PROGRAMAÇÃO WEB CLIENTE E SERVIDOR

UFCD(s) 5414 - 5417

GUIA DE LABORATÓRIO 5.1
POO EM JAVASCRIPT

(ßeta)

OBJECTIVOS

- Estudar os mecanismos que dão suporte a Programação Orientada por Objectos (POO) em JavaScript
- Perceber o modelo pseudoclássico para POO, assente em protótipos e herança prototipal
- Aprender a programar com "classes" ES6
- Aprender uma alternativa ao modelo clássico para POO: o modelo funcional, sem classes

INSTRUÇÕES

PARTE I – SUMÁRIO EXECUTIVO

1. Neste laboratório vamos estudar Programação Orientada por Objectos (POO) em JavaScript. Vamos ver que **modelos de POO** JavaScript suporta e, para cada um destes modelos, quais os **mecanismos da linguagem** mais apropriados para programar de acordo com o que cada um desses modelos prescreve.

Os modelos e mecanismos que aqui vamos estudar são os seguintes:

Modelo 1 - Pseudoclássico: Protótipos e Herança Prototipal

Modelo 2 - Clássico: Classes ES6/ES2015

Modelo 3 - Sem Classes e Funcional: Fábricas de Objectos c/ "Herança" Funcional

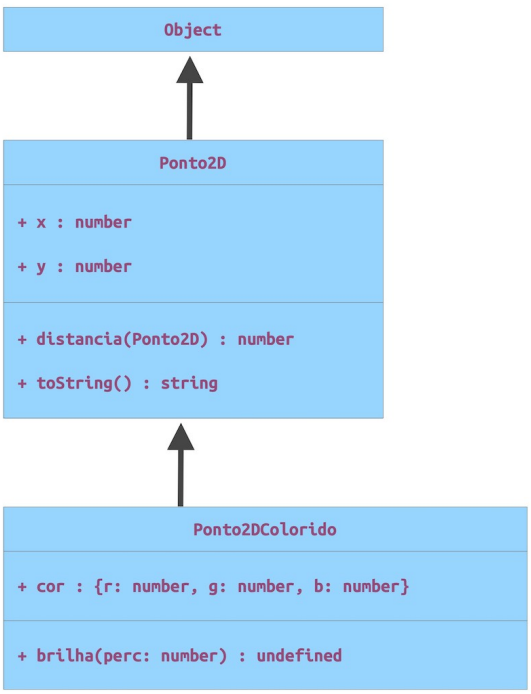
Seguem-se as **conclusões** e **recomendações** do laboratório, dirigidas especialmente aos "apressados" que não pretendem seguir o laboratório todo até ao fim, e que apenas querem saber qual é receita que devem utilizar:

1. Sempre que possível, devemos utilizar o modelo 3, uma vez que os os outros dois impõem uma organização muito rígida ao código, o que dificulta a sua manutenção e extensibilidade, além de que não oferecem privacidade (à data do laboratório, Julho de 2020). O modelo 3 é também mais simples e fácil de compreender, uma vez que é uma aplicação prática dos conceitos da programação funcional, encaixando-se melhor nesta metodologia de programação. Finalmente, o modelo 3 não utiliza *new* nem *this*, evitando, assim, toda uma categoria de erros e complicações que estas palavras-reservadas tendem a introduzir.

2. Sempre que não for possível, ou viável, utilizar o modelo 3, recomenda-se então a utilização do modelo 2, de modo a tirar partido das novas classes ES6. Os dois modelos "clássicos" são muito populares, mas, com a chegada do ES6 em 2015, muitas bibliotecas e frameworks passaram a definir APIs assentes em classes ES6. Como veremos, o modelo 2 também utiliza protótipos e herança prototipal, usufruindo, portanto, das vantagens do modelo 1 (eg, herança retroactiva). As classes como que escondem este pormenor. Tendo nós consciência disto, então mais vale tirar partido da conveniência sintática oferecida pelas classes, o que leva a código mais expressivo, legível e fácil de alterar. Além disto, "para o bem e para o mal", o modelo 2 está mais próximo dos modelos utilizados em linguagens como Java, C#, C++ ou Python.

