgres';
```

sh

### Execute os seguintes comandos

```
CREATE TABLE cliente(id_cliente INTEGER, nome_cliente VARCHAR (255), CONSTRAINT pk_id_cliente
```

```
CREATE TABLE pedido(id_pedido INTEGER, total REAL, cliente_id_cliente INTEGER, FOREIGN KE
```

```
INSERT INTO cliente (id_cliente,nome_cliente) VALUES (2334, 'Joao da silva');
```

```
INSERT INTO cliente (id_cliente,nome_cliente) VALUES (3456, 'Ana Maria Braga');
```

```
INSERT INTO cliente (id_cliente,nome_cliente) VALUES (8275, 'Joana Barcelos Veiga');
```

```
INSERT INTO cliente (id_cliente,nome_cliente) VALUES (9812, 'Carlos Schallenger');
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (752, 100.23, 2334);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (334, 1456.00, 2334);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (498, 278.98, 9812);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (125, 874.98, 9812);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (365, 286.30, 9812);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (775, 134.54, 9812);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (834, 187.34, 3456);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (998, 234.34, 8275);
```

```
INSERT INTO pedido(id_pedido,total,cliente_id_cliente) VALUES (101, 456.87, 8275);
```

```
SELECT * FROM cliente;
```

```
SELECT * FROM pedido;
```

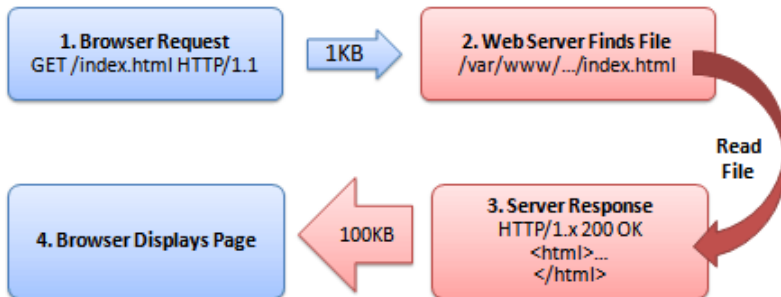
```
UPDATE cliente SET nome_cliente = 'Ronaldo';
```

```
SELECT * FROM cliente;
```

```
DELETE FROM cliente WHERE id_cliente = 8275;
```

# O HTTP

## HTTP Request and Response



O protocolo HTTP pode ser entendido como **a fundação da internet**, pois ele permite que se envie informações de arquivos HTML, CSS e JavaScript de um computador para o outro, o que é fundamental para a vida das páginas web (a forma principal de comunicação utilizada na internet).

Isso quer dizer que, quando alguém acessa um site, para poder vê-lo funcionando num browser é preciso antes receber os dados contendo as definições HTML, CSS e JavaScript deste site através do HTTP.

Evidentemente, não basta apenas receber tais dados, eles precisam ser interpretados para que façam sentido e se tornem uma página bonita e funcional. Essa é mais uma tarefa para os browsers: interpretar os dados recebidos por HTTP e transformá-los em páginas utilizáveis por seres humanos.

## Browser e HTTP passo a passo

Então, observando o que acontece quando uma pessoa acessa um site qualquer, podemos ver tudo que um browser faz por nós:

### Comunicação HTTP:

- Alguém acessa o site no endereço *www.google.com.br*
- O browser faz uma **requisição HTTP** para o servidor que está no endereço *www.google.com.br*
  - Em outras palavras, o browser diz, falando em HTTP com o servidor: *"Por favor, me passe os dados da página"*
- O servidor, de maneira formal, responde ao browser com uma **resposta HTTP** contendo as informações pedidas de forma organizada e padronizada, de um jeito que browser consegue entender o que foi recebido

### Renderização de uma página:

- O browser recebe o conteúdo HTML, CSS e JS que foi enviado pelo servidor através do HTTP
- O browser começa a interpretar tal conteúdo, transformando-o em uma página

- o Baseada na estrutura descrita pelo HTML
- o Nos estilos descritos nos CSS
- o E em tudo mais que estiver nos JavaScripts

## Como se parece uma requisição HTTP?

No exemplo anterior, o browser fez uma requisição para o site **www.google.com.br**. Na prática, ele enviou através de uma comunicação HTTP, um texto mais ou menos como este:

```
GET HTTP/1.1
Host: www.google.com.br
Cache-Control: no-cache
```

E o servidor da Google, por sua vez, respondeu com algo mais ou menos assim:

```
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="pt">
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
    <meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" itemprop="j
  <title>Google</title>
  <script>
    (function(){
      window.google={kEI:'2vToWbWN0c0bwQSXhKLADA', kEXPI:'1352821,1353383,1354277,1354401,1
    })();
  ...
```

Ou seja, a resposta do browser foi um documento HTML, contendo scripts JavaScript e links para outros documentos CSS e mais scripts. Esta é a informação que o browser interpretará e transformará na famosa página de busca da Google.

## HTTP na prática

Caso você queira ver essa comunicação acontecendo na prática, pode usar o comando **telnet** :

Primeiro, estabeleça a conexão com a Google via telnet:

```
telnet google.com 80
```

Após a execução deste comando, a seguinte mensagem deve aparecer:

```
Trying 172.217.4.46...
Connected to google.com.
Escape character is '^['.
```

Após esta mensagem, digite o seguinte e pressione **enter** :

```
GET /
```

sh

Depois disso, o servidor da Google irá responder com um monte de HTML (que é o que o nosso browser recebe quando navegamos até google.com) e então encerrará a conexão.

Neste último comando, existem duas informações muito importantes à respeito de como se organiza uma aplicação web:

- O verbo HTTP (no caso do comando, usamos o **GET** )
- A rota (no caso, usamos **/** )

```
GET /
```

```
^^^ ^
```

**Verbo Rota**

sh

## Verbos e rotas HTTP

---

O protocolo HTTP permite especificar um **verbo** e uma **rota** quando fazemos uma requisição para uma aplicação web. Estas duas informações nos ajudam a organizar a aplicação, separando a lógica que deve ser implementada para cada rota e verbo.

### Verbos

O HTTP possui vários verbos que são utilizados para descrever qual o tipo de operação queremos executar. Os mais relevantes geralmente são:

- **GET** : Quando indicamos que o request que estamos fazendo quer buscar informações da aplicação
- **POST**: Usamos post para enviar informações para que a aplicação as salve em algum lugar (um banco de dados, por exemplo)

### Rotas

As rotas não possuem um padrão definido (como os verbos). Fica a critério de quem está desenvolvendo a aplicação definir quais rotas ela responderá. Quando algum programa faz um request para uma rota que não é suportada pela aplicação, temos o famoso erro **404** .

## Um exemplo prático de rotas e verbos

---

Para entender melhor quando e como utilizar determinados verbos e rotas, vamos pensar em um exemplo: Um *e-commerce*.

Primeiro, vamos pensar em algumas funcionalidades comuns de um site de e-commerce:

- Lista de produtos
- Login
- Carrinho
- Busca

Agora, imaginando que estamos criando a aplicação de e-commerce, como suportaríamos estas funcionalidades? Vamos pensar somente na parte do HTTP, sem mais detalhes.

## O site

Primeiro, por se tratar de um site, vamos assumir que nossa aplicação estará hospedada no endereço `www.o-comercio.com.br`.

## A Lista de produtos

Para implementarmos uma lista de produtos, é bem provável que usaremos o método http `GET`, já que queremos buscar uma lista de produtos e exibí-la para as pessoas usuárias. Portanto, podemos ter nossa primeira rota: `/produtos`.

Um exemplo de pseudo-request seria:

