# O livro do PROGRAMADOR

*Dorian Torres*

# O Livro do programador

*A bússola definitiva para você programar em qualquer linguagem*

*Dorian Torres*

# Por que você deve ler este livro

Ao longo de minha carreira profissional como desenvolvedor de sistemas, trabalhando com tecnologias de *back* e *front-end*, nos mais diversos nichos (desde aplicações financeiras, ERP, bases de dados, *e-commerce*, EAD), me deparei com os mais diversos tipos de situações, problemas e cenários.

Entre a evolução das diversas tecnologias envolvidas nesses cenários, acompanhei o andamento de linguagens e tecnologias diversas, como:

- *PHP*
- *Javascript*
- *HTML5 e CSS3*
- *JAVA*
- *C#*
- *Design Patterns,*
- *Processos de desenvolvimento*
- *Surgimento de frameworks como React, Vue.JS, Angular, Node JS*
- *Ferramentas de desenvolvimento 3D*
- *Fluxos de plataformas de pagamento*
- *Módulos de gerenciadores de conteúdo*

E como é natural, toda essa evolução costuma gerar confusão e dificuldade no aprendizado, principalmente em quem está iniciando na área.

Afinal, hoje em dia nos deparamos com uma característica importante, que é, ao mesmo tempo, a solução e problema: A quantidade de informações que temos disponível.

Chamo de **solução** pois nos dá um mar de possibilidades para pesquisa e aprendizado, nos mais diversos canais: livros, *audiobooks*, vídeos no *Youtube*, artigos de blog, exemplos em vários idiomas, códigos abertos para serem baixados e testados, editores de texto diversos.

E é também vejo como um grande problema: Se você enfrenta alguma dificuldade em organizar suas ideias e pensamentos, e construir uma boa estratégia de aprendizado.

A overdose de informação é um problema complicado…

Por causa disso tudo, eu resolvi reunir toda minha experiência, conhecimento, técnicas, estratégias de aprendizado, e então, elaborar um material rico e cheio de conteúdo, com exemplos e formas de elaboração distintas.

A intenção é **facilitar** seu aprendizado.

Eu quero fazer com que este livro seja um manual que abra as portas e janelas em sua mente, que ajude seu cérebro a fazer as conexões neurais da forma mais eficiente possível.

E isso inclui utilizar de forma produtiva e positiva os demais materiais ricos que já existem por aí.

Somos todos uma grande comunidade, reunida para somar, e multiplicar projetos e soluções.

Leia livros, assista aos vídeos, absorva conhecimento de outros profissionais. Filtre o que for necessário e aplique o que fizer sentido para você!

Afinal, nossa mente nunca se farta de conhecimento.

# Introdução

Estou escrevendo este livro em 2020.

O ano em que a programação é uma das atividades mais valorizadas do mundo.

Lado a lado com a venda, que é a atividade responsável por movimentar os poderes monetários no planeta.

Hoje, quem entende de programação, fala a língua dos grandes.

Fala a língua do homem e da máquina.

Entende criatura e criador.

Se você refletir por alguns segundos sobre isso, perceberá todo o poder que tem em mãos.

Diante disso, você está com uma oportunidade de ouro nas mãos.

Este livro vai ensinar você a ser não apenas um excelente programador, mas alguém capaz de aprender a resolver problemas.

Sejam eles simples ou complexos.

Essa é a uma das habilidades que eu mais valorizo: A capacidade de aprendizado. De adaptação.

Entender a estrutura por trás da ferramenta é o caminho exato para dominá-la.

Eu vou te ensinar a estrutura por trás das linhas de código.

E você será capaz de aprender a programar o que quiser.

Boa leitura!

# Tópicos do livro

# Estrutura do Computador

Se você é ou pretende ser um bom programador, é importante que entenda como os computadores funcionam, em sua estrutura básica.

A intenção dessa parte do livro é ir direto ao ponto, de forma objetiva e muito clara. Afinal, o que buscamos são resultados práticos.

É claro que não temos a intenção de perder anos da nossa vida entendendo como o computador funciona.

O mundo em que vivemos é dinâmico, e nós também somos.

As informações contidas nessa seção do livro podem ser bem básicas e repetitivas para pessoas que já trabalham na área ou já tem algum conhecimento em desenvolvimento de software.

Mas, se você é iniciante, ou atua mais com *front-end*, pode ser que essas informações sejam muito úteis para o seu dia a dia.

Então, chega de enrolação! Bora pro que interessa:

# Como o computador funciona

O que conhecemos como computador, é tecnicamente chamado de CPU – *Central Process Unit* e significa *Unidade Central de Processamento.*

É na CPU que todo acontece. Todos os cálculos, somas, atribuições, são feitas ali.

Esses cálculos são transformados em impulsos elétricos positivos ou negativos, que são **entradas** de circuitos lógicos diversos.

Esses circuitos lógicos transformam essas **entradas** em **saídas,** e geram os resultados que queremos (desde abrir um software até acionar uma impressora).

É uma “dança” digital muito bem orquestrada.

Para que nós, humanos, fôssemos capazes de manipular os impulsos elétricos (positivos e negativos), criamos um sistema numérico que permite apenas duas possibilidades: 0 e 1. Ligado ou desligado.

Isso se chama **sistema binário**.

Entenda que o sistema binário pode ser interpretado como o meio de campo entre homem e máquina.

É ele que permite que a máquina entenda o que queremos, e que a gente entenda o que ela produz.

# Linguagem de montagem

O sistema binário é um grande avanço na forma que nos relacionamos com a máquina. Porém, a solução trouxe consigo outro problema: a evolução do sistema.

Era fácil manipular dados binários enquanto tínhamos apenas 5, 8, 20, ou 100 possibilidades.

Mas... como poderíamos trabalhar com dados grandes, como é o caso de filmes, que tem milhões, ou trilhões de possibilidades de "zeros" e "uns"?

E mais: como um homem ou uma mulher seria capaz de conversar com a máquina (programar) a esse nível?

Esse problema fez com que fosse desenvolvida a **linguagem de montagem.**

Imagine uma linguagem de montagem como uma grande tabela que relaciona nomes com o seu equivalente em bits (dados binários).

Vamos trabalhar com um exemplo bem simples. Imagine dois valores: SOMA e SUBTRAÇÃO.

SOMA representa o bit "1".

SUBTRAÇÃO representa o bit "0". Da seguinte forma:

*SOMA – 0000 0001*

*SUBTRAÇÃO – 0000 0000*

Lembra da CPU? Lá na CPU existe um sistema chamado **compilador de linguagem de montagem.**

Esse sistema é como um dicionário. Ele verifica a palavra que entrou, e converte a saída para o seu equivalente em binário.

Então, quando o programador inserisse a palavra "SOMA", o compilador tentava encontrar nesse dicionário. Se encontrasse,

fazia a conversão e enviava os dados já convertidos.

Agora, imagine essa tabela com milhões, ou bilhões de instruções diferentes?

Seria impossível para um ser humano programar tudo isso em números binários.

Mas, é perfeitamente possível quando podemos utilizar uma linguagem que se parece mais com a que utilizamos para nos comunicarmos.

# Linguagens de alto nível

Mesmo depois que a linguagem de montagem tornou-se popular, o problema continuou crescendo. Afinal, as instruções eram confusas e complexas.

Os problemas sociais e gerais da humanidade demandavam uma maior produtividade (mais resultados com menor tempo de execução).

Então, novamente, foi necessário pensar em soluções ainda mais avançadas.

Para isso, a linguagem tinha que ser cada vez menos parecida com a linguagem de máquina, e cada vez mais parecida com a linguagem humana.

E mais do que isso: era necessário que as máquinas fizessem cada vez mais o trabalho automático, para que nós, cada vez mais, pudéssemos dar atenção à criações de soluções inteligentes (e não nos prendermos a resolver problemas de comunicação homem-máquina).

Então, criou-se uma nova “camada” de conversação.

## O compilador

Agora, imagine um compilador semelhante ao da linguagem de máquina que falamos antes.

Só que muito mais inteligente, eficiente e com mais recursos.

Agora, esse compilador transforma uma linguagem “superior” na linguagem de montagem (que falamos antes).

Isso possibilitou que criássemos as **linguagens de alto nível**.

Aquelas com condições e estruturas de fácil entendimento, e com uma linguagem muito semelhante ao nosso idioma natural.

