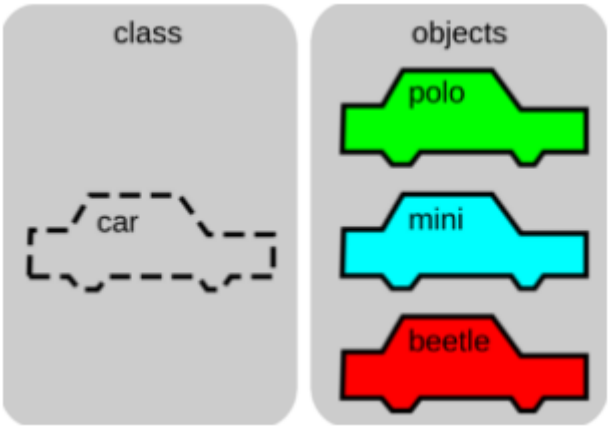
Vantagens de se utilizar TypeScript

- O TypeScript apresenta erros no momento da organização, enquanto o JavaScript, no tempo de execução.
- O TypeScript oferece as vantagens da composição estática discricionária: os tipos de TS podem ser adicionados a fatores, capacidades, propriedades e assim por diante.
- O TypeScript sustenta especificamente a composição estática. A composição estática pode ser valiosa para ajudar a arquivar capacidades, explicar a utilização e diminuir a sobrecarga psicológica (dicas de tipo de interface e obtenção de erros esperados de programação contínua).
- O TypeScript é executado em qualquer programa ou motor JavaScript.
- Ferramentas extraordinárias com IntelliSense que fornecem pistas dinâmicas como o código adicional.
- TypeScript ajuda na organização do código.
- TypeScript tem uma ideia de namespace ao caracterizar um módulo.
- As explicações do TypeScript podem ser discricionárias.
- TypeScript mantém interfaces.
- Os módulos gerenciadores do TypeScript oferecem um destaque entre outros insights do engenheiro IDE.
- O TypeScript tem uma documentação melhor para APIs que está em um estado de harmonia com o código-fonte. Algumas organizações relatam uma diminuição nos bugs quando mudam para o TypeScript.