```
GET www.o-comercio.com.br/produtos
```

## Login

Para que uma pessoa possa fazer login no site, ela precisa fornecer um nome de usuário e senha. Já que aplicação precisa receber informações para executar a tarefa de login, esta rota precisa ser do tipo `POST`. Podemos chamá-la de `/login`.

Um exemplo de pseudo-request seria:

```
POST www.o-comercio.com.br/login
usuario=doot&senha=tood
```

## Carrinho

Um carrinho ficará atrelado à um perfil de usuária, então cada vez que a pessoa adicionar um novo item no carrinho, ela terá que fornecer para a aplicação o produto que foi adicionado e seu nome de usuário,



para que a aplicação saiba para quem atribuir aquele produto. Podemos chamar nossa rota de `/adicionar-produto` . Como precisamos fornecer informações, esta rota terá que ser do tipo **POST** .

Um exemplo de pseudo-request seria:

```
POST www.o-comercio.com.br/adicionar-produto
usuario=doot&idProduto=123123
```

## Busca

Nossa última funcionalidade é a de busca. Onde a pessoa digita o nome de um produto e a aplicação exibe uma lista de produtos que tenham um nome parecido com o que foi digitado. Nesta rota, também temos que enviar dados para a aplicação (o nome do produto pesquisado), no entanto, *estes dados não ficarão guardados na aplicação*, eles apenas serão usados para efetuar a pesquisa, logo, esta rota pode ser do tipo **GET** , ainda que ela envie dados para o servidor. Podemos chamar esta rota de `/busca` .

Um exemplo de pseudo-request seria:

```
GET www.o-comercio.com.br/busca?nomeProduto=balde
```

Pode-se observar que esta URL possui detalhes a mais, mais especificamente, este pedaço: `? nomeProduto=balde` . Chamamos esta parte da URL de **query** e ela serve para passarmos parâmetros para um request **GET** . Em um request do tipo **POST** , os dados enviados ficam no corpo da requisição. Usamos as **queries** em requests **GET** porque requests do tipo **GET** *não possuem corpo*. Não se preocupe se isso ficar abstrato demais agora, uma vez que vejamos código, isso ficará mais claro.

Basicamente, é assim que nos comunicamos com uma aplicação via HTTP.

## Códigos de status HTTP

Os códigos HTTP servem como uma ferramenta para que saibamos o que aconteceu com a nossa requisição. Estes status sempre são retornados juntamente com a resposta. Os mais conhecidos/utilizados são

- **200**: Significa que a nossa requisição funcionou como deveria e que o que pedimos deve ter sido retornado.
- **404**: Significa que o recurso que estamos procurando não foi encontrado no servidor, por exemplo, quando pedimos uma página que não existe no servidor.
- **500**: Significa que alguma erro aconteceu no servidor e que ele não pode responder a nossa requisição.

Estes são alguns dos mais utilizados, podemos encontrar uma lista mais detalhada em:

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>

# Como desenvolver aplicações web com HTTP?

---

O HTTP é fundamental para criar aplicações web e existem diversas ferramentas em várias linguagens para facilitar o desenvolvimento deste tipo de aplicações. Na Aceleradora, vamos usar:

- Java (linguagem de programação)
- Spring Boot (framework java para desenvolvimento de aplicações)
- MVC (padrão de projeto)

Estas ferramentas e conceitos tentam abstrair o HTTP em maneiras mais práticas de processar mensagens do protocolo. Em outras palavras, o Spring (principalmente, mas não exclusivamente) resolverá vários problemas complexos para que nós possamos focar em implementar as regras de negócio da nossa aplicação em vez de investirmos muito tempo em reinventar a roda (lidar como mensagens HTTP, roteamento de requisições, interpretação e conversão de dados, etc.).

Esta apostila tenta introduzir as ideias destas ferramentas e padrões que iremos utilizar na prática.

## Exercícios de Fixação

---

Baseado no material desta apostila realize cada uma das tarefas a seguir:

1. Para que serve o método GET?
2. Para que serve o método POST?
3. Para que usamos o método DELETE?
4. Qual a semelhança entre os métodos PUT/PATCH?

# Introdução ao Spring

O Spring é um framework Java criado com o objetivo de facilitar o desenvolvimento de aplicações (de vários tipos, não somente web), oferecendo um conjunto de soluções prontas para problemas comuns. Alguns dos problemas que o Spring resolve:

- Implementação do padrão MVC
- Persistência de dados (comunicação com bancos de dados)
- Injeção de dependências
- Autenticação de usuários
- Comunicação com mecanismos de fila
- Processamento em lote (Spring Batch)

O Spring é uma ferramenta extremamente poderosa, especialmente para aplicações web. Vamos revisar abaixo como ele pode nos ajudar a construir uma aplicação web de maneira eficiente.

## O padrão MVC

---

O MVC é um padrão arquitetural muito utilizado no desenvolvimento de aplicações web.

### O que é um padrão arquitetural?

Um padrão arquitetural nada mais é do que a forma como organizamos o código da nossa aplicação. Os padrões surgem como uma maneira de impor as melhores práticas para se desenvolver um software.

#### Responsabilidade única e papéis bem definidos

É importante que cada classe faça somente uma coisa e é preciso definir estas coisas de maneira clara.

### E como se usa MVC?

Diferentemente de uma ferramenta (como o Spring), o MVC não é algo que podemos adotar diretamente, ou seja, não é possível fazer download de algo e sair utilizando. O MVC é um conjunto de conceitos e ideias, portanto, utilizar o MVC se refere a como vamos organizar o código da nossa aplicação. a sigla MVC significa **M**odelo, **V**isão e **C**ontrolador. Vamos entender o que cada coisa significa:

#### Modelo

Classes que representam entidades manipuladas pelo sistema.

## Visão

Classes, templates e recursos (HTML, CSS, JavaScript) utilizados para desenvolver a interface do sistema. Formam os mecanismos pelos quais os usuários interagem com o sistema.

## Controle

Classes que organizam o relacionamento entre o modelo e a visão.

## Inversão de Controle

---

É um padrão de desenvolvimento onde se insere determinado código da aplicação dentro do framework, que ficará responsável pelo controle da chamada dos métodos diferente da programação tradicional, ou seja, não é determinada diretamente pelo programador. Se um dia o nome da classe ou o lugar onde ela está armazenada for alterado, nós apenas alteraríamos um arquivo de configuração, não mexeríamos em uma única linha do código da classe.


## Injeção de Dependência

---

Nesta solução as dependências entre os módulos não são definidas programaticamente, mas sim pela configuração de uma infraestrutura de software (container) que é responsável por "injetar" em cada componente suas dependências declaradas. A Injeção de dependência se relaciona com o padrão Inversão de controle mas não pode ser considerada um sinônimo deste.

## Outros projetos Spring

---

- Spring MVC: para desenvolvimento de aplicações web (módulo do Spring Framework).
- Spring Security: para inserção de funcionalidades de autenticação e autorização.
- Spring Data: para aplicações que usam novas tecnologias de armazenamento de dados como bancos NoSQL e serviços na nuvem.
- Além de outros. Visite <http://spring.io/projects> 

## Spring Boot

---

O Spring Boot é uma versão **opiativa** e **customizável** do Spring. Seu objetivo é facilitar a criação de aplicações Spring autônomas e de produção (que você pode "executar"). É uma estrutura leve que simplifica a configuração de aplicações Spring e facilita a publicação de nossas aplicações. A intenção é ter o seu projeto rodando o mais rápido possível e sem complicação.

# Vantagens do Spring Boot

Em primeiro lugar o Spring Boot é opinativo, ou seja, tem opiniões. É outra forma de dizer que o Spring Boot tem padrões razoáveis; assim, é possível desenvolver um aplicativo rapidamente utilizando esses valores usados com frequência.

Por exemplo, o Tomcat é um contêiner da web muito popular. Por padrão, um aplicativo da web Spring Boot usa um contêiner Tomcat integrado.

Em segundo lugar, ele é customizável, ou seja, uma estrutura opinativa não será muito boa se não puder mudar de opinião. É possível customizar facilmente um aplicativo Spring Boot conforme suas preferências, tanto na configuração inicial quanto posteriormente, no ciclo de desenvolvimento.

Por exemplo, se você prefere o Maven, pode facilmente fazer alterações de **<dependência>** no seu arquivo **pom** para substituir o valor padrão do Spring Boot.

## Por onde iniciar?

Existem várias formas de se criar um projeto com Spring Boot. Você pode fazer “na mão”, pode-se usar o Spring Boot pela linha de comando, uma IDE ou utilizar o **Spring Initializr**.

**SPRING INITIALIZR** bootstrap your application now

Generate a Maven Project with Java and Spring Boot 2.0.2

### Project Metadata

Artifact coordinates

Group

Artifact

### Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

Generate Project alt + ↵

Caso você esteja utilizando o **IntelliJ**, ele fornece uma opção para se criar um projeto utilizando o Spring Initializr.

## Anotações

**@SpringBootApplication** Para quem usa Spring Boot, essa é uma das primeiras que você. Ela engloba a @Component, @ComponentScan e mais uma chamada @EnableAutoConfiguration, que é

utilizada pelo Spring Boot para tentar adivinhar as configurações necessárias para rodar o seu projeto.

**@Controller** Usada para classes controladoras que possuem métodos que processam Requests numa aplicação web. Um Controller é responsável tanto por receber requisições como por enviar a resposta ao usuário, algo

**@RequestMapping** Geralmente utilizada em cima dos métodos de uma classe anotada com @Controller. Serve para você colocar os endereços da sua aplicação que, quando acessados por algum cliente, deverão ser direcionados para o determinado método.

### **Exemplo:**

Veja como é simples criar um Controller, mas veja que este não possui nenhum “mapping” atrelado a ele. Então criemos uma view (.jsp) chamada “home.jsp” dentro da pasta “/WEB-INF/views” e criaremos o mapping “/home” para exibir a view criada. Veja a Listagem 2.