**Linguagem C** foi uma das primeiras linguagens de alto nível construídas.

(E sim, sabemos que hoje, linguagem C não parece tão alto nível assim… (risos)).

Mas, é graças a essa linguagem que temos o mundo de possibilidades que temos hoje, seja em desenvolvimento web, desktop, aplicações mobile, e por aí vai.

E isso não parou mais.

Dia após dia, novas linguagens, novas tecnologias e novas formas de programar foram surgindo.

As linguagens de programação nos permitem, hoje, escrever verdadeiros contos.

É isso que você vai aprender neste livro: Você vai escrever histórias com as linguagens de programação.

E isso é só o começo.

# A Base da Programação

Se você nunca programou, essa parte é extremamente importante.

É a base de todo o funcionamento de linguagem de programação que você venha a ter contato.

Eu tenho como premissa básica que a base de qualquer aprendizado, é dominar a estrutura do assunto. Isto é, como funciona, e por que funciona.

Sendo assim, arrisco dizer: se você entende essa base que será mostrada a seguir, será capaz de programar em qualquer linguagem que você desejar.

# O que é um programa?

Ao utilizar as linguagens de programação, nós escrevemos diversos códigos e salvamos em um arquivo de determinado tipo.

Pode ser .c, .java, .php, .js, .asp, .aspx, .cshtml, .py…

Dependendo da linguagem utilizada, você salva o arquivo em um formato diferente.

Esse conjunto de códigos foi escrito seguindo uma lógica de passos. Este passo a passo é chamado de **algoritmo.**

Sabendo disso, um programa é um conjunto completo de um ou vários algoritmos.

Quando colocamos um programa em execução, seja em um navegador, um servidor ou em um computador pessoal, ele é compilado e/ou interpretado.

E a partir do momento que entra em execução, ele é chamado de **processo**.

Processos utilizam recursos de **disco**, **processamento** e memória **RAM**.

O processamento é a capacidade (e velocidade) que a *CPU* tem de calcular e gerar o resultado das instruções.

Já ouviu falar que alguns programas são chamados “pesados” ?

Então, quando um software qualquer exige muito processamento (como uma renderização 3D). Este tipo de conteúdo precisa de um número muito grande de instruções sendo executadas e avaliadas em conjunto.

Se o computador não der conta, costuma ocorrer os travamentos e aquecimentos.

Naturalmente, os fatores envolvidos não são apenas processamento. O disco e a memória RAM também fazem parte disso.

São eventos em conjunto.

# Memória física x Memória virtual (HD x RAM)

Tanto o disco quanto a memória RAM são memórias.

A primeira, é a memória física: Ela é armazenada mesmo depois que você desliga seu computador.

Essa memória pode ficar armazenada das mais diversas formas, seja em discos magnéticos, mídia ou memória flash.

O importante, é que ela é armazenada.

A RAM é uma memória que só funciona enquanto o sistema tem energia elétrica.

É considerada uma memória volátil.

A memória RAM é largamente utilizada para armazenar e controlar recursos em execução.

Quando você abre um documento em um editor de texto, por exemplo, o sistema carrega uma “cópia” desse documento na memória RAM.

E ainda, carrega vários outros recursos que permitirão que você edite o documento.

Quando você fechar o arquivo e o programa de edição, a memória RAM é liberada.

Esse espaço ocupado, tem um “endereço”. Tem um nome e a informação de qual espaço está sendo utilizado pelo que.

O processo de alocação de memória feita por um programa, é automático.

Não acessamos os endereços de memória diretamente. O computador aloca a memória conforme a necessidade do programa, dinamicamente.

Por isso a RAM é chamada de randômica.

A propósito, o que torna um software eficiente (desde um sistema operacional como o Windows até uma simples aplicação de abrir fotos), é sua capacidade de gerenciar recursos.

O gerenciamento de memória e processamento é fundamental para que uma aplicação funcione de forma adequada.

# Interpretação x Compilação

Agora que já sabemos o que é um programa, é preciso saber como eles são transformados em processos e por fim, gerar resultados, como exibir um gráfico, escrever um texto ou carregar uma imagem.

Isso se dá através da etapa que transforma o programa escrito em linguagem de alto nível em uma linguagem de baixo nível que a máquina possa entender (os zeros e uns, ou bits, ou impulsos elétricos).

Programas que são compilados precisam passar por um compilador antes de serem transformados em pacotes prontos para utilização.

Esse pacote costuma ser chamado de **build** pelos desenvolvedores.

Usamos esse processo no Javascript, quando trabalhamos com Node JS, por exemplo, que cria um servidor local, e compila o código e o transforma em um código para ser interpretado.

Na outra esfera, temos os programas interpretados. PHP por exemplo, é uma linguagem de servidor interpretada.

O javascript puro também é interpretado pelo navegador.

A principal diferença na interpretação, é o que o código vai sendo executado passo a passo, na ordem de execução que citamos antes: De cima para baixo, e da esquerda para a direita.

Na prática, vamos supor que temos um código com 5 blocos. Os 3 primeiros estão corretos e o bloco 4 tem um erro.

O navegador executará os 3 primeiros e mostrará o resultado na tela.

Ele encontrará o erro no bloco 4, e vai parar a execução.

O bloco 5 não será executado. Veja no exemplo a seguir, utilizando a interpretação de código de um navegador, a tentativa

de acessar uma propriedade com nome errado:

html

```
<div id="saida1"></div>
<div id="saida2"></div>
<div id="saida3"></div>
<div id="saida4"></div>
<div id="saida5"></div>
```

Javascript

```
// Recuperando variáveis do HTML, de acordo com atributo id
var saida1 = document.getElementById('saida1');
var saida2 = document.getElementById('saida2');
var saida3 = document.getElementById('saida3');
var saida4 = document.getElementById('saida4');
var saida5 = document.getElementById('saida5');
saida1.innerText = "Texto exibido na saída 1";
saida2.innerText = "Texto exibido na saída 2";
saida3.innerText = "Texto exibido na saída 3";
saidaerrado4.innerTexteRRADO = "Texto exibido na saída 4";
saida5.innerText = "Texto exibido na saída 5";
```

Você pode ver este exemplo neste código:
[Acessar no CodePen]

Este mesmo exemplo, em um código compilado, como em uma aplicação feita em *NodeJS*, o console de erros exibiria o erro, a compilação não seria feita.

O primeiro caso, é um **erro de execução.**

O segundo, é um **erro de compilação.**

Na linguagem interpretada, é considerado um erro de execução, pois ela SEMPRE está em execução. A linguagem interpretada sempre é um processo em andamento.

Ela não passa por um compilador ou um comparador léxico (que determina se a linguagem está escrita de forma correta).

Erros de execução não acontecem apenas na interpretação. Um erro de interpretação clássico é a divisão por zero.

É comum que linguagens interpretadas permitam a divisão por zero, e elas retornam algum valor do tipo *NaN* (Not a Number – Número não número). Por isso, não geram erro de execução.

Linguagens compiladas, por sua vez, costumam disparar erro na tentativa da divisão por zero.

# Tipos de dados

De forma direta: Tipos de dados em programação é o que define a alocação de recursos, especificamente: a memória RAM.

De forma básica, os processos envolvidos nas aplicações utilizam recursos de processamento e memória. Esses, são calculados em bits.

Os tipos de dados falam ao computador de quanto espaço será necessário para utilizar aquele valor.

E existem padrões para isso. Cada tipo de dados tem sua própria fatia de espaço.

Quem já trabalhou com Linguagem C, deve se lembrar como era complexo o processo de alocação de memória.

Graças a evolução das tecnologias, hoje é tudo muito simples.

O que precisamos saber, é que cada informação tem um tipo de dado diferente. Em geral, temos as seguintes nomenclaturas para os tipos básicos:

- char (caracteres individuais, como ‘a’, ‘e’, ‘b’)

- string (cadeia de caracteres, como “essa é a uma string com 36 caracteres”)

- int (números inteiros, negativos ou positivos)

- float (números decimais)

- double (números decimais com maior precisão)

- number (algumas linguagens utilizam number para todos os tipos numéricos, como o Typescript).

- array (o array em si não é um “tipo de dado”, é uma lista de dados que pode conter um ou vários tipos)

Alguns tipos de dados podem ter comportamentos diferente em linguagens diferentes.