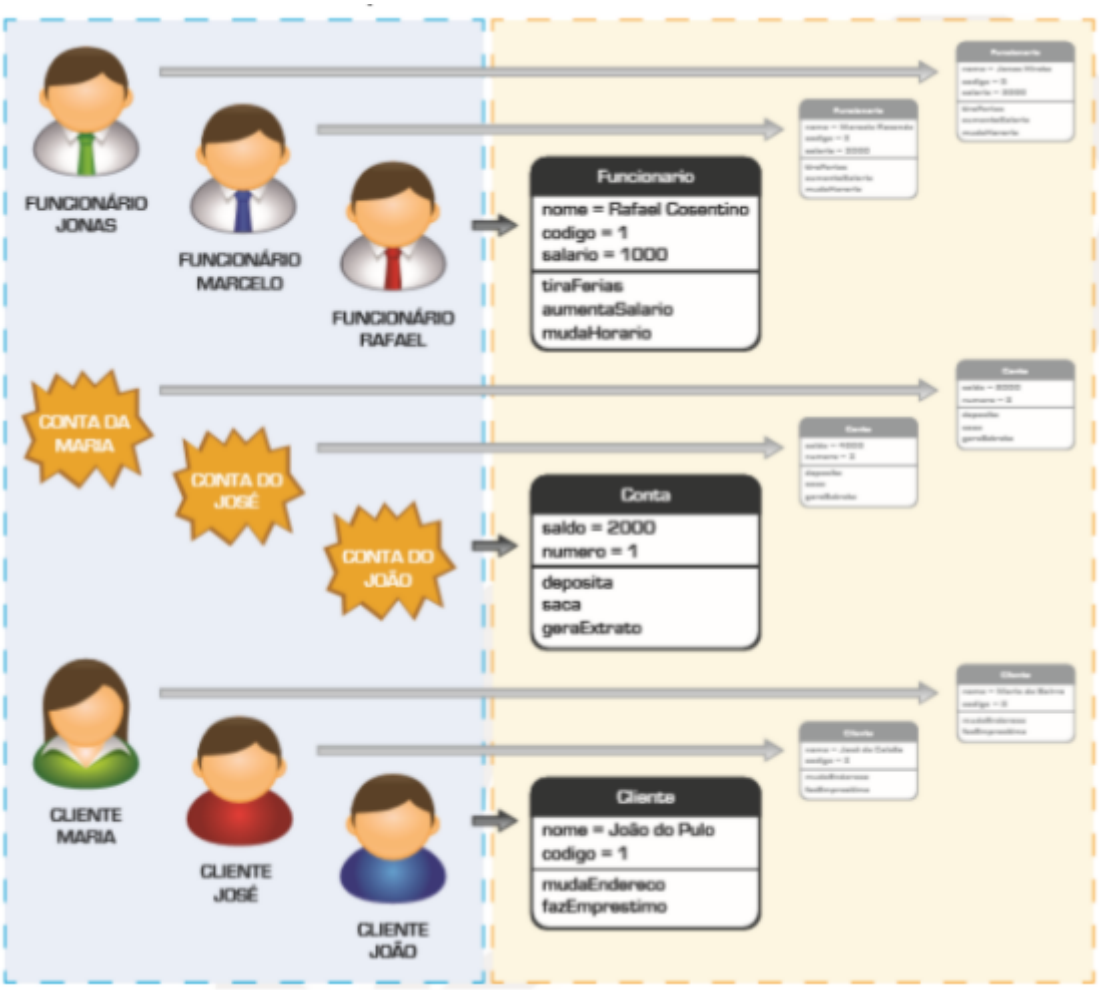
Classes e Objetos

Em javaScript os programas são escritos em pequenos pedaços separados, chamados de objetos. Objetos são pequenos programas que guardam dentro de si os dados – em suma, as variáveis – que precisam para executar suas tarefas. Os objetos também trazem em si, como sub-rotinas, as instruções para processar esses dados. As variáveis que um objeto guarda são chamadas de atributos, e as suas sub-rotinas são chamadas de métodos.

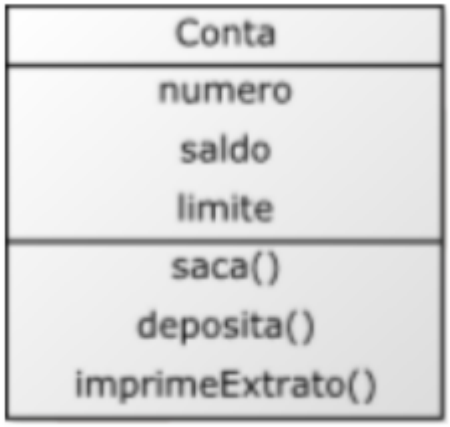
Vamos colocar um exemplo bem simples para entender esses conceitos. Pensem em um carro, este carro possui um motor, uma cor, portas, câmbio, etc. Ele também possui comportamentos que, provavelmente, foram o movo de sua compra, como acelerar, desacelerar, acender os faróis, buzinar e tocar música. Podemos dizer que o carro novo é um *objeto* , onde suas características são seus *atributos* (dados atrelados ao objeto) e seus comportamentos são ações ou *métodos*.
Ao mesmo tempo o seu carro, é apenas um dentre muitos outros dentro da loja, ou seja, seu carro pode ser classificado como um carro que o seu carro será uma *instância* dessa classe chamada carro.



Podemos dar um outro exemplo em um domínio bancário:



Podemos representar uma classe através de diagramas UML.



Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de *instanciação da classe* , como se estivéssemos usando esse molde (classe) para criar um objeto.
Bem, visto os conceitos fundamentais de orientação à objetos, agora vamos aplicar todos esses conceitos na prática utilizando a linguagem de programação javaScript. Colocamos uma classe genérica *Pessoa* , onde teria que guardar os vários pedaços do seu nome, então para isso teríamos que criar alguns atributos para a mesma, por exemplo:

```
class Pessoa {
    private  primeiroNome: string;
    private  ultimoNome: string;
    private  nomesDoMeio: string;
}
```

Criamos a classe Pessoa com três atributos, que será modelo para a criação de outros objetos. Explicando melhor, então criamos três atributos do po private que serve para deixar o atributo privado, ou seja, somente métodos da própria classe Pessoa que poderá acessá-lo e manipular o mesmo.
Mas não basta criar os atributos, temos que também dar algumas funcionalidades para os mesmos, para isso criamos um método para retornar o nome completo por exemplo.

```
class Pessoa {
    private  primeiroNome: string;
    private  ultimoNome: string;
    private  nomesDoMeio: string;
    public getNomeCompleto() {
        var nomeCompleto = this.primeiroNome + " "+  this.nomesDoMeio + " " +
this.ultimoNome;
        return nomeCompleto;
    }
}
```

A primeira linha, public getNomeCompleto(), especifica o método. Primeiro, declara-se, através da palavra-chave public, que o método é público – isto é, qualquer método, de qualquer classe, pode invocá-lo. O método retorna objetos do tipo string e se chama getNomeCompleto. O par de parênteses vazio significa que ele não recebe parâmetro algum.

O conteúdo do método vem entre um par de chaves. Dentro do método, declara-se uma nova variável, do tipo string, chamada nomeCompleto. Ao contrário de variáveis como primeiroNome, nomeCompleto não é um atributo, mas sim uma *variável local*. primeiroNome existirá desde a criação do objeto até sua retirada da memória, mas nomeCompleto só existirá enquanto o método getNomeCompleto() estiver sendo executado, e para cada chamada do método uma nova variável será criada.

A variável nomeCompleto recebe o resultado da concatenação de string. O sinal de atribuição é =, e a concatenação de string é feita através do operador +. Ao final, um ponto-e-vírgula sinaliza o fim deste comando. Abaixo, temos o comando return. Quando ele é invocado, o método termina e o valor que está à sua frente (no caso, o valor referenciado pela variável nomeCompleto) é retornado.
Até agora ainda não construímos nenhum objeto, para isto javaScript tem uma ferramenta chamada *construtor*. Geralmente os construtores tem o mesmo nome da classe. Esse construtor cria um novo objeto e este novo objeto é armazenado na variável *pessoa*.

```
Pessoa pessoa = new Pessoa();
```

Agora vamos criar uma pessoa chamada Francisco Pinho Nunes.

```
Pessoa pessoa = new Pessoa();
pessoa.primeiroNome = “Francisco”;
pessoa.nomeDoMeio = “Pinho”;
pessoa.ultimoNome = “Nunes”;
console.log(pessoa.getNomeCompleto());
```

No entanto, isto não é possível porque os atributos são privados. Apenas os métodos da classe *Pessoa* podem acessá-los. Isso pode ser solucionado de várias maneiras, e uma das mais elegantes é criando o nosso próprio construtor, como abaixo:

```
function Pessoa(primeiro, meio, ultimo ) {
    primeiroNome = primeiro;
    ultimoNome = ultimo;
    nomeDoMeio = meio;
}
```

A declaração do construtor é sempre o nome da classe seguido pela lista de parâmetros. A palavra *public* indica que o construtor é público, de modo que pode ser invocado por qualquer classe. Um ponto importante sobre construtores é que eles não criam nem retornam objetos; quem faz isso é a palavra reservada *new*. O construtor apenas executa algum procedimento sobre o objeto criado pelo comando *new*. Este construtor, no caso, recebe os nomes como parâmetros e os atribui aos atributos.

```
Pessoa pessoa = new Pessoa( "Francisco", "Pinho", "Nunes" );
```