PARTE II – REVISÕES SOBRE OBJECTOS LITERAIS EM JAVASCRIPT

2. Todos os passos do laboratório assentam num mesmo exemplo, a representação de pontos num eixo cartesiano.



Programação Orientada por Objectos (POO) consiste numa metodologia de programação que utiliza ***objectos*** *- uma agregação de atributos e operações numa só unidade - para representar informação concreta sobre os conceitos que o software pretende representar. Esses conceitos (eg, cliente, produto, encomenda, livro, ficheiro, ligação, device driver, local, utilizador, etc.) são definidos através de um qualquer mecanismo da linguagem, tipicamente, através de* ***classes****, um mecanismo muito popular em linguagens com suporte nativo para POO. Uma* ***classe****, ou mecanismo equivalente, define como é que os objectos são criados, que informação "carregam" consigo e que operações suportam. Cada objecto representa uma "criação" da classe, ie, uma* ***instância*** *da classe. Uma classe, não só actua como que uma especificação dos objectos dessa classe, como também, funciona como um* ***modelo*** *(ie, um "template") através do qual criamos os objectos. Na maioria das linguagens que suportam classes, uma classe é também um* ***tipo de dados****.*

Assim, temos:

- Pontos 2D: pontos simples, contendo apenas as respectivas coordenadas - X e Y - e algumas operações relacionadas
- Pontos 2D coloridos: estes são pontos 2D que contêm também uma representação da cor, em termos das propriedades RGB, juntamente com uma ou outra operação especializada neste tipo de pontos.

3. Como primeira abordagem, vamos utilizar objectos literais e funções globais para implementar as operações pretendidas:

```
const ponto1 = {x: 10, y: -20};
const ponto2 = {x: -1, y: 40};

console.log("ponto: %o", ponto1);
console.log("ponto: %o", ponto2);
console.log("(X, Y) -> (%d, %d)", ponto1.x, ponto1.y);

function distancia(p1, p2) {
    const pow = Math.pow;
    const xDistSq = pow(p1.x - p2.x, 2);
    const yDistSq = pow(p1.y - p2.y, 2);
    return Math.sqrt(xDistSq + yDistSq);
}

function toString(ponto) {
    return `<${ponto.x}, ${ponto.y}>`;
}
```

NOTA: Não se esqueça de testar o código apropriadamente, agora, e sempre que, ao longo do laboratório, achar apropriado. Alguns passos do laboratório, mas não todos, incluem instruções `console.log`. No entanto, sinta-se à vontade para introduzir as suas instruções de *log* sempre que achar conveniente testar ou inspeccionar algo.

4. Vamos agora definir outros objectos: uma cor e um ponto com cor:

```
const cor1 = {r: 221, g: 101, b: 23};

let ponto3 = {
    x: 20,
    y: 15,
    cor: {r: 221, g: 101, b: 23}
};

console.log(toString(ponto3), ponto3.cor);
```

5. Como definir um `toString` para aceitar pontos e cores? Como verificar se o objecto em questão é um ponto ou uma cor? Pelo tipo? Não, porque o tipo é `Object`:

```
console.log(typeof ponto1, typeof ponto2, typeof ponto3, typeof cor1);
```

6. Podemos usar ponto `instanceof` para testar se um objecto é um ponto o uma cor? Não, porque:

a) Não definimos construtor; objectos literais são do "tipo" `Object`

b) `instanceof` não funciona bem em JavaScript, conforme veremos.

Introduza:

```
console.log(ponto1 instanceof Object, ponto1.constructor.name);
```

7. Solução: múltiplas implementações com nomes diferentes. Sim, não é uma solução agradável, mas JavaScript não suporta sobrecarga de funções (*function overloading*):

```
function toStringCor(cor) {
    return `r: ${cor.r} g: ${cor.g} b: ${cor.b}`;
}

function toStringPonto(ponto) {
    let pontoStr = `<${ponto.x}, ${ponto.y}>`;
    if (ponto.hasOwnProperty('cor')) {
        pontoStr += ` (${toStringCor(ponto.cor)})`;
    }
    return pontoStr;
}

ponto3.cor = {r: 200, g: 100, b: 189};

console.log(toStringCor(ponto3.cor));
console.log(toStringPonto(ponto3));
```

PARTE III – PROPRIEDADE THIS E OPERADOR NEW

8. Ainda antes de introduzirmos protótipos, vamos primeiro falar da palavra-reservada **`this`** e de funções construtoras. Para tal, vamos desviar-nos um pouco do exemplo anterior. Introduza:

```
const pessoa = {
    nome: "Alberto",
    apelido: "Alves",
    nomeCompleto() {
        return `${this.nome} ${this.apelido}`
    }
}
```