```
java
@Controller //Define que minha classe será uma controladora
public class HomeController {

    @RequestMapping("/home") //Define a URL que quando for requisitada ira chamar o metodo
    public ModelAndView home() {
        //Retorna a view que deve ser chamada, no caso home (home.jsp) aqui o .jsp é omitido
        return new ModelAndView("home");
    }
}
```

**@Repository** É associada com classes que isolam o acesso aos dados da sua aplicação. Comumente associada a DAO's.

**@Autowired** Anotação utilizada para marcar o ponto de injeção na sua classe. Você pode colocar ela sobre atributos ou sobre o seu construtor com argumentos. Marca um construtor, um campo, um método setter ou um método

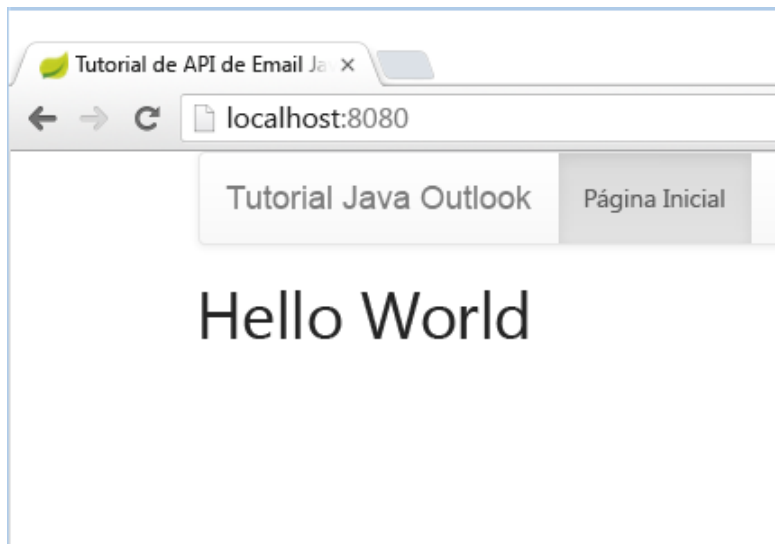
**@ResponseBody** Utilizada em métodos anotados com @RequestMapping para indicar que o retorno do método deve ser automaticamente escrito na resposta para o cliente.

### **Exemplo:**