Só para idenficar, vejamos como ficou a classe *Pessoa* completa:

```
    class  Pessoa
    {
        primeiroNome:string;
        nomeDoMeio:string;
        ultimoNome:string;
        constructor(primeiroNome:string, nomeDoMeio:string, ultimoNome:string)
        {
```

```
            this.primeiroNome = primeiroNome;
            this.nomeDoMeio = nomeDoMeio;
            this.ultimoNome = ultimoNome;
        }
        public getNomeCompleto() {
            var nomeCompleto = this.primeiroNome + " "+  this.nomeDoMeio + " " +
this.ultimoNome;
            return nomeCompleto;
        }
    }
```

E a classe ProgramaNome, que utiliza a classe Pessoa para gerar um nome completo a partir das partes.

```
Pessoa pessoa = new Pessoa("Francisco","Pinho","Nunes");
console.log(pessoa.getNomeCompleto());
```

Objeto this

A palavra reservada *this* faz referência ao próprio objeto, quando usado dentro de um método, por exemplo.
No código-fonte da classe Aeronave, vamos criar um método chamado alterarTotalAssentos, que receberá um argumento com o novo número de passageiros a ser atribuído à variável de instância totalAssentos.

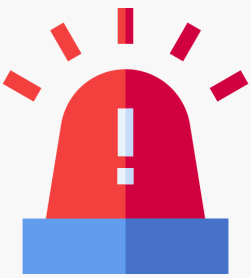
```
class Aeronave {
    totalAssentos:number;
    assentosReservados:number;

    reservarAssentos(assentos:number):void {
        var assentosReservados =+ assentos;
    }

    calcularAssentosDisponiveis():number {
    return  this.totalAssentos - this.assentosReservados;
    }

     alterarTotalAssentos(totalAssentos:number):void {
    this.totalAssentos = totalAssentos;
    }
}
```

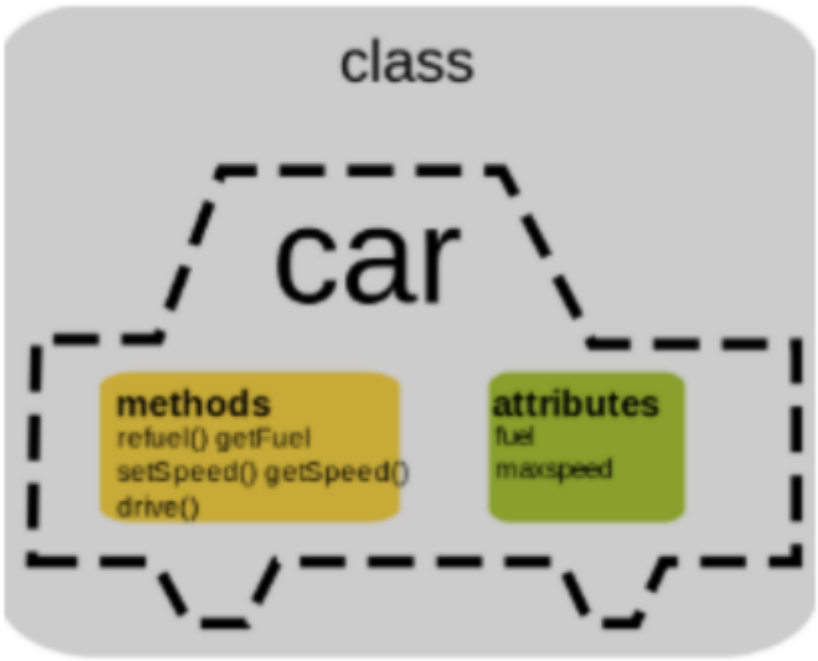
Estamos dizendo que queremos atribuir o valor da variável local à variável de instância. Quando usamos *this* , estamos deixando essa informação explícita.