*O propósito geral da propriedade **this** é providenciar às funções que estão associadas a um objecto, ie, aos seus **métodos**, um acesso a esse mesmo objecto. Considere o objecto `obj1` e o respectivo método `met1`. O **this** permite que `met1`, quando invocado na forma `obj1.met1()`, aceda a `obj1`.*
*Na realidade, em JavaScript, o **this** é uma propriedade de um determinado contexto (scope) de execução (contexto global ou contexto local de uma função). Em **modo não-rigoroso** (non-strict), o **this** é sempre uma referência para um objecto; em **modo rigoroso**, pode ser um objecto ou um outro valor qualquer, tipicamente **undefined**.*

*No tratamento dado à palavra-reservada **this**, o comportamento do JavaScript é um pouco diferente do de outras linguagens. Além de que varia de modo rigoroso (strict) para não-rigoroso. O valor de **this** é **dinamicamente** determinado (ou seja, não é, como acontece em Java ou C#, determinado em **tempo de compilação**, mas sim em **tempo de execução**). De facto, na maioria do casos, o valor de **this** é determinado pelo modo como uma função é invocada: se invocada como função global, se invocada através de um objeto e do operador `.` , se invocada através do operador **new**, se invocada com `call`, `apply` ou `bind`.*

A propriedade `nomeCompleto` também poderia ter sido definida da seguinte forma:

```
const pessoa = {
  ...
  nomeCompleto: function() {
      return `${this.nome} ${this.apelido}`;
  }
}
```

*Não poderíamos, no entanto, utilizar uma "arrow function" uma vez que estas vão buscar o objecto **this** ao contexto mais local do objecto literal que está a ser definido o que, neste caso, é o contexto global, ou seja, o objecto **globalThis**.*

```
const pessoa = {
  ...
  nomeCompleto: () => `${this.nome} ${this.apelido}`;
}
pessoa.nomeCompleto();  // "undefined undefined"
```

*Quando falarmos de construtores, veremos que este tratamento diferente dado à palavra-reservada **this** pelas "arrow functions" terá as suas vantagens, ainda que seja confuso **this** referir-se a coisas diferentes consoante o tipo de função.*

9. Agora teste com:

```
pessoa.nomeCompleto()
```

10. A criação de uma pessoa pode ser personalizada. Defina:

```
function pessoa(nome, apelido) {
    return {
        nome: `${nome[0].toUpperCase()}${nome.slice(1).toLowerCase()}`,
        apelido: `${apelido[0].toUpperCase()}${apelido .slice(1).toLowerCase()}`,
        nomeCompleto() {
            return `${this.nome} ${this.apelido}`;
        }
```

```
    };
}
```

11. Teste com:

```
const pessoa1 = pessoa('alberto', 'alves');
const pessoa2 = pessoa('armando', 'almeida');
console.log(pessoa1.nomeCompleto());
console.log(pessoa2.nomeCompleto());
pessoa('arnaldo', 'antunes').nomeCompleto();
```

> *Funções como estas são designadas de **construtores**. Um construtor tem como missão criar e inicializar um objecto a partir de informação válida, deixando-o num estado que cumpre os requisitos especificados para objectos deste tipo. Como consequência desta missão, o construtor deve validar a informação utilizada para inicializar o objecto.*
> *Note que esta função é utilizado para criar objectos, mas não os identifica, isto é, se tentarmos inspeccionar os objectos criados estes continuam a ser do tipo `Object`.*

12. Os objectos continuam a ser (apenas) do "tipo de dados" `Object`:

```
console.log(typeof pessoa1, pessoa1 instanceof pessoa);
```

13. A propriedade ***`this`*** pode também ser utilizada dentro de uma função. Vamos redefinir `pessoa`:

```
function pessoa(nome, apelido) {
    this.nome = `${nome[0].toUpperCase()}${nome.slice(1).toLowerCase()}`;
    this.apelido = `${apelido[0].toUpperCase()}${apelido.slice(1).toLowerCase()}`;
    this.nomeCompleto = function() {
        return `${this.nome} ${this.apelido}`;
    };
    return this;
}
```

> *Por omissão, em modo 'non-strict' a propriedade **`this`** refere-se ao objecto global: objecto `window`, no browser; objecto `global`, em Node.js.*
> *Em modo strict, a propriedade **`this`** começa por ter o valor **`undefined`**. Daí o erro que se obtém quando se executa função com `'use strict';` no início da mesma.*
> *O ES6 trouxe uma uma nova propriedade, designada de **`globalThis`**, que mapeia para o objecto global correcto em cada uma das implementações de JavaScript Ou seja, **`globalThis === global`** em Node.js, ao passo que, no browser, temos **`globalThis === window`**.*