```
java
@Controller
public class HomeController {

    @RequestMapping("/home")
    @ResponseBody // Essa anotação indica para o Spring renderizar o retorno do metodo como texto
    public String home() {
        return "<h1>Hello World</h1>";
    }
}
```

Como mostra a imagem:



**@Service** Associada com classes que representam a ideia do Service do Domain Driven Design. Para ficar menos teórico pense em classes que representam algum fluxo de negócio da sua aplicação. Por exemplo, um fluxo de finalização de compra envolve atualizar manipular o carrinho, enviar email, processar pagamento etc. Este é o típico código que temos dificuldade de saber onde vamos colocar, em geral ele pode ficar num Service.

**@Component** A annotation básica que indica que uma classe vai ser gerenciada pelo container do Spring. Todas as annotations descritas acima são, na verdade, derivadas de **@Component**. A ideia é justamente passar mais semântica.

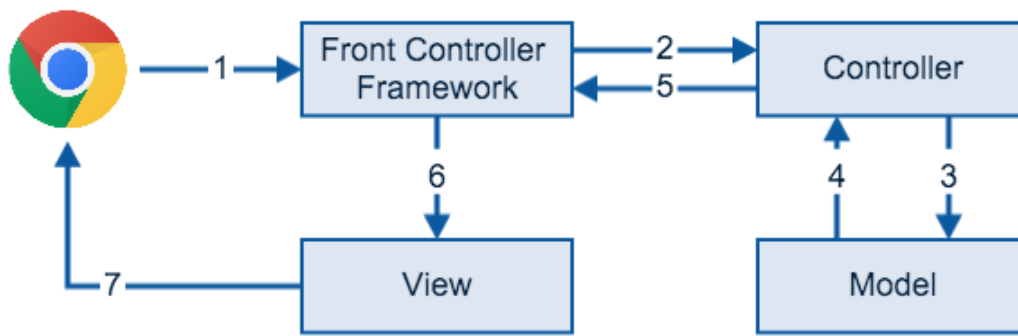
## Spring MVC

O Spring MVC é um dos frameworks para desenvolvimento Web mais utilizados hoje em dia. Com ele, temos à nossa disposição uma implementação do padrão MVC em conjunto com os principais recursos do Spring. Ele já tem todas as funcionalidades que precisamos para:

- atender as requisições HTTP;
- delegar responsabilidades de processamento de dados para outros componentes;
- preparar a resposta que precisa ser dada.

É uma excelente implementação do padrão MVC.

MVC é abreviação de Model, View e Controller, e é bacana entender o papel de cada um deles dentro do sistema, então vamos a explicação:



1. Acessamos uma URL no browser que envia a requisição HTTP para o servidor que roda a aplicação web com Spring MVC. Perceba que quem recebe a requisição é o controlador do framework, o Spring MVC.
2. O controlador do framework irá procurar qual classe é responsável por tratar essa requisição, entregando a ela os dados enviados pelo browser. Essa classe faz o papel do controller.
3. O controller passa os dados para o model, que por sua vez executa todas as regras de negócio, como cálculos, validações e acesso ao banco de dados.
4. O resultado das operações realizadas pelo model é retornado ao controller.
5. O controller retorna o nome da view, junto com os dados que ela precisa para renderizar a página.
6. O Framework encontra a view que processa os dados, transformando o resultado em um HTML.
7. Finalmente, o HTML é retornado ao browser do usuário.

Na prática, o controller é a classe Java com os métodos que tratam essas requisições. Portanto, tem acesso a toda informação relacionada a ela como parâmetros da URL, dados submetidos através de um formulário, cabeçalhos HTTP, etc.



# Hibernate

## O que é?

O Hibernate é um framework utilizado para mapear as classes java que são suas entidades de negócio para tabelas no banco relacional, economizando tempo.

## Bibliotecas do Hibernate e JPA

Vamos usar o JPA com Hibernate, ou seja, precisamos baixar os JARs no site do Hibernate. O site oficial do Hibernate é o [www.hibernate.org](http://www.hibernate.org), onde você baixa a última versão na seção ORM e Download.

Com o ZIP baixado em mãos, vamos descompactar o arquivo. Dessa pasta vamos usar todos os JARs obrigatórios (required). Não podemos esquecer o JAR da especificação JPA que se encontra na pasta jpa.

Para usar o Hibernate e JPA no seu projeto é necessário colocar todos esses JARs no classpath.

O Hibernate vai gerar o código SQL para qualquer banco de dados. Continuaremos utilizando o banco MySQL, portanto também precisamos o arquivo .jar correspondente ao driver JDBC.

## Mapeando uma classe Tarefa para nosso Banco de Dados

Para isso, continuaremos utilizando a classe que representa uma tarefa:

```
package br.com.caelum.tarefas.modelo;

public class Tarefa {
    private Long id;
    private String descricao;
    private boolean finalizado;
    private Calendar dataFinalizacao;
}
```

java

Criamos os getters e setters para manipular o objeto, mas fique atento que só devemos usar esses métodos se realmente houver necessidade.

Essa é uma classe como qualquer outra que aprendemos a escrever em Java. Precisamos configurar o Hibernate para que ele saiba da existência dessa classe e, desta forma, saiba que deve inserir uma linha na tabela Tarefa toda vez que for requisitado que um objeto desse tipo seja salvo. Em vez de usarmos o termo "configurar", falamos em mapear uma classe a tabela.

Para mapear a classe Tarefa, basta adicionar algumas poucas anotações em nosso código. Anotação é um recurso do Java que permite inserir metadados em relação a nossa classe, atributos e métodos. Essas anotações depois poderão ser lidas por frameworks e bibliotecas, para que eles tomem decisões baseadas nessas pequenas configurações.

Para essa nossa classe em particular, precisamos de apenas quatro anotações:

```
@Entity
public class Tarefa {

    @Id
    @GeneratedValue
    private Long id;
    private String descricao;
    private Boolean finalizado;

    @Temporal(TemporalType.DATE)
    private Calendar dataFinalizacao;

    // métodos...
}
```

java

**@Entity** indica que objetos dessa classe se tornem "persistível" no banco de dados.

**@Id** indica que o atributo id é nossa chave primária (você precisa ter uma chave primária em toda entidade) e **@GeneratedValue** diz que queremos que esta chave seja populada pelo banco (isto é, que seja usado um auto increment ou sequence, dependendo do banco de dados).

Com **@Temporal** configuramos como mapear um Calendar para o banco, aqui usamos apenas a data (sem hora), mas poderíamos ter usado apenas a hora (**TemporalType.TIME**) ou timestamp (**TemporalType.TIMESTAMP**). Essas anotações precisam dos devidos imports, e pertencem ao pacote `javax.persistence`.

Mas em que tabela essa classe será gravada? Em quais colunas? Que tipo de coluna? Na ausência de configurações mais específicas, o Hibernate vai usar convenções: a classe Tarefa será gravada na tabela de nome também Tarefa, e o atributo descricao em uma coluna de nome descricao também!

Se quisermos configurações diferentes das convenções, basta usarmos outras anotações, que são completamente opcionais. Por exemplo, para mapear o atributo dataFinalizacao numa coluna chamada data\_finalizado faríamos:

```
@Column (name = "data_finalizado", nullable = true)
private Calendar dataFinalizacao;
```

java

Para usar uma tabela com o nome tarefas:

```
@Entity  
@Table (name="tarefas")  
public class Tarefa
```

## EXERCÍCIOS DE FIXAÇÃO

---

### Primeiros passos para mapear uma tabela simples com hibernate

Imagine que temos o hibernate configurado, o próximo passo é realizar o mapeamento das classes às tabelas do banco de dados.

1. Qual é a anotação que usaremos para indicar que a classe será utilizada como uma entidade, ou seja, que os dados serão "persistíveis" pelo banco de dados?
2. Qual anotação é utilizada para indicar o identificador único da tabela?
3. Qual a notação que informa que queremos que esta chave seja populada pelo banco?

# Thymeleaf

## O que é?

---

Thymeleaf é um motor de templates para Java, ou seja, um mecanismo com capacidade para processar e criar HTML, XML, JavaScript, CSS e texto. Os templates são escritos, em sua maioria, com código HTML5 sendo mais adequado para servir XHTML / HTML5 na camada de visualização de aplicativos da Web baseados em MVC, mas pode processar arquivos mesmo em ambientes off-line e tem boa integração com o Spring Framework.

## Como Funciona:

---

### Dialeto padrão:

O thymeleaf vem com algo chamado dialetos padrão (chamados *Standard* e *SpringStandard*) que definem um conjunto de recursos que devem ser mais do que suficientes para a maioria dos cenários. Você pode identificar quando esses dialetos padrão estão sendo usados em um modelo porque ele conterá atributos começando com o **th**(prefixo), como `<span th:text="...">`.

Os dialetos Standard e SpringStandard são quase idênticos, exceto que o SpringStandard inclui recursos específicos para integração em aplicações Spring MVC.

### Sintaxe de expressão padrão:

A maioria dos atributos Thymeleaf permite que seus valores sejam definidos como ou contendo expressões que chamaremos de Expressões Padrão por causa dos dialetos nos quais são usados. Eles podem ser de cinco tipos:

- `${...}` : Expressões variáveis.
- `*{...}` : Expressões de seleção.
- `#{...}` : Mensagens (i18n) expressões.
- `@{...}` : Expressões de link (URL).
- `~{...}` : Expressões de fragmento.

### Exemplos:

- **th:each** : Percorre uma coleção de objetos enviada pelo controller;
- **th:if** : Habilita e desabilita controles do HTML de acordo com a condição recebida;
- **th:object** : Define o objeto que o controller irá receber e enviar por meio de um formulário;
- **th:field** : Faz bind dos atributos do objeto do formulário com os inputs;
- **th:href** : Para adicionar um link.

No código:

URLs absolutas permitem que você crie links para outros servidores. Eles começam especificando um nome de protocolo (**http://** ou **https://**)