Já outras podem ter tipos de dados que não existem em determinadas linguagens. Isso depende muito da tecnologia que está sendo utilizada.

Na sessão de Orientação a Objetos, você entenderá formas mais eficientes de estruturar tipos. Já que os tipos de dados falados anteriormente são os “tipos primitivos”, diferente das classes, que podem ser tanto criadas pelo programador, como ser preexistentes em algumas linguagens.

# Variáveis

As variáveis existem em quase todas as linguagens existentes.

Imagine as variáveis como caixinhas para guardar informação.

Então, quando eu quero armazenar um nome, eu crio uma caixinha chamada Nome, e coloco o nome dentro.

Quando quero armazenar um número, eu crio uma caixinha chamada Número, e coloco dentro.

É claro que utilizamos termos que sejam de fácil entendimento, para não gerar confusão.

Imagine um sistema de cadastro de alunos de até 16 anos. Este sistema tem dois campos:

- *Nome do aluno*
- *Nome do responsável*

Neste caso, eu posso criar as variáveis assim:

*var nomeAluno;*
*var nomeResponsavel;*

Assim, fica fácil para qualquer pessoa ler e entender do que se trata.

Evite criar variáveis com nomes genéricos, tipo:

*Nome1*, *nome2*, *nome3* ou *x*, *y*, *z*, *a*,*b*,*c*...

Em sistemas grandes, isso pode causar problemas sérios.

É comum também que as variáveis acompanhem o tipo de dados. No javascript puro, você pode atribuir uma *string* e depois um número em uma mesma variável.

Porém, não é o recomendado.

Em linguagens fortemente tipadas (e isso acontece também com o Typescript), cada variável deve ter seu tipo. Geralmente, os tipos de variáveis são definidos com o nome do tipo antes, ou após dois-pontos (:).

*Nome de variável com o tipo antes*

*String nomeAluno;*

*Nome de variável utilizando dois-pontos (utilizado no Typescript)*

*var nomeAluno: string;*

Essa nomenclatura depende da linguagem utilizada.

Variáveis também podem ser declaradas e atribuídas.

A declaração de uma variável define apenas seu nome, ou nome e tipo.

A atribuição coloca o valor dentro da variável. No javascript, utiliza-se o sinal de igual (=) para isso.

É possível fazer a declaração e atribuição ao mesmo tempo. Aliás, às vezes é até necessário, depende das necessidades do bloco de código sendo implementado.

# Conversão de tipos

É comum a situação em que seja preciso transformar tipos de dados.

Por exemplo: uma tela em que o usuário digita um número. Mas este número sempre gera uma STRING.

Essa string precisa ser transformada em um número (NUMBER) pelo sistema.

Cada linguagem tem suas próprias regras de conversão.

Linguagens fortemente tipadas irão disparar um erro caso você tente atribuir valores de tipos diferentes.

Outras, permitirão a atribuição, se a conversão for possível.

O nome disto é **conversão implítica.**

A conversão implícita acontece quando a variável recebe um valor e o interpretador consegue transformar sozinho o tipo de dado.

Por exemplo: Uma variável do tipo Number recebe uma string “1”.

Ele entende que “1” é uma string, mas pode ser convertido para o número 1.

Caso a tecnologia não permita essa atribuição, será preciso tentar uma conversão explícita. Isto é: Forçamos o interpretador a entender que queremos uma conversão.

Algumas formas geralmente utilizadas para conversão explícita:

*var numero = (int) numeroEmTexto; (C#)*

*var numero = numeroEmTexto as Number; (Typescript)*

*var numero = new Number(“1”) (Javascript)*

*$numero = intval(“1”); (PHP)*

# Estruturas de dados

As estruturas de dados são conjuntos complexos de informação, que contém um ou mais tipos de dados.

As estruturas de dados utilizam espaço estático ou dinâmico na memória, e devem ser utilizados com cautela, pois podem causar problemas de performance.

As estruturas de dados mais comuns são:

- *Arrays (matrizes)*
- *Queues (filas)*
- *Stacks (pilhas)*
- *Lists (listas)*
- *Dictionaries (dicionários)*

Além desses, podem existir outras estruturas, baseadas nessas, que são mais complexas e com utilidades específicas. Isso vai depender da linguagem, do *framework* e da tecnologia utilizados.

As estruturas de dados são muito úteis e largamente utilizadas, quando trabalhamos com coleções, essencialmente.

Por exemplo: Se eu tenho 1 nome, eu crio uma variável.

Se eu tenho 3 nomes, eu posso criar 3 variáveis.

Mas, se eu tenho 10 nomes, ou eu não sei quantos nomes posso ter, é interessante trabalhar com uma estrutura de dados.

Geralmente, fazemos isso quando temos dados dinâmicos que vem de um banco de dados, por exemplo.

Geralmente, a declaração de estruturas são acompanhadas de colchetes:

*var array = [];*
*string[] nomes;*

# Operadores

Na programação, são os operadores que realizam os cálculos, fazem atribuições, condições e outras possibilidades. Os operadores são divididos em grupos.

Naturalmente, a lista de operadores aqui é para que você tenha uma ideia de como funciona. Consulte a documentação específica de cada linguagem para entender as particularidades  e possíveis operadores adicionais.

*Operadores aritméticos*

+ = soma dois números. Em algumas linguagens, concatena (junta) strings.

- = subtrai dois números.

* = multiplicação

/ = divisão

*Operadores lógicos*

&& = AND (Verifica se duas condições são verdadeiras)

|| = or (verifica se uma ou outra condição é verdadeira)

*Operadores bit a bit*

Os operadores bit a bit são um pouco mais complexos, e manipulam os valores dos bits das variáveis envolvidas, deslocando bits para a direita e para esquerda.

Sua utilização depende de um domínio das operações e cálculos binários e tem utilidades específicas.

*Operadores de atribuição*

Você pode atribuir qualquer valor com o sinal de igual (=).

E você também pode calcular e atribuir qualquer operação aritmética.

Por exemplo:

+= (soma e atribui)

-= (subtrai e atribui)

++ (neste caso, soma e atribui 1. O 1 é implícito)

– (subtrai e atribui 1. O 1 é implícito)

# Estruturas de código

As diversas estruturas de código (existentes em todas as linguagens de programação) é o que nos permite trabalhar com os dados, manipular valores, considerar condições, tomar decisões, automatizar tarefas.

As principais estruturas de código conhecidas e utilizadas são:

*for (início; condição de parada; alteração)*

O famoso laço de repetição “para”. É a estrutura básica para repetir dados.

Ela é baseada em 3 partes diferentes controladas por um número inteiro, geralmente: Início, condição de parada e regra de alteração.

*Início = É o DE – De onde começar*

*Parada = É o PARA – PARA onde vai.*

*Alteração = Incremento ou decremento. Pode ser +1 -1, ou qualquer outra fórmula. Vale a observação: Dependendo da condição, o FOR nunca para. Fique atento!*

Exemplo de uso:

```
var inicio = 0;
var parada = 10;

//Este código significa que vai começar em “inicio” (0). Vai executar enquanto “início” for menor que “parada” (inicio < parada). A cada vez que executar o código de dentro do for, será incrementado 1 em “inicio” (inicio++)
for(inicio, inicio < parada, inicio++) {
// Aqui vai sua lógica
}
```

*while (condição)*

O while também é uma estrutura de repetição, que se baseia apenas na condição de parada. É importante controlar a condição dentro do while, para que ele não se torne um laço infinito.

Exemplo:

```
var parada = 0;
while(parada < 10) {
//sua lógica aqui. Depois, incrementa 1 na parada.
parada ++;
}
```

*if (condição)*

O if é a estrutura que verifica se o resultado (condição) é TRUE ou FALSE.

Exemplo:

```
//Exemplo de retorno FALSE. Não entra na condição
var valor1 = 2;
var valor2 = 3;
var resultado = 10;
if (valor1 + valor2 == resultado) {
}

//Exemplo de retorno TRUE. Entra na condição
var valor1 = 2;
var valor2 = 3;
var resultado = 5;
if (valor1 + valor2 == resultado) {
}
```

*switch (opção)*

O switch existe como uma estrutura que define opções diversas. Atualmente, é uma estrutura de código pouco utilizada, devido aos padrões de código e *design patterns* (padrões de projeto) utilizados.

Dentro do **switch** usamos a estrutura **case** que define algo como “quando a opção for esta”.