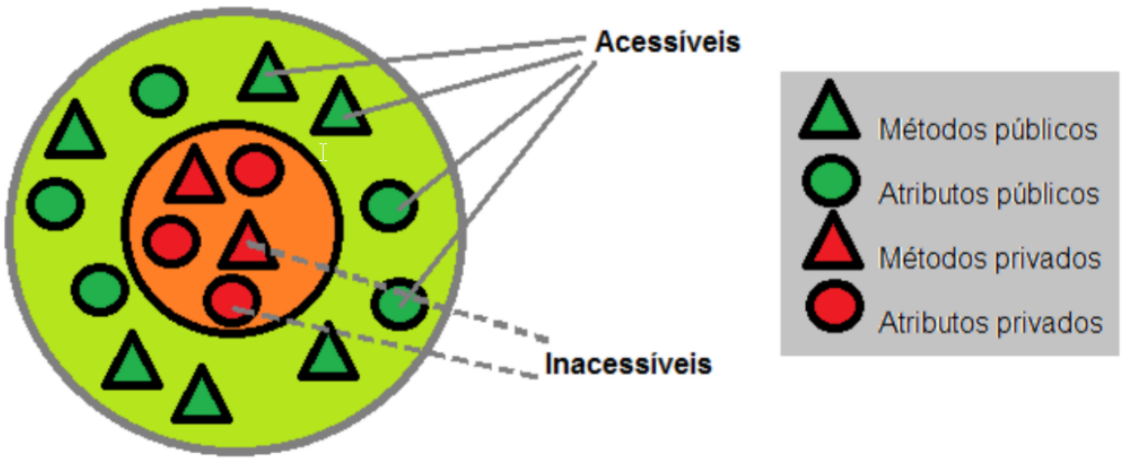
Alerta de BSM: ***Sempre importante utilizar a comunicação para a leitura dos materias e para perguntar para o instrutor caso tenha duvida***

Encapsulamento

Vamos voltar ao exemplo que demos do carro:



Os métodos do carro, como acelerar, podem usar atributos e outros métodos do carro como o tanque de gasolina e o mecanismo de injeção de combustível, respectivamente, uma vez que acelerar gasta combustível.
Mas alguns métodos e atributos não podem ser permitidos qualquer tipo de alteração, ou seja, eles não são visíveis para fora do carro.
Na POO, um atributo ou método que não é visível de fora do próprio objeto é chamado de " *privado* " e quando é visível, é chamado de " *público* ".



Ler ou alterar um atributo encapsulado pode ser feito a partir de *geters e seters* (colocar referência).
Esse *encapsulamento* de atributos e métodos impede o chamado *vazamento de escopo* , onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe. Isso evita a confusão do uso de variáveis globais no programa, deixando mais fácil de idenficar em qual estado cada variável vai estar a cada momento do programa, já que a restrição de acesso nos permite idenficar quem consegue

modificá-la.
Exemplo:

```
class Carro {

    private velocidade: number;
    private modelo: string;
    private MecanismoAceleracao: MecanismoAceleracao;
    private cor:string;

    /* Repare que o mecanismo de aceleração é inserido no carro ao ser construí
    public Carro(String modelo, MecanismoAceleracao mecanismoAceleracao) {
        this.modelo = modelo;
        this.mecanismoAceleracao = mecanismoAceleracao; this.velocidade = 0; }
    public void acelerar() {
        this.mecanismoAceleracao.acelerar(); }
    public void frear() { /* código do carro para frear */

    constructor(modelo: string, mecanismoAceleracao: MecanismoAceleracao,
cor:string) {
        this.modelo = modelo;
        this.MecanismoAceleracao = mecanismoAceleracao;
        this.cor = cor;
    }
    acelerar():void {
        this.mecanismoAceleracao.acelerar();
    }
    frear():void {/* código do carro para frear */}
    acenderFarol():void {/* código do carro para acender o farol */}
    getVelocidade():number {
        return this.velocidade
    }
    setVelocidade():void {
        /* código para alterar a velocidade do carro */ /* Como só o próprio
carro*/
    }
    getModelo() {
         return this.modelo;
         }
    getCor() {
        return this.cor;
    }
    /* podemos mudar a cor do carro quando quisermos */

    setCor(cor:string) {
        this.cor = cor;
    }

}
```