```
<a th:href="@{http://www.thymeleaf/documentation.html}">
```

html

Os tipos de URLs mais usados são os relativos ao contexto. Estas são as URLs que devem ser relativas à raiz da aplicação Web. URLs relativos ao contexto começam com /:

```
<a th:href="@{/order/list}">
```

html

Basicamente um for each, onde percorre uma lista de objetos(no nosso caso mensagens)

```
<tr th:each="message : ${messages}"></tr>
```

html

# Bulma

O Bulma é uma estrutura CSS livre e de código aberto baseada no Flexbox (organiza elementos na página quando o layout precisa ser visualizado em telas e dispositivos de tamanhos diferentes).

## A configuração

Configurar o Bulma é super fácil, e você pode fazê-lo de várias maneiras diferentes, baixá-lo diretamente dos documentos ou usando um CDN. Após instalar o bulma, adicionaremos ao código:

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.6.2/css/bulma.min.css">
```

Isso nos dará acesso às classes Bulma. E isso é tudo o que Bulma é: uma coleção de classes.

## Modificadores (classes)

A primeira coisa que você deve aprender sobre o Bulma são as classes modificadoras. Estes permitem que você defina estilos alternativos para quase todos os elementos Bulma. Todos começam com *is-* ou *has-*, e então você substitui o \*com o estilo que você quer.

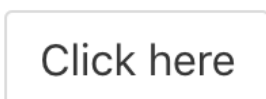
### *Vamos começar com alguns exemplos básicos:*

#### *Botões*

para transformar um botão normal em um botão Bulma, vamos simplesmente dar a classe de button.

```
<button class = "button">Click here</ button>
```

Que resulta no seguinte estilo:



Como você pode ver, ele tem um bom design plano por padrão. Para mudar o estilo, usaremos modificadores Bulma. Vamos começar fazendo o botão maior (classe "is-larger"), verde (classe "is-success") e com cantos arredondados (classe "is-rounded"):

```
<button class = "button is-larger is-success is-rounded">Click here</ button>
```

O resultado é um botão de aparência agradável:

Click here

Finalmente, vamos também usar um dos “has-\*” modificadores. Eles normalmente controlam o que está dentro do elemento. No nosso caso, o texto. Vamos adicionar “has-text-weight-bold” para deixar o texto em negrito.

Aqui está o resultado:

Click here

Eu recomendo que você brinque com combinações das várias classes para entender como esse sistema é flexível. As combinações são quase infinitas. Confira a seção de botões nos documentos para mais informações.

## Colunas

O principal de qualquer framework CSS é como eles resolvem colunas, o que é relevante para quase todos os sites que você já construiu. O Bulma é baseado no Flexbox, então é muito simples criar colunas. Vamos criar uma linha com quatro colunas.

```
<div class="columns">  
  <div class="column">First column</div>  
  <div class="column"> Second column </div>  
  <div class="column"> Third column </div>  
  <div class="column"> Fourth column </div>  
</div>
```

html

Primeiro, estamos criando uma `<div>` container com uma classe `“columns”`, em seguida, damos a cada uma das divs menores uma classe `“column”`. Isso resulta no seguinte:

First column	Second column	Third column	Fourth column
--------------	---------------	--------------	---------------

Observe que você pode adicionar quantas colunas desejar. O Flexbox se encarrega de dividir o espaço igualmente entre eles.

Para dar as cores das colunas, podemos substituir o texto dentro delas por uma `<p>` tag e dar a ela a “notification” classe e um “is-\*” modificador. Assim, por exemplo:

```
<p class = "notification is-success"> Primeira coluna </p>
```

Vamos fazer isso usando os “is-info” , “is-success” , “is-warning” e “is-danger” modificadores, o que resulta no seguinte:



A classe “notification” é, na verdade, apenas destinada a alertar os usuários sobre algo, pois permite preencher o plano de fundo com uma cor usando os "is-\*" modificadores.

Também podemos controlar facilmente a largura de uma coluna. Vamos adicionar o "is-half" modificador à coluna verde.

```
<div class="column is-half">  
  <p class = "notification is-sucess"> Segunda coluna </p>  
</div>
```

html

O que resulta na segunda coluna ocupando agora metade da largura, enquanto os outros três ocupam um terço da metade restante de cada um.



## Hero

Finalmente, vamos também aprender como criar um hero em Bulma. Vamos usar a semântica `<section>` , dar uma classe “hero” e “is-info” dar alguma cor. Também precisamos adicionar uma `<div>` criança à turma hero-body.

```
<section class="hero is-sucess">  
  <div class = "hero-body"> </ div>  
</section>
```

html

O resultado será este:



Para fazer esse hero fazer algo significativo, vamos adicionar um elemento de contêiner dentro do corpo e adicionar um título e uma legenda.

```
<div class="container">  
  <h1 class="title">Primary title</h1>
```

html



```
<h2 class="subtitle">Primary subtitle</h2>
</div>
```

Primary title  
Primary subtitle

Agora está começando a ficar bom! Se quisermos que seja maior, podemos simplesmente adicionar "is-medium" na própria tag:

```
<section class="hero is-info is-medium">
...
</section>
```

html

Primary title  
Primary subtitle

E é isso!

Existem vários componentes como cards, tabelas, menus, barras de navegação e várias outras coisas fáceis de usar e simples de compreender. Você agora tem um gostinho básico de como Bulma funciona, e a melhor parte é que o resto da biblioteca é tão intuitivo e fácil quanto os conceitos que você viu até agora. Então, se você entender isso, você entenderá o resto sem problemas.

Caso queira explorar, a documentação do Bulma se encontra em: <https://bulma.io/documentation/>

## EXERCÍCIOS DE FIXAÇÃO

Baseado no material desta apostila realize cada uma das tarefas a seguir:

- Crie um botão e mude ele para tres cores diferentes usando as classes do bulma;
- Mude a largura do botão;
- Deixe o botão arredondado;
- Crie um Hero;

- Mude o tamanho do Hero;
- Modifique o que está dentro do elemento, no nosso caso o texto do botão;
- Crie uma coluna e defina a sua largura.

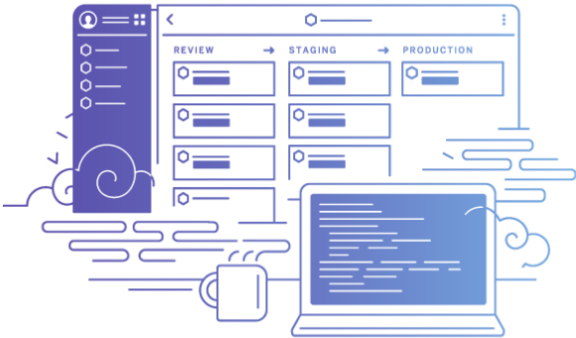
# Heroku

## O QUE É?

- Plataforma como serviço (PaaS).
- Criação de diversas aplicações: node, python, php.

 **HEROKU** [Products](#) [Marketplace](#) [Pricing](#) [Documentation](#) [Support](#) [More](#)

[Log in](#) or [Sign up](#)




### DEVELOPERS

## Focus on your apps


Invest in apps, not ops. Heroku handles the hard stuff — patching and upgrading, 24/7 ops and security, build systems, failovers, and more — so your developers can stay focused on building great apps.