A condição de parada do **switch** é o **break** ou **return.** Se a estrutura de parada não for utilizada, o **switch** executará todas as próximas opções, à partir da encontrada.

O **switch** também permite a estrutura **default,** que será executada caso nenhum **case** seja encontrado.

Vários **case´s** podem ser colocados juntos, buscando um mesmo retorno para dois casos diferentes. Essa regra também vale para o **default**.

```
//Exemplo com break
var nome = “”;
switch(opcao) {
	default:
	case 1:
	nome = “Nome 1”
	break;
	case 2:
	nome = “Nome 2”
	break;
	case 3:
	case 4:
	nome = “Nome 3 ou 4”
	break;
}
```

```
//Exemplo com return
var nome = “”;
switch(opcao) {
	default:
	case 1:
	nome = “Nome 1”
	return;
	case 2:
	nome = “Nome 2”
	return;
	case 3:
	case 4:
	nome = “Nome 3 ou 4”
	return;
}
```

# Funções

As funções são estruturas de códigos independentes que podem ser chamadas em outros lugares do código.

As funções geralmente tem nome, tipo de retorno e seus argumentos. Exemplo:

```
// Exemplo de função em javascript
function somar(valor1, valor2) {
   return valor1 + valor2;
}

// Exemplo de função em Typescript
somar(valor1: number, valor2: number): number {
   return valor1 + valor2;
}
```

Quando chamamos uma função, passamos os **parâmetros**, caso tenha

```
// Chamando a função “somar”, que retornará 13, neste caso.
 somar(5, 8)
```

# Programação Orientada a Objetos

## Classes, propriedades e métodos

A orientação a objetos foi um dos grandes passos que a tecnologia deu, e possibilitou que os sistemas tomassem uma escala muito maior.

A **programação orientada a objetos** é a definição para transformar o que escrevemos de código em dados como se fossem objetos, facilitando o entendimento e a organização dos códigos.

Sendo assim, a orientação a objetos é baseada em modelos, que chamamos de **classes.**

Explicando de forma leiga: Uma **classe** é um modelo que define regras de comportamento e características.

As características chamamos de **propriedades.**

Os comportamentos chamamos de **métodos.**

Para usar um exemplo da vida real, vamos criar uma empresa e os colaboradores dentro dela. Veja:

**Classe Programador**

- **Propriedades:** *Nome, altura, idade, lista de linguagens de programação, formação.*
- **Métodos:** Andar, programar, tomar café.

Agora, vamos transformar isso em uma classe na linguagem C#:

```
class Programador {
        string Nome;
        double Altura;
        int Idade;
        string[] LinguagensDeProgramacao;
        string Formacao;
        public void Andar() {
        Console.WriteLine(“Programador está andando”);
        }
        public void Programar() {
        Console.WriteLine(“Programador está programando”);
        }
        public void TomarCafe() {
        Console.WriteLine(“Programador está tomando café”);
        }
}
```

Naturalmente, se esse código for compilado e executado, e essa classe for utilizada, apenas uma mensagem será exibida no console, e nada mais.

Os métodos devem realizar tarefas úteis, cálculos, retornar informações, e tudo aquilo que for útil para o sistema.

E como utilizamos uma classe? Através das **instâncias.**

Uma classe por si só não tem utilidade. Como falamos antes, ela é apenas um modelo. E se não for utilizada, isto é, **instanciada**, ela não terá utilidade.

Nós criamos uma instância de uma classe e então temos um **objeto**.

Veja:

```
var Pedro = new Programador();
Pedro.Nome = “Pedro”;
Pedro.Idade = 27;
Pedro.Altura = 178;
Pedro.LinguagensDeProgramacao = new string[] { “C#”, “Cobol”, “Javascript” };
```

No exemplo acima, nós criamos uma **instância** da classe Programador, e colocamos em uma variável chamada “Pedro”.

E então, definimos algumas das **propriedades**.

Agora sim, temos um objeto, que pode ser manipulado e utilizado pelo sistema.

# Herança

Herança em orientação a objetos é um recurso muito útil. Imagina o mesmo exemplo do programador acima.

Se eu precisasse definir programador Front-End e programador Back-End, de modo que eu consiga manipular essas classes de forma separada, e reconhecê-las em meu sistema.

Eu poderia criar duas novas classes. Veja como ficaria em C#:

```
class ProgramadorFrontEnd: Programador {
    string[] LinguagensDeMarcacao;
}

class ProgramadorBackEnd: Programador {
    string[] BancosDeDados;
}
```

Apenas para exemplificar: a classe ProgramadorFrontEnd tem uma propriedade “LinguagensDeMarcacao” e a classe ProgramadorBackEnd tem uma propriedade “BancosDeDados”.

Essas classes **herdam** de Programador (em c#, a herança é repreentada pelos dois-pontos logo após o nome da classe). E essas classes possuem todas as propriedades e métodos que a classe “Programador” possui.

Sendo assim, eu não preciso criar as duas propriedades específicas que criei nessas classes, na classe “Programador”. E eu consigo ganhar algum nível de performance e organização de código.

Exemplos de utilização:

```
var Joao = new ProgramadorFrontEnd();
Joao.Nome = “João”;
Joao.Idade = 22;
Joao.Altura = 168;
Joao.LinguagensDeProgramacao = new string[] { “Javascript” };
Joao.LinguagensDeMarcacao = new string[] { “XML”, “HTML5” };

var Laura = new ProgramadorBackEnd();
Laura.Nome = “Laura”;
Laura.Idade = 23;
Laura.Altura = 170;
Laura.LinguagensDeProgramacao = new string[] { “Java”, “C#”, “PHP” };
Laura.BancosDeDados = new string[] { “Sql Server”, “MySql”, “Oracle DB” };
```

# Interfaces

Uma **interface** é um contrato que define regras. Diferente das classes, uma interface geralmente não pode ser utilizada de forma direta. Ela deve ser **implementada.**

Isso significa que deve ser criada uma classe que implemente essa interface.

A interface também define um nível de hierarquia e garante que toda classe que a implemente seguirá as mesmas regras.

Usando o exemplo do programador citado anteriormente, imagine que agora existe uma Interface IProgramador. Veja o exemplo em C#:

```
interface IProgramador {
        public void Andar();
        public TomarCafe();
        public Programar();
}

class Programador: IProgramador {
        public void Andar() {
        Console.WriteLine(“Programador está andando”);
        }
        public void Programar() {
        Console.WriteLine(“Programador está programando”);
        }
        public void TomarCafe() {
        Console.WriteLine(“Programador está tomando café”);
        }
}
```

Se um ou mais métodos da interface não fossem implementados, o **compilador** retornaria um erro de compilação. Pois ele exige que a interface seja obedecida.

Interfaces são extremamente úteis para garantir padronização de projetos e reutilização de códigos.

# Getters e Setters

Getters e Setters são funcionalidades comuns em orientação a objetos, e elas representação as funções de **atribuir** ou **recuperar** valores de uma propriedade.

Eles são úteis para escrever regras para cada propriedade, se for preciso, ou mesmo limitar o acesso de algumas e ampliar de outras.

Na linguagem C#, eles são representados pelas palavras-chave *get* e *set*.

Uma propriedade somente-leitura poderia ser criada quando criamo-na apenas com um *get*. Já que uma propriedade sem um *set* não pode ser atribuída.

# Níveis de acesso

Os níveis de acesso na orientação a objetos são importantes para diversos fins.

Eles definem o **escopo** de visualização de propriedades e métodos de uma classe.

Geralmente, são baseados em: **público, protegido e privado,** comumente representados pelos termos em inglês: *public*, *protected* e *private*.

Um item **private** só pode ser acessado pela própria classe. E nenhuma outra. Ou seja: Uma instância não pode acessar um item privado. Nem uma classe herdada.

Um item **protected** só pode ser acesso por classes dentro do mesmo escopo. Isso significa que classes herdadas ou dentro do mesmo **namespace** (conjunto de classes), dependendo das regras de cada linguagem. Uma instância também não pode acessar um item **protected.**

Um item **public** pode ser acessado por qualquer instância desta classe.

A organização dos níveis de acesso são úteis e importantes para manter a organização e as regras de negócio.

Por exemplo, vamos supor que nossa classe Programador tem uma nova propriedade Nível, que representa o nível do programador. E então, temos uma outra propriedade Descricao que representa uma descrição completa do programador.