14. Volte a definir `pessoa1` como em cima. O que sucedeu?

15. Execute agora:

```
console.log(nome, apelido, nomeCompleto());
console.log(globalThis.nomeCompleto());
```

16. Logo no início da função, acrescente a string `'use strict';` (não se esqueça do ponto-e-vírgula):

```
function pessoa(nome, apelido) {
    'use strict';
    // ...
}
```

17. Volte a testar. O que sucedeu? Porquê?

18. Precisamos de garantir que ***`this`*** é um objecto novo e não ***`globalThis`***. É aqui que entra o

operador ***new***. Vamos redefinir o construtor:

```
function Pessoa(nome, apelido) {
    'use strict';
    this.nome = `${nome[0].toUpperCase()}${nome.slice(1).toLowerCase()}`;
    this.apelido = `${apelido[0].toUpperCase()}${apelido .slice(1).toLowerCase()}`;
    this.nomeCompleto = function() {
        return `${this.nome} ${this.apelido}`;
    };
}
const pessoa1 = new Pessoa('alberto', 'alves');
const pessoa2 = new Pessoa('armando', 'almeida');
console.log(pessoa1.nomeCompleto());
console.log(pessoa2.nomeCompleto());
console.log(pessoa1.constructor.name);
console.log(pessoa1 instanceof Pessoa);
```

*O nome capitalizado Pessoa é uma convenção adoptada em JavaScript para assinalar que esta função é um construtor, que usa a propriedade **this**, e, como tal, deve ser invocado com **new**. Quando invocado com **new**, o construtor devolve sempre o novo objecto, logo não é necessário o **return** final. Além de criar o objecto, associá-lo à propriedade **this**, e garantir que este é devolvido quando o construtor termina, o operador **new** também preenche a propriedade `construtor` do objecto com uma referência para a função Pessoa. Com isto, **instanceof** consegue verificar que o objecto é do tipo Pessoa.*
Como esta função construtora contém toda a definição dos objectos, incluindo a definição dos métodos, ela acaba por desempenhar o papel que as classes desempenham em linguagens como Java e C#.

19. Bem melhor! No entanto, os problemas com ***this*** ainda não terminaram. Em JavaScript, é muito comum querermos "capturar" uma referência para um método e guardá-la nalgum local. Experimente:

```
const nomeComp = pessoa1.nomeCompleto;
nomeComp();   // ooops, this é undefined
```

20. Como resolver isto? Não vamos aprofundar este aspecto, mas isto resolve-se com funções como `apply` ou `call`. Uma alternativa melhor passa por utilizar *closures*, conforme veremos no modelo 3.

21. Outro aspecto que interessa mencionar é que todos os objectos criados por `Pessoa` possuem a sua versão de `nomeCompleto` (ainda que o código seja o mesmo). Ou seja, o JavaScript cria um *function object* `nomeCompleto` por cada objecto `Pessoa` criado:

```
console.log(pessoa1.nomeCompleto === pessoa2.nomeCompleto);
```

22. Antes de regressarmos ao exemplo dos pontos, e de entrarmos então no modelo pseudoclássico, é de notar que já podemos utilizar uma "arrow function" para definir o método `nomeCompleto`:

```
function Pessoa(nome, apelido) {
    // ...
    this.nomeCompleto = () => `${this.nome} ${this.apelido}`;
}
```

PARTE IV – MODELO PSEUDOCLÁSSICO: POO COM PROTÓTIPOS E HERANÇA PROTOTIPAL

23. Voltando ao exemplo da representação de pontos 2D, crie o seguinte construtor:

```
function Ponto2D(x, y) {
    this.x = x;
    this.y = y;
}
```

Todos os objectos nascem com uma ligação a um objecto designado de ***protótipo*** *(*). Sempre que tentamos aceder a uma propriedade de um objecto, o JavaScript procura-a primeiro no próprio objecto. Se não existir, a propriedade é então procurada no protótipo. Caso não exista, é procurada no protótipo do protótipo, e assim sucessivamente até chegarmos a Object. Só se não existir em Object, após ter percorrido toda a cadeia de protótipos, é que o JavaScript declara que a propriedade não existe, devolvendo* ***undefined****.*
Objectos criados com o mesmo construtor partilham o mesmo protótipo. Deste modo, "tudo" o que adicionarmos ao protótipo fica acessível a todos os objectos. Então, se associarmos métodos ao protótipo, atendendo à forma como as propriedades são procuradas ao longo da cadeia de protótipos, então estes ficam disponíveis para todos os objectos.
Conforme veremos, no ***Modelo Pseudoclássico*** *não criamos objectos a partir de classes, mas sim a partir de outros objectos, os protótipos. Estes armazenam um conjunto de propriedades partilhadas por todos os objectos*