[SIGN UP FOR FREE](#)

[Explore the Heroku Platform](#)



## INICIANDO SUA APLICAÇÃO

 **Heroku Dev Center** [Get Started](#) [Documentation](#) [Changelog](#) [More](#)

[Search Dev Center](#) [Menu](#) [User](#)

## Getting Started on Heroku with Node.js

[Introduction](#)

[Set up](#)

[Prepare the app](#)

[Deploy the app](#)

[View logs](#)

[Define a Procfile](#)

[Scale the app](#)

[Declare app dependencies](#)

### Introduction

This tutorial will have you deploying a Node.js app in minutes.

Hang on for a few more minutes to learn how it all works, so you can make the most out of Heroku.

The tutorial assumes that you have a free [Heroku account](#), and that you have [Node.js](#) and [npm](#) installed locally.

[I'm ready to start](#)

**Clique no link** <https://devcenter.heroku.com/articles/getting-started-with-gradle-on-heroku#deploy-the-app>

## INICIANDO SUA APLICAÇÃO COM NODE.JS

```
const cool = require('cool-ascii-faces')
const express = require('express')
const path = require('path')
const PORT = process.env.PORT || 5000

express()
  .use(express.static(path.join(__dirname, 'public')))
  .set('views', path.join(__dirname, 'views'))
  .set('view engine', 'ejs')
  .get('/', (req, res) => res.render('pages/index'))
  .get('/cool', (req, res) => res.send(cool()))
  .listen(PORT, () => console.log(`Listening on ${ PORT }`))
```

## INICIANDO SUA APLICAÇÃO COM EXPRESS-GENERATOR

### **Criar repositório GitHub (dar clone)**

1. \$ sudo npm install express-generator -g
2. \$ express --view=hbs /tmp/foo && cd /tmp/foo
3. \$ npm install
4. \$ npm start
5. Acesse: localhost:3000/

## INICIANDO SUA APLICAÇÃO COM JAVA

Será necessário o arquivo “pom.xml” na raiz do projeto.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  ...
</dependencies>
```

## SUAS IMPORTÂNCIAS

- Solução de alto nível.
- Abstrai os detalhes de infraestrutura.
- Manutenção.
- Escalabilidade.
- Agilidade para disponibilizar uma aplicação na web.

## DEPLOY DA APLICAÇÃO

## AMBIENTES STAGING - DEV

### PROCFILE?!?

```
procfile  x
1  web: java -jar target/java-getting-started-1.0.jar
```

Arquivo específico do Heroku;

Procfile não é necessário com linguagens suportadas pelo Heroku;

A plataforma detecta automaticamente a linguagem;

Procfile é recomendado para ter maior controle e flexibilidade na sua aplicação.

### Tipo do serviço: ação do Heroku

./gradlew assemble - gera o .jar

apontar o jar criado no Procfile

## HEROKU LOGS

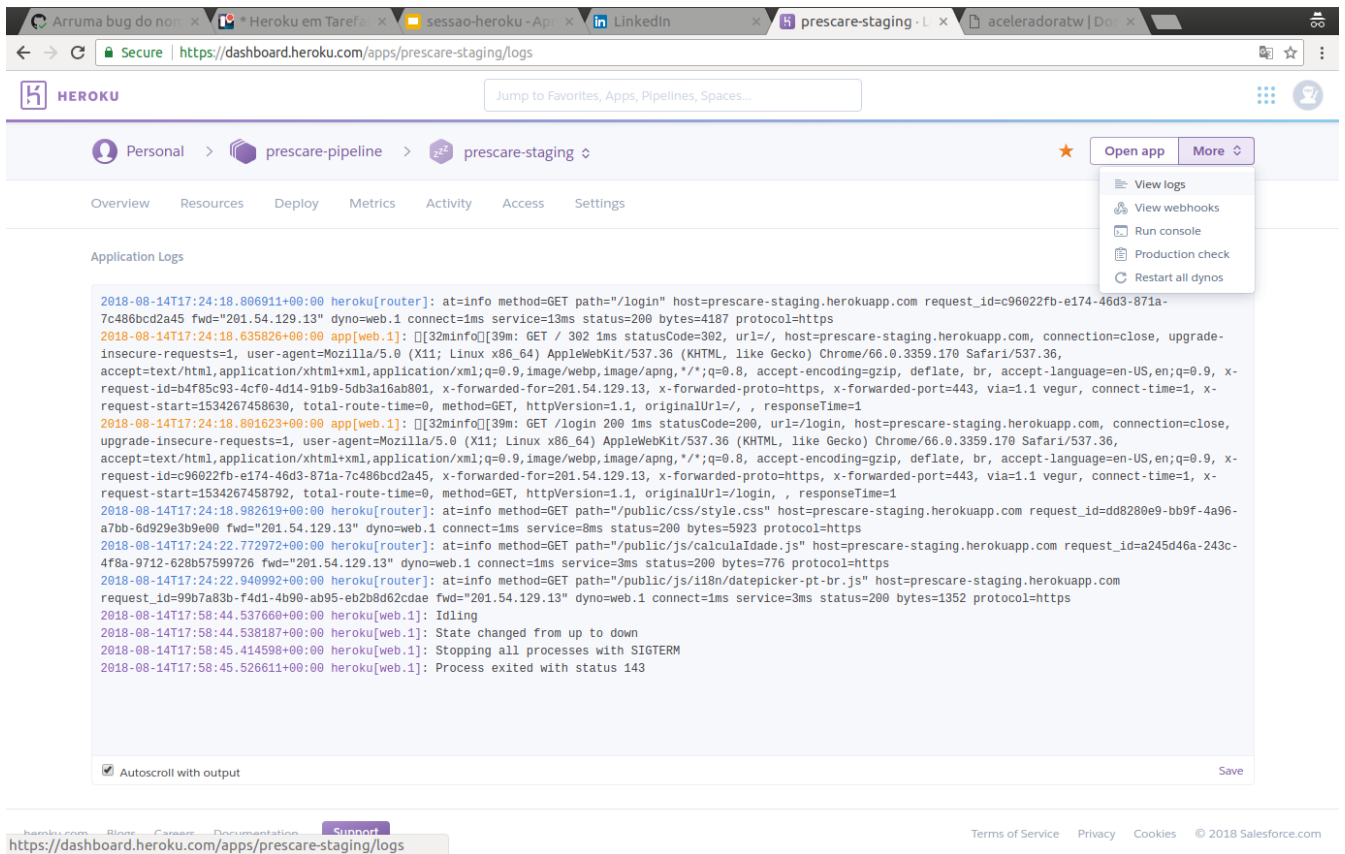
Os logs são um fluxo de eventos com registro de data e hora agregados dos fluxos de saída de todos os processos em execução.