Veja:

```
class Programador {
        private string _nivel;
        public string Nivel {
        set {
        _nivel = value;
        CriarDescricao();
        }
        get {
        return _nivel;
        }
        }
        public string Descricao {private set; get;}

        private CriarDescricao() {
        Descricao =  Nome + " | " + " Desenvolvedor " +  Nivel;
        }
}
```

Nesta implementação, só adicionamos os itens novos. E neste caso, temos uma propriedade nova: _nome e um método novo “CriarDescricao”. Ambos *private*.

Este código faz com que, assim que Nível for atribuído, será chamado o método CriarDescricao, apenas internamente, para definir o valor de Descricao.

O método CriarDescricao não pode ser acessado externamente. E a propriedade “Descricao” retornará a descrição completa do programador.

Por sua vez, “Descricao” também possui um *private set*. Isso é uma peculiaridade da linguagem C# e significa que ela tem um *setter* privado, que só pode ser acessado internamente, e nunca de uma instância.

# Estrutura da Web

Nesta sessão, vamos entender como a internet funciona.

Entender a estrutura de qualquer assunto é o princípio básico para dominar os procedimentos envolvidos.

Quando você vai começar a construir uma casa, por exemplo, você começa pela estrutura.

No mundo digital é a mesma coisa.

A partir do momento que você começar a entender a estrutura de como funciona, você será capaz de pensar de forma objetiva para desenvolver as soluções que precisa.

Considere isso vale para front-end, back-end e qualquer outra tecnologia para desenvolvimento de software.

# Como os dados trafegam na internet

A estrutura de comunicação na internet toda, basicamente se dá em 3 passos – os mesmos passos da comunicação humana: Emissor, canal e receptor. Isto é: Alguém pede uma informação.

Essa solicitação passa por um canal, e chega até o receptor. E esse ciclo pode se repetir ou fazer o caminho inverso.

Na internet, acontece de modo semelhante:

*1. Web request:* Quem solicita a informação chama-se **CLIENTE**.

Esse processo é chamado de Web Request (traduzindo para o português: Requisição Web).

*2. Envio da mensagem:* Essa solicitação passa por um **canal**, que é informação digital trafegando pelo seu provedor de internet, por exemplo.

*3. Web response:* A solicitação chega até o receptor, que neste caso chama-se **SERVIDOR.**

O servidor lê, interpreta, verifica se tudo está correto, e tenta encontrar o que foi solicitado.

E então, devolve a informação para o canal que volta ao **CLIENTE.** Este processo é chamado de Web Response (resposta web).

É isso que acontece quando alguém clica em um link na internet.

Percebeu quem é o cliente nesse caso? Sim, o navegador.

Assim que o navegador recebe a informação do servidor (ali em cima no passo 3), esta é verificada, tratada, manipulada se

preciso for, “maquiada”, e tudo mais o que for necessário para exibir da forma desejada, em **HTML.**

# Infraestrutura (sites, servidores e IP)

Além da estrutura do funcionamento e fluxo da internet, é importante entender como a infraestrutura básica funciona.

Entenda que a internet (a grande rede onde todos nós estamos conectados) é toda baseada em endereços de rede. Esses endereços são números que chamamos de IP (*Internet Protocol*).

Existem dois tipos de IP´s: fixos e dinâmicos.

**IP dinâmico** é aquele do modem de sua residência. Cada vez que você liga e desliga seu modem, este IP se altera.

**IP fixo** é utilizado por servidores em uma rede. É um caminho que nunca se altera, permitindo que qualquer outro ponto na rede (outro IP) consiga acessá-lo (respeitando, é claro, as limitações de permissões, segurança e tudo mais).

Sites, por exemplo, são hospedados em um **servidor**, e por consequência, possuem IP fixo.

E isso nos leva a um outro assunto, que é o sistema de nome de domínio, popularmente conhecido como DNS (continue a leitura).

## Rotas e DNS

Agora que já entendemos o que é e como funciona o protocolo da internet (IP), imagine se a cada vez que fôssemos acessar um site, tivéssemos que lembrar cada IP ou ler isso em algum histórico? Ficaríamos perdidos.

Para resolver esse problema, foi criada uma "**tabela**" conhecida como **DNS.**

**DNS** é o *Domain Name System* – Sistema de Nomes de Domínio.

De forma básica, o DNS faz a conversão do IP de um site para um nome específico.

Por exemplo:

Na tabela de DNS do google, existe uma definição assim:

**google.com |** *142.250.218.78*

Isso indica que ao digitar google.com, o endereço de rede acessado será o IP definido acima.

E então, temos a definição de **rota.**

Uma rota (quando falamos de DNS) é a conexão entre o domínio / subdomínio e determinado IP.

Um IP pode ter mais de uma rota diferente.

Quando adquirimos um **domínio** (pelo Registro BR por exemplo), passamos a ter uma propriedade digital, que, no momento da compra, ainda não possui rotas definidas.

O domínio comprado tem uma informação importante, que são os **Servidores DNS.** Os servidores DNS são endereços na internet que controlam o DNS daquele domínio.

Os servidores DNS são extremamente úteis para apontarmos para serviços externos, que por sua vez faz a conexão com servidores, onde um site será hospedado, por exemplo.

Vamos supor que você comprou o domínio **xpto.com.br.**

E você contratou uma hospedagem que tem o IP **999.999.999.99.**

Essa hospedagem tem dois nomes de servidores DNS, por exemplo:

*hospedagem.servidor1*

*hospedagem.servidor2*

No serviço onde você comprou o seu domínio, existe um lugar onde você configura os servidores DNS. Entrando lá, você altera esses valores, para os valores da sua hospedagem (geralmente, isso é identificado como NS1, NS2, NS3).

A partir desse momento, o seu domínio tem como servidor DNS definido para a hospedagem que você contratou.

Basta aguardar o tempo de propagação (o período varia de 5 minutos até 48 horas). Assim que estiver concluído, você será capaz de configurar o DNS do seu domínio diretamente na sua hospedagem.

Pronto, mas agora, seu site ainda não está no ar.

Para que isso aconteça, é preciso mapear a **rota** do seu domínio para o IP da sua hospedagem.

Sendo assim, você criará na ZONA de DNS do seu domínio, dentro da sua hospedagem, um registro que aponte para o IP da sua hospedagem.

Esse registro é do “tipo A”.

Cada registro DNS tem seu próprio tipo, dependendo das necessidades. Registro DNS de apontamento é chamado de ‘A’ (de alias).

Nesse novo registro tipo ‘A’, que recebe um NOME e um VALOR, será configurado assim:

**TIPO: A**

**NOME**: fica em branco, ou um @, que define o ‘root’, isto é, se refere ao domínio principal. A definição de ‘root’ pode ser diferente para hospedagem ou serviço, e é informada por eles.

**VALOR**: o IP da hospedagem.

Pronto.

Agora, basta seguir as instruções da sua hospedagem para subir arquivos no seu site (veja detalhes sobre isso na sessão **DEPLOY).**

Se tudo foi feito corretamente, e todas as alterações já foram propagadas, quando você acessar seu domínio de um navegador, será exibido o conteúdo do seu site, que está hospedado na sua nova hospedagem.

**Passo a passo para a configuração do DNS**

1. No site onde você comprou o domínio, encontre a definição de servidores DNS.

2. Na hospedagem onde você terá seu site, encontre os servidores DNS.

3. Troque os servidores DNS do domínio, pelos servidores DNS da sua hospedagem.

4. Na sua hospedagem, encontre o seu IP.

5. Na tabela de DNS, descubra qual a definição do *root* (geralmente é um @)

6. Na tabela de DNS, encontre o botão “novo registro”.

- *Insira um registro do tipo ‘A’.*
- *Em ‘nome’, coloque a definição do root encontrada acima.*
- *Em ‘valor’ insira o IP da hospedagem.*

7. Pronto, seu domínio está configurado, apontando para a hospedagem contratada.

## SSL – Secure Sockey Layer

Sempre acessamos um site através do protocolo HTTP.

Derivado do HTTP, temos o HTTPS, que indica que a URL é segura.

Isso significa que existe um certificado digital que foi emitido e que atesta que a conexão com este servidor é segura, criptografada, e tem níveis de segurança (de acordo com as definições deste certificado).

As URL´s seguras são extremamente importantes nos dias atuais. E temos serviços que auxiliam nesse processo, inclusive, como o *Cloudflare*, que é utilizado em larga escala por grandes empresas.

A implementação de Url segura em um servidor pode ser feito através de:

1. Adquirindo um certificado digital (gratuito ou pago) que é instalado no servidor em questão

2. Utilizando um serviço como o Cloudflare, citado acima. Sendo assim, todo o tráfego daquele servidor é “controlado” pelo Cloudflare, é um intermediário entre o cliente acessando o recurso, e o servidor em questão. Isso é feito através da definição dos servidores DNS do domínio (veja na sessão de DNS).

## Deploy

Chamamos de *deploy* o processo de publicar desenvolvimento e/ou alterações de sites ou sistemas em um servidor de hospedagem.

Geralmente, os serviços de hospedagem tem um manual ou um descritivo explicando o processo de publicação, bem como meios utilizados (como dados FTP).

Basicamente, após finalizar seu desenvolvimento, você fará a publicação, que é feita em 2 etapas: a **publicação** e o **deploy.**

Costuma-se chamar de **publicação**, o processo de transformar o que foi desenvolvido em um pacote pronto para o *deploy*. Não significa necessariamente que o *deploy* será feito na sequência.

É possível fazer uma publicação, gerar um pacote (que pode ser uma pasta de arquivos, um *ZIP* ou um conjunto de *DLL´s*), e deixar salvo.

O *deploy* acontece quando decidimos pegar todo esse conteúdo e colocar no servidor, tornando-o pronto para uso.

A publicação pode ser diferente para cada *framework* ou tecnologia. Por exemplo, uma publicação de um projeto Angular é feito através do comando: *ng build*

Uma publicação de um projeto *.Net* pode ser feito diretamente pelo *Visual Studio*, ao clicar com o botão direito, e utilizar a opção "*Publicar...*".

E projetos como PHP ou HTML / CSS e Javascript "puros" (que são interpretados), basta colocar o conteúdo diretamente no servidor (pulamos a etapa da "publicação" e fazemos o *deploy* de forma direta).

Após estar pronto para subir seus arquivos (seja uma pasta com os arquivos compilados ou o projeto na sua máquina), é necessário saber a forma de deploy e dados de acessos.

O modo mais comum e antigo é via FTP. Onde basicamente, é feita uma cópia de arquivos de um lugar para outro.

Isso é feito através de:

- *Endereço do servidor*
- *Login*
- *Senha*

Basta configurar tudo isso em um *client* de FTP, e pronto, você estará com acesso às pastas do servidor. Geralmente, a pasta que é acessada assim que o site é solicitado tem nomes comuns como "*wwwroot*", "*public*" ou "*www*".

Essas informações são descritas pelo serviço de hospedagem.

Naturalmente, os servidores possuem permissões e regras de escrita e leitura em determinadas pastas. Então, é comum que algumas configurações ou validações de dados desse tipo sejam verificados para que tudo funcione corretamente.

Além do FTP, é bastante comum utilizarmos processos automáticos de *deploy*, como as **integrações contínuas,** que são integrações feitas de um **repositório de versionamento** (como o GIT), diretamente para o ambiente em questão.

Isso é feito através de configurações e regras bem definidas.

## Repositórios de Versionamento (GIT)

Versionamento é uma funcionalidade muito importante para garantir desenvolvimento fácil, organizado, simples e com fácil acesso a alterações passadas.

Todas as alterações feitas em desenvolvimento, que forem enviadas para um servidor de versionamento, tem um número próprio de versão, e ainda permite que você inclua sua própria mensagem de observação sobre o que foi desenvolvido / alterado em cada versão

Basicamente, esses servidores funcionam da seguinte forma:

- No servidor, existe a **versão original** do código.

- Essa versão original pode ser dividida em várias **ramificações** (branches)

- Essas **ramificações** podem ter desenvolvimentos específicos

- Então, para você fazer seu desenvolvimento, você baixa uma cópia da versão original na sua máquina. Chamamos esse processo de **Clone.**

- Você faz o que precisa fazer na sua própria cópia, e depois sobe de volta as alterações para o servidor com a versão original. Chamamos esse processo de **push.** Pronto, o servidor original foi atualizado com uma nova versão do código.

Veja alguns detalhes importantes sobre o versionamento:

**Branchs**

Geralmente, existe uma *branch* principal de desenvolvimento, e uma *branch* principal de **produção** chamada **Master.** Sendo assim, todos os envolvidos sabem que tudo o que está na **Master** é a versão oficial (que está no ambiente de produção, que é acessado pelos clientes, por exemplo).

**Merge**

Quando desenvolvimentos nas ramificações são finalizados, é possível fazer a **junção (merge)** com a branch original de desenvolvimento, e assim ela ficará atualizada com os novos recursos.

**Commit**

Quando você finaliza seu desenvolvimento ou alterações, você cria uma nova versão do código na sua estrutura local (**commit),** para depois fazer o **push.**

**Pull / Push**

Após um ou mais *commits,* você deve enviar as novas versões para a versão original (**push),** fazendo assim com que a versão original fique atualizada.

Se você tentar fazer um **push** antes de atualizar a sua versão local, o Git o impedirá. É preciso baixar a versão original (**pull**) antes.

Por isso, é sempre uma boa prática fazer o **pull**, principalmente quando existem muitos desenvolvedores trabalhando no mesmo código.

**Conflict**

Eventualmente pode acontecer um **conflito** de versões. Costuma acontecer quando duas pessoas sobem uma alteração em um mesmo lugar. O servidor atualiza, mas dará um alerta de conflitos. Esses conflitos precisam ser resolvidos manualmente.

**Stash / Pop**

Se você precisa parar com seu desenvolvimento por qualquer motivo, e não quer fazer *commit* das suas alterações, e também

não quer descartá-las, você pode “esconder” suas alterações ***(stash).***

O Git armazenará suas alterações em uma lista ***(stash list)*** e removerá da sua unidade de trabalho.

Futuramente você pode retomar essas alterações ou removê-las. Isso é bastante útil quando você precisa atualizar sua versão local antes de fazer um próximo *commit*, ou caso você descubra que alguém fez alterações importantes que gerarão impacto no seu trabalho.

Existem algumas ferramentas disponíveis que facilitam a visualização de alterações no GIT, sem precisar fazer tudo via linhas de código.

Naturalmente, se você está aprendendo, é uma boa prática efetuar as tarefas básicas para entender o processo.

# Introdução ao HTML

HTML é uma sigla para Hyper-Text Markup Language (Linguagem de marcação de hipertexto) – e é o “esqueleto” da internet.

O HTML é um conjunto de elementos escritos que informa ao navegador o que deve ou não ser exibido.

O HTML não tem programação, lógica ou cálculos. É apenas uma marcação. Uma marcação que instrui o navegador o que deve ser feito.

É como um mapa do tesouro, onde a linha tracejada é um caminho, um círculo é um a árvore, e o “X” é onde está a recompensa.

É mais ou menos dessa forma que os navegadores entendem o HTML.

Sem entrar em detalhes, o navegador entende duas grandes estruturas no HTML: A *head* e o *body*.

A *head* é onde fica todo o conteúdo lógico de estrutura do documento HTML.

O *body*, como diz o nome, é o corpo do documento. É a parte que o usuário tem acesso e interage.

## Head

A *head* é onde ficam as informações que são processados no início do carregamento e quem utiliza o navegador não tem acesso por meios convencionais a este conteúdo.

Na *head* podemos utilizar a tag *script*, para criar âncoras (*links*) para scripts como o *javascript*.

Também podemos utilizar *link* para ancorar estilos CSS externos.

Na *head* podemos também declarar *scripts* e estilos com conteúdo direto (e não apenas linkando documentos externos).

É aí também que definimos elementos utilizados em SEO (*Seach Engine Optimization*), como título, autor, descrição e palavra-chave, muito utilizados no mundo digital, bem como dados para redes sociais (conhecidos como Open Graph*).

**Open Graph é um padrão de nomenclatura para exibição de conteúdo social, como foto, link, url exibida, informações de autor, e outras informações*

## Body

O *body* é o corpo do documento: É a parte que o usuário manipula e visualiza de forma direta.

Um formulário que será preenchido, um botão pressionado, textos, animações, tudo isso fica dentro do *body*.

## DOM

O HTML completo recebe o nome de DOM (Document Object Model – Modelo de objeto do documento), é isso que nos permite manipular através de tecnologias como o Javascript e o CSS.