```
---$ heroku logs -a "nome-da-aplicação"---
```

sh

```
aluno5@aluno5-OptiPlex-990:~$ heroku logs -a prescare-staging
> Warning: heroku update available from 7.0.66 to 7.7.8
2018-08-14T16:45:38.835762+00:00 heroku[router]: at=info method=GET path="/public/js/i18n/datepicker-pt-br.js" host=prescare-staging.herokuapp.com request_id=25783ea2-5574-4b59-94ee-1a32f194999b fwd="201.54.129.13" dyno=web.1 connect=0ms service=9ms status=200 bytes=1352 protocol=https
```

### HEROKU LOGS continuação



## COMO CONECTAR O BANCO NO HEROKU?

### Configurações necessárias:

Antes de iniciar o processo de configuração do ambiente, deve-se ter o heroku instalado

### Criar o banco usando postgresql no Heroku:

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

sh

## Heroku Add-ons

**Clique no link** <https://elements.heroku.com/addons>

## COMO FUNCIONA O BANCO NO HEROKU?

O postgresQL fica rodando no servidor junto com a aplicação.

## E O CRUD?

Comando para habilitar o Heroku CLI, Fazer os comandos do SQL direto para o Heroku:

```
$ heroku pg:psql -a <NOME DA APLICAÇÃO>
```

sh

Heroku Postgres é integrado com Interface de Linha de Comando CLI

heroku pg:psql — Executa um script

## LINKS PARA ESTUDO

Artigo para a config do banco em português: <https://bit.ly/2MJFDnP> 

Tutorial do Heroku: <https://bit.ly/1wiwQuP> 

Vídeo sobre Deploy no Heroku e Procfile: <https://bit.ly/2emdJw5> 

Curso gratuito de Heroku na Udacity: <https://bit.ly/2nDRc5j> 


## EXERCÍCIOS DE FIXAÇÃO

---

### Criando uma conta

O primeiro passo é criar uma conta, o processo é bem simples. Só precisa de nome/sobrenome/email/empresa e confirmar o email.

### Criando uma aplicação, fazendo deploy

Crie uma nova app(<https://dashboard.heroku.com/new-app> ) , a única informação requerida é a região (US or EU). O nome da aplicação é opcional (se não informar o heroku gera um nome aleatório). Sobre linguagens, o Heroku suporta Ruby, Python, Node, Php, Go, Java (e outras linguagens da JVM como scala / clojure / groovy).

### Fazendo deploy

É possível fazer deploy no heroku de várias formas, sendo as mais simples usar a ferramenta do heroku (heroku-cli), usar git ou dropbox.Faça o deploy.

# Docker

Docker é uma plataforma open-source escrita em GO cuja finalidade é criar ambientes isolados para aplicações e serviços. Com esse isolamento o docker garante que cada container tenha tudo que um serviço precisa para ser executado.

Uma das vantagens dessa abordagem é você poder iniciar esse serviço em qualquer máquina que sempre irá rodar da forma esperada, com bibliotecas, dependências e permissões configuradas da forma correta, sem surpresas.

## Instalando o Docker

```
apt-get install docker.io
```

## Comandos Básicos

```
docker pull [nome da imagem];
```

sh

Baixa a imagem.

```
docker images;
```

sh

Lista todas as imagens baixadas

```
docker run [nome da imagem];
```

sh

Inicia um container da imagem que você escolheu.