O DOM é um modelo de dados que transforma todas essas marcações em “objetos manipuláveis”.

Cada objeto do DOM pode ser acessado através de seu nome, bem como seus atributos.

O html define atributos nos objetos do DOM, por exemplo:

- Atributos dos links (tag: ‘a’)

*href* – Acrônimo para *Hyperlink Reference*, define a âncora

*title* – Define um título que será exibido quando o mouse estiver sobre o link

- Atributos das imagens (tag: ‘img’)

*src* – Acrônimo para *Source*, define a origem da imagem.

*title* – Texto exibido caso a imagem não seja encontrada

- Atributos gerais (utilizados em qualquer elementos

*id* – *Identity* e indica um identificador único do elementos

*class* – Define uma lista de classes CSS. Cada classe é passada utilizando seu nome e separadas por espaço.

*onclick* – Evento que dispara uma função em Javascript ao ser clicado

*style* – Atributo utilizado para passar estilo *inline*

Esses foram alguns exemplos de atributos de atributos dos objetos do DOM.

Para destacar os elementos que são mais comuns no DOM, temos:

- Seções (section)
- Estruturas de navegação (nav)
- Formulários (form)

- Parágrafos (p)
- Containers de divisão (div)
- Rodapé (footer)
- Âncoras de hyperlink (a)
- *Cabeçalho (header)* – não confundir com a *head* da raiz do *HTML*.

# HTML Básico

*Tags*

Tudo o que for escrito no HTML é escrito através de **tags.**

Uma tag tem 2 elementos: abertura e fechamento.

O elemento de abertura é escrito com um sinal de menor, o nome, e o sinal de maior assim:

*<nome>*

Na tag de abertura definimos os atributos, através do nome, um sinal de igual (=) e o valor dentro entre aspas, como nos exemplos:

*<nome atributo=”valor-do-atributo”>*

*<a href=”https://google.com” title=”Abre a página do Google”>*

O elemento de fechamento é escrito de forma semelhante, mas, com uma barra invertida antes do nome, dessa forma:

*</nome>*

* *

Exemplos:

* Isto é um parágrafo *

* Isto é outro parágrafo *

*Tags aninhadas*

As tags podem ser aninhadas (colocadas umas dentro das outras)

```
<!doctype html>
    <body>
    <header>
        <h2>Título da página</h2>
    </header>
        <section>
        <p>Aqui vai um parágrafo</p>
        </section>
    </body>
</html>
```

Você pode colocar quantas tags precisar.

O navegador cuida para que nada se perca e interpreta tudo corretamente (desde que seja construído sem erros, é claro).

*Atributos*

Toda tag no html permite atributos dos mais diversos tipos. Cada elemento possui alguns atributos próprios.

Existem atributos obrigatórios e opcionais em todos os elementos.

Você verá, quando iniciarmos no CSS, que TODO atributo pode ser usado para selecionar elementos. E é uma estratégia muito útil para organização e estrutura.

Atributos mais comuns:

- *Id:* Identificador único de um elemento. Não pode existir dois ids iguais na mesma página.

- *class:* Identifica uma classe. Um elemento pode ter várias classes e uma mesma classe pode aparecer em vários elementos.

- *href (Hyperlink reference):* Indica um link. Usado em elementos de âncora (a), link de css (no head) e outros.

- *src (source):* Indica o caminho de uma imagem, mídia, arquivo em geral. Utilizado em elementos diversos, como imagem (img), style (no head) e outros.

- *type:* Geralmente utilizado em elementos do tipo *input*, que vão dentro de um formulário (form). O type indica se é um campo texto, senha, número, e-mail, caixa de seleção, entre outros

- *data:* Este atributo representa o *dataset*, tem diversas utilidades, principalmente quando se trabalha com Javascript.

Ele funciona com um hífen e um nome. Este nome pode ser acessado e utilizado para fins específicos pelos programadores, ou até mesmo no próprio CSS. Exemplo:

<nav data-tipo=”usuario-logado”>...</nav>

# Como o navegador interpreta o HTML

O navegador interpreta todas as marcações escritas, na ordem de leitura *top-down-left-right*.

Ou seja, da esquerda para a direita, e de cima para baixo.

Em quase todos os *scripts* utilizados isso faz diferença.

No CSS, por exemplo, que depende da ordem de execução das regras, o impacto é considerável.

De igual modo no Javascript, o script é lido e interpretado pelo navegador na mesma ordem.

Porém, dependendo da estratégia de utilização e dos frameworks, não necessariamente as funcionalidades são executadas em ordem. Você vai entender as peculiaridades adiante.

# O que são e como funcionam as APIs

API é o acrônimo para *Application Programming Interface.*

Na web, API é conhecida como uma forma de acessar informações estruturadas via HTTP, tal como é o acesso a um site a partir de um navegador.

Uma API permite acessar dados diversos de um sistema, de modo que possamos manipular os dados, de acordo com as **permissões** definidas.

Elas também são utilizadas em contextos diversos para descentralizar responsabilidades de um sistema, de modo que seja possível construir várias aplicações que acessem a uma mesma API, para suas próprias necessidades.

É o caso de aplicações SPA (*Single Page Application*), desenvolvidas em *Angular* ou *React*, ou até mesmo aplicativos desenvolvidos em plataformas e/ou linguagens como Kotlin, React Native, Flutter, entre outras.

Uma API geralmente é acessada através de partes da URL chamadas de r**otas** ou **endpoints.**

Um **endpoint** é acessado através de um **método** de acesso HTTP (GET, POST, DELETE, PUT).

E podem ou não receber parâmetros. No caso do **GET,** os parâmetros são permitidos na URL. No exemplo abaixo, “2” é um parâmetro do **endpoint “**alunos”.

*http://dominio/alunos/2*

No caso do **POST**, podemos utilizar o **BODY** da requisição para passar dados mais complexos. Continue lendo para um exemplo prático da construção de uma API.

## Autenticação x Autorização

Uma API pode permitir acesso anônimo (ou seja, não é preciso nenhum tipo de autenticação para isso), ou exige alguma **autenticação.**

A **autenticação** de uma API consiste em fornecer dados (como usuário, e-mail, login e senha) de modo que o sistema reconheça este usuário e libere o acesso.

Além da autenticação, uma API também pode ter níveis de **autorização,** que podem definir:

- *Quais informações você tem acesso*

- *Como você pode acessar as informações*

- *Que tarefas você pode fazer na API (como inclusão, exclusão, edição de dados).*

Tudo depende das regras definidas pelo sistema.

## Formato JSON

Atualmente, o formato de dados mais comum utilizado no retorno de consultas em API´s é o JSON.

**JSON** é um padrão de comunicação baseada em objetos do **Javascript** que possibilita fácil manipulação de informação.

Há algum tempo atrás, as API´s e serviços externos tinham como padrão de comunicação o **XML**.

Ler um **XML** em uma API era um trabalho de difícil leitura, que demandava a construção de recursos específicos, conversão de dados, e uma série de questões que tornava o trabalho complexo e muitas vezes demorado.

A construção de uma API que retorna o formato JSON é relativamente simples nos dias atuais.

Praticamente todas as tecnologias e ferramentas já possuem suporte para este formato. Basta que na configuração da API (os *headers* do HTTP) seja definido o formato entregue, e a entrega do servidor seja de acordo.

Veja abaixo exemplos de um **endpoint** chamado **“alunos”** tendo o back-end em C# e o front-end em **Javascript**:

```
[HttpPost]
[Route("alunos")]
public IActionResult PostAluno([FromBody] AlunoModel options)
{
        var ativo = options.ativo;
        var alunoId = options.alunoId;
        var aluno = _alunoRepository.Get(alunoId);
        return View(aluno);
}

[HttpGet]
[Route("alunos/{alunoId}")]
public IActionResult PostAluno(int alunoId)
{
        var aluno = _alunoRepository.Get(alunoId);
        return View(aluno);
}
```

//Exemplo chamada POST em Javascript:

```
async function consultar() {
        var url = 'https://dominio/alunos';
        var options = {
        method: 'POST',
        body: JSON.stringfy({
        alunoId: 2,
        status: 'ativo'
        })
        };
        var response = await fetch(url, options);
        if (response.ok) {
        var jsonData = await response.json():
        //Neste exemplo, nosso request retorna um objeto JSON com
os              dados id, nome e idade.
        const {id, nome, idade} = jsonData;
        }
}
```

```
//Exemplo chamada GET em Javascript:
async function consultar() {
        var url = 'https://dominio/alunos';
        var options = {
        method: 'GET'
        };
        var response = await fetch(url, options);
        if (response.ok) {
        var jsonData = await response.json():
        const {id, nome, idade} = jsonData;
        }
}


//Exemplo de retorno:
{
        id: 2,
        nome: 'Pedro',
        idade: 23
}
```

# DNS – Sistema de Nomes

Quando acessamos um site,

# Glossário de tecnologias

No momento que este livro está sendo escrito, algumas tecnologias se destacam no desenvolvimento web e aplicações móveis.

Nesta sessão do livro, eu destaquei algumas das principais tecnologias que eu pessoalmente trabalho, e também outras que não fazem parte da minha “caixa de ferramentas”, mas também estão aí, disponíveis no mercado.

A intenção desta pequena lista é oferecer um norte sobre tais tecnologias, seja para fins de conhecimento ou para ajudar você a decidir um próximo investimento de aprendizado, curso ou projeto.

## .Net Core

O .Net Core é uma das principais tecnologias da Microsoft utilizada no desenvolvimento web. Consiste em uma versão evoluída e com novos recursos do .Net Framework.

O .Net Core permite que você trabalhe em ambientes Windows e Linux com abordagens atualizadas do mercado, bem como uma comunidade muito bem desenvolvida e cheia de informação. É possível desenvolver em .Net Core utilizando as linguagens C# ou VB.

## Entity Core

Entity Core é a um ORM para .Net Core e é bastante utilizado para realizar consultas a bancos de dados de forma facilitada. Sua estrutura e sintaxe permitem padronização e agilidade de desenvolvimento.

## Node JS

Node JS é um framework de back-end em Javasript que possibilita diversas abordagens. Entre elas, fornecer um ambiente de compilação e transpilação para tecnologias de front-end como React e Angular.

Atualmente, é possível construir aplicações completas e robustas em um servidor Node.

# Express JS

Express JS é uma ferramenta muito útil para a construção de API´s em Node. Com ele você consegue efetuar consulta à bancos de dados, construir rotas e endpoints de API´s e definir toda a estrutura do back-end do seu projeto.

# Next JS

Next JS é um framework de desenolvimento que utiliza o React como tecnologia de front-end.

O Next permite que você crie toda a estrutura de uma aplicação back-end e front-end. Possui algumas pecularidades para definição de rotas, parâmetros de URL entre outras características de estrutura cliente x servidor.

# Angular

Um dos principais frameworks de desenvolvimento front-end utilizados atualmente, o Angular é suportado pela Google e possui uma estrutura robusta baseada em modularização de componentes. Utiliza-se do Javascript (EcmaScript) e Typescript.

# React JS

O React é uma biblioteca Javascript suportada pelo Facebook e também é amplamente utilizado para construção de aplicações front-end de forma prática e rápida. O React é baseado em componentes e estados. Dominar o ciclo de vida dos componentes é o que garante projetos bem desenvolvidos.

Traz consigo uma nova linguagem de marcação conhecida como JSX, que se assemelha ao HTML, porém, permite que você mescle com a lógica da estrutura do desenvolvimento.

React tem por característica ser simples e de fácil implementação.

# React Native

O React Native é um framework de desenvolvimento mobile Android e iOS baseado em React e Node que possibilita um desenvolvimento muito próximo ao nativo de cada plataforma.

Ele oferece simplicidade e um ganho de tempo e dinheiro muito interessantes para desenvolvimento mobile.

# Expo CLI

O Expo CLI é um *client* de desenvolvimento em React Native que fornece uma gama muito grande de recursos e componentes próprios para desenvolvimento mobile.

A utilização do Expo Cli depende das necessidades do projeto e da familiaridade / gosto pessoal sobre a tecnologia.

# Cloud (Azure / Google Cloud / AWS)

A utilização da computação em nuvem é uma realidade no desenvolvimento e projetos de grandes empresas.

Utilização de nuvem como infraestrutura oferece diversas características importantes, como:

- *Escala de projetos*
- *Armazenamento*
- *Controle de custos e recursos*
- *Versionamento de código (GIT) e Pipeline através do Devops*
- *Integração contínua (CI / CD)*
- *Redundância de servidores*
- *Controle de serviços*
- *Controle de chaves secretas de serviços (secret keys)*

# Avalie este livro

Se você gostou da leitura e dos conhecimentos compartilhados nesse livro, compartilhe seu comentário e dê sua nota na Amazon.

Seu *feedback* é de grande ajuda para que possamos tornar nossos conteúdos melhores e mais objetivos a cada dia.

Saber a sua visão e opinião é de extrema importância para mim!

Além disso, recomende este livro para outras pessoas, que também podem ser beneficiadas com o aprendizado e prática citados aqui.

Se quiser falar diretamente comigo, sugerir algo, fazer um comentário, envie um e-mail para *pessoal@doriantorres.com.br*.

Minha sincera gratidão!

# Sobre o autor

Dorian é entusiasta do comportamento e mente humana.

Formado em Ciência da Computação, hoje atua como engenheiro de software, e analista de sistemas, coordenação de projetos de desenvolvimento de software e aplicativos e desenvolvimento de sistemas voltados para marketing digital.

Também desenvolve atividades como profissional da área de marketing, comunicação, artes e *copywriting*.

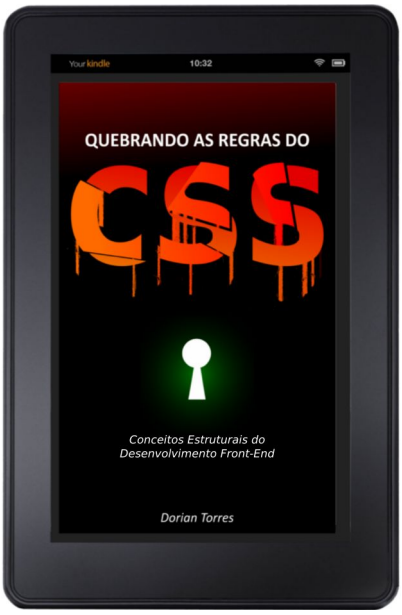
A escrita é parte de sua trajetória e sua vida desde criança.

Sua paixão pelo desenvolvimento humano faz com que consiga unir o melhor dos dois mundos: a visão analítica das Ciências Exatas com o comportamento das pessoas.

E assim, traduzir sua visão em livros como esse.

Estudante e praticante de PNL, curioso dos assuntos relacionados à mente humana, apreciador da Hipnose, estudante e pesquisador de tecnologias que colaboram com o comportamento humano e sua comunicação com o meio e autoconhecimento.

# Outros livros de Dorian Torres




**Quebrando as Regras do CSS**

*Conceitos Estruturais do Desenvolvimento Front-End*

*Aprenda a aplicar HTML5 e CSS3 em qualquer projeto front-end, mesmo que você seja um completo iniciante, em mais de 200 páginas de puro conhecimento e exemplos práticos.*

***Conheça o livro neste link***

**Como aprender mais e melhor**
*Um guia para acelerar o seu aprendizado*
*Categoria*: Aprendizado, PNL, autoajuda
Para conhecer este livro, é só clicar neste link

**CSS3: O guia completo do Flexbox**
*Categoria*: Desenvolvimento web e front-end
Para conhecer este livro, é só clicar neste link

**CSS3: Grid Layout sem segredos**
*Categoria*: Desenvolvimento web e front-end
Para conhecer este livro, é só clicar neste link

**CSS3: Cores, Gradiente e Geometria**
*Categoria*: Desenvolvimento web e front-end
Para conhecer este livro, é só clicar neste link

**10 Modelos de Vendas para Marketing Digital**
*É melhor começar agora!*
*Categoria*: Marketing digital e vendas
Para conhecer este livro, é só clicar neste link

**Como vender na internet sem segredos em 10 passos**
*Categoria*: Marketing digital
Para conhecer este livro, é só clicar neste link

**ASMR**
*A Bíblia do orgasmo mental*
*Categoria*: Relaxamento e ASMR

**Para conhecer este livro, é só clicar neste link**

*“Você é aquilo que faz repetidas vezes”*

*- Aristóteles, filósofo grego*