```
docker ps;
```

sh

Lista os containers em execução

```
docker stop
```

sh

Para a execução de todos dos containers

```
docker rm
```

sh



Exclui todos os containers criados

**Nota:** Para parar/excluir um container específico, é só colocar o nome do contaneir no final dos comandos listados.

## EXERCÍCIOS DE FIXAÇÃO

---

Baseado no material desta apostila informe os comandos necessários para realizar cada uma das tarefas a seguir:

Imagine que instalamos o docker

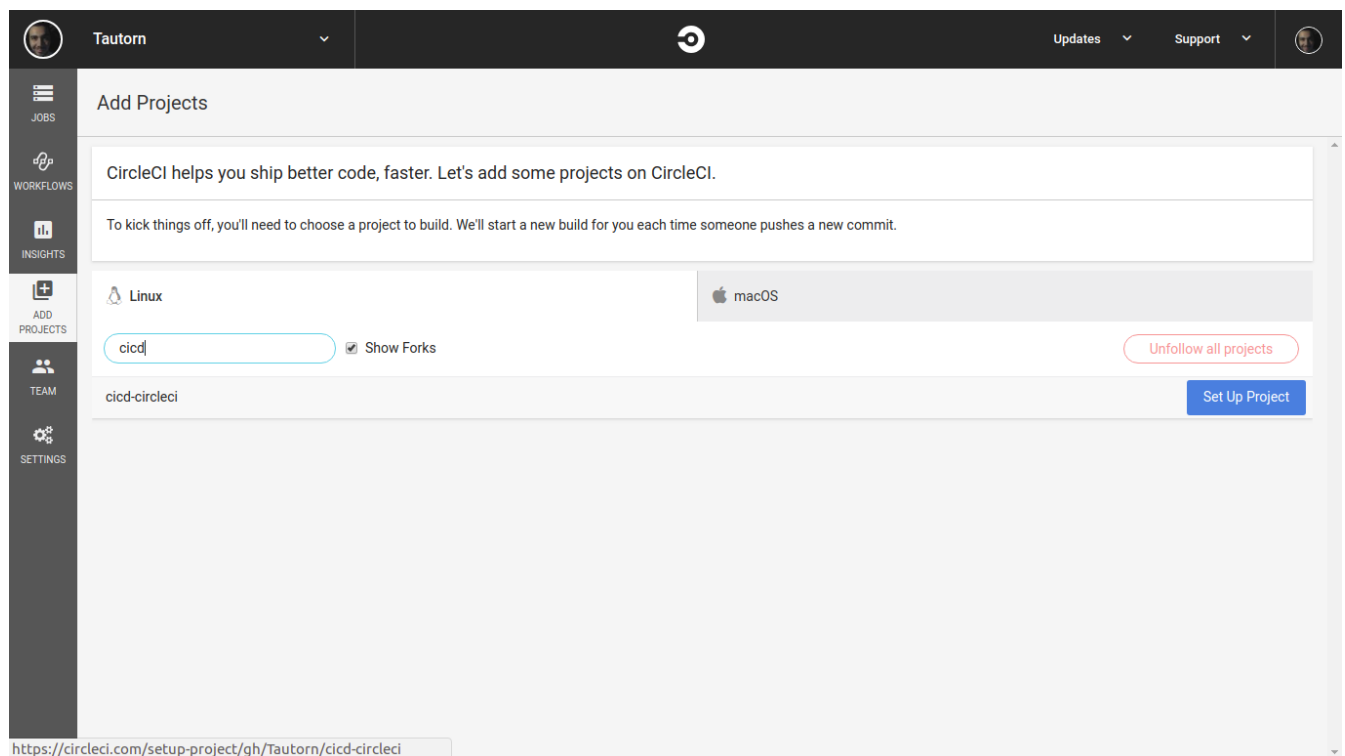
- ☐ Baixe a imagem
- ☐ Liste todas as imagens baixadas
- ☐ Inicie um container da imagem que você escolheu
- ☐ Liste os containers em execução
- ☐ PARE a execução de todos os containers
- ☐ Exclui os containers criados

# Circle CI

## Beleza, então como começar?

Faça o cadastro no <https://circleci.com/> e inicie o projeto selecionando o seu repositório (github).

Agora Vá em *Add projects* que irá listar todos os seus projetos, escolha um e clique *Set Up Project*



Agora escolha o Sistema Operacional e a linguagem.

pushes a new commit.

Select from the following options to generate a sample .yaml for your project.

Operating System

Linux macOS

Language

Clojure Elixir Go Gradle (Java) Maven (Java) Node PHP Python Ruby Scala Other

Next Steps

You're almost there! We're going to walk you through setting up a configuration file, committing it, and turning on our listener so that CircleCI can test your commits.

Want to skip ahead? Jump right [into our documentation](#), set up a .yaml file, and kick off your build with the button below.

**⚠️ If you start building before you've added a config.yml file, this will start a project on CircleCI 1.0, which will no longer be supported after August 31, 2018. [Add a project on CircleCI 2.0.](#)**

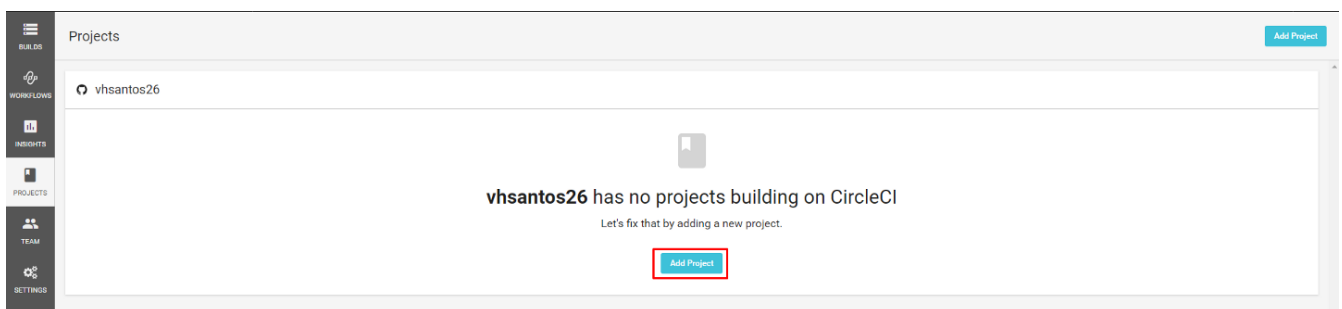
1. Create a folder named `.circleci` and add a file `config.yml` (so that the filepath be in `.circleci/config.yml`).
2. Populate the config.yml with the contents of the sample .yaml (shown below). [Copy To Clipboard](#)
3. Update the sample .yaml to reflect your project's configuration

Crie uma pasta nomeada `.circleci` adicione um arquivo `config.yml` (Preencha o `config.yml` com o conteúdo do arquivo `yaml` (mostrado abaixo no site)).


E então atualize o arquivo `.yaml` para ficar de acordo com seu projeto.

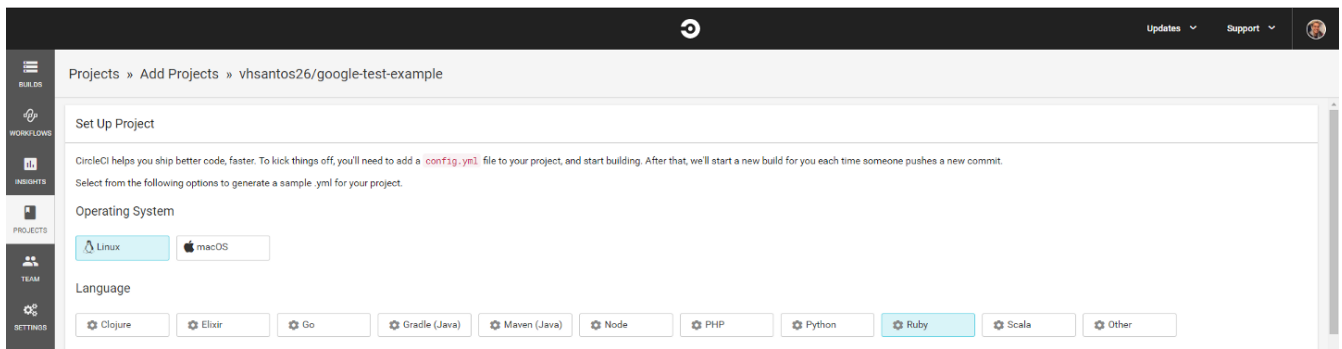
## EXERCÍCIOS DE FIXAÇÃO

Criando um projeto com CircleCI

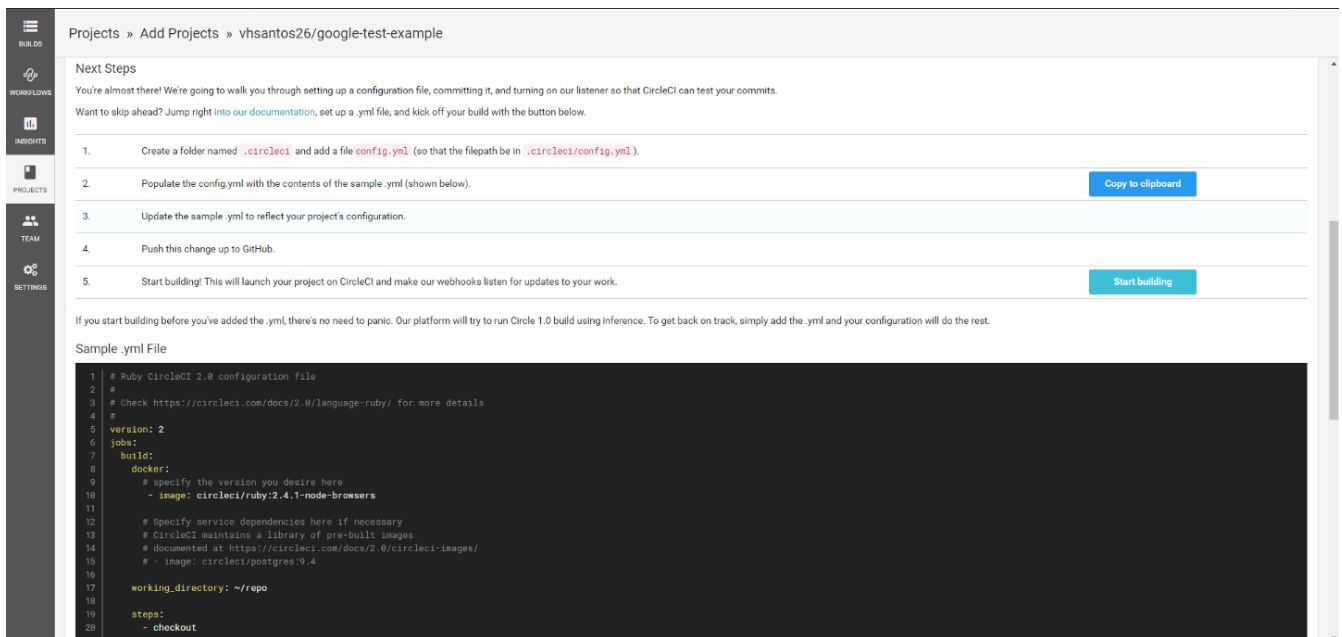


Acesse à aba Projects e adicione um novo projeto.

 Imagem Nesta tela, são exibidos todos os projetos existentes no github linkado. Selecione o projeto desejado clicando em Setup project.



Informações e configurações, como seleção do sistema operacional desejado, plataforma e linguagem do projeto que já vem configurada com a linguagem selecionada.



É possível saber como podemos configurar nosso projeto passo-a-passo para que a execução automática ocorra. Primeiramente, vamos criar o arquivo config.yml dentro do diretório .circleci/ no nosso projeto. Após isto, vamos copiar o conteúdo o Sample.yml sugerido pelo CircleCI para o arquivo config.yml.