(*): Dado um objecto qualquer, o seu protótipo está associado a uma propriedade interna designada de `[[Prototype]]`*. Se necessitarmos de aceder ao protótipo, ao invés de aceder directamente a esta propriedade (o que não é possível), utilizamos o método* `Object.getPrototypeOf(objecto)`*. Conforme veremos, o objecto que se obtém através de* `getPrototypeOf` *também pode ser obtido através da propriedade* `prototype` *da função construtora que construiu o objecto em questão. Por exemplo, dado o array* `arr1`*, então verifica-se o seguinte:*

```
Object.getPrototypeOf(arr1) === Array.prototype
```

24. Em vez de cada objecto "carregar" consigo os métodos, vamos então associá-los ao protótipo:

```
Ponto2D.prototype.toString = function() {
    return `<${this.x}, ${this.y}>`;
};

Ponto2D.prototype.distancia = function(p2) {
    const pow = Math.pow;
    const xDistSq = pow(this.x - p2.x, 2);
    const yDistSq = pow(this.y - p2.y, 2);
    return Math.sqrt(xDistSq + yDistSq);
};
```

25. Vamos já criar dois pontos para testar este código:

```
const ponto1 = new Ponto2D(10, -20);
const ponto2 = new Ponto2D(-1, 40);
console.log(ponto1, ponto2);
console.log(ponto1.toString(), ponto2.toString());
```

Cada objecto tem o seu `x`*,* `y` *mas todos partilham* `toString`*.*

```
console.log(ponto1.toString === ponto2.toString)
```

26. `Ponto2D`, `ponto1` e `ponto2` têm uma referência para um mesmo objecto, o protótipo dos objectos criados com `Ponto2D`:

```
console.log(Ponto2D.prototype === Object.getPrototypeOf(ponto1));
console.log(Ponto2D.prototype === Object.getPrototypeOf(ponto2));
```

NOTA: Ver diagrama em baixo com as ligações entre `ponto1`, `Ponto2D.prototype`, `Ponto2D`, `Object`, `Object.prototype`.

27. Vamos acrescentar propriedades a `ponto1` apenas:

```
ponto1.cor = {r: 21, g: 19, b: 78};
console.log(ponto1);
console.log(ponto2);
```

28. Vamos acrescentar propriedades a todos os pontos:

```
Ponto2D.prototype.visivel = true;
console.log(ponto1, ponto1.visivel);
console.log(ponto2, ponto2.visivel);
```

Todos os pontos começam com esta propriedade (`visivel`) no protótipo e com o mesmo valor. Porém, alterações à propriedade são "locais" a cada objecto. Ou seja, quando se altera o valor dessa propriedade para um objecto, altera-se apenas para esse objecto.

29. Modifique o valor da propriedade anterior apenas para `ponto1`:

```
ponto1.visivel = false;
console.log(ponto1.visivel, ponto2.visivel);
```

30. Vamos acrescentar o equivalente a uma variável estática noutras linguagens (ie, aquilo que em Java ou C# seria uma variável definida com **`static`**, ou seja, uma variável de classe):

```
Ponto2D.versao = "1.0";
```

31. Note que o acesso a esta variável estática deve ser sempre feito através do construtor:

```
console.log(Ponto2D.versao);
console.log(ponto1.versao);     // errado
```

32. E agora vamos utilizar um "método estático" para definir um construtor alternativo. Este aceita uma string na forma `<numero[.numero],[ ]*numero[.numero]>`:

```
Ponto2D.fromString = function(pontoStr) {
    if (!/^<[\-+]?\d+(\.\d+)?,\s*[\-+]?\d+(\.\d+)?>$/.test(pontoStr)) {
        throw new Error(`Invalid string value for point ${pontoStr}`);
    }
    let [xStr, yStr] = pontoStr.split(',');
```

```
        xStr = xStr.trim().slice(1);
        yStr = yStr.trim().slice(1);
        return new Ponto2D(parseFloat(xStr), parseFloat(yStr));
    }
```

33. E agora testamos com:

```
const ponto3 = Ponto2D.fromString("<5.7, 23>");
console.log(ponto3);
const ponto4 = new Ponto2D(5.7, 23);
// comparação seguinte é falsa porque são objectos diferentes
// apesar de terem o mesmo conteúdo
console.log(ponto3 === ponto4);

// mas as seguintes já dão true
console.log(ponto3.toString() === ponto4.toString());
console.log(JSON.stringify(ponto3) === JSON.stringify(ponto4));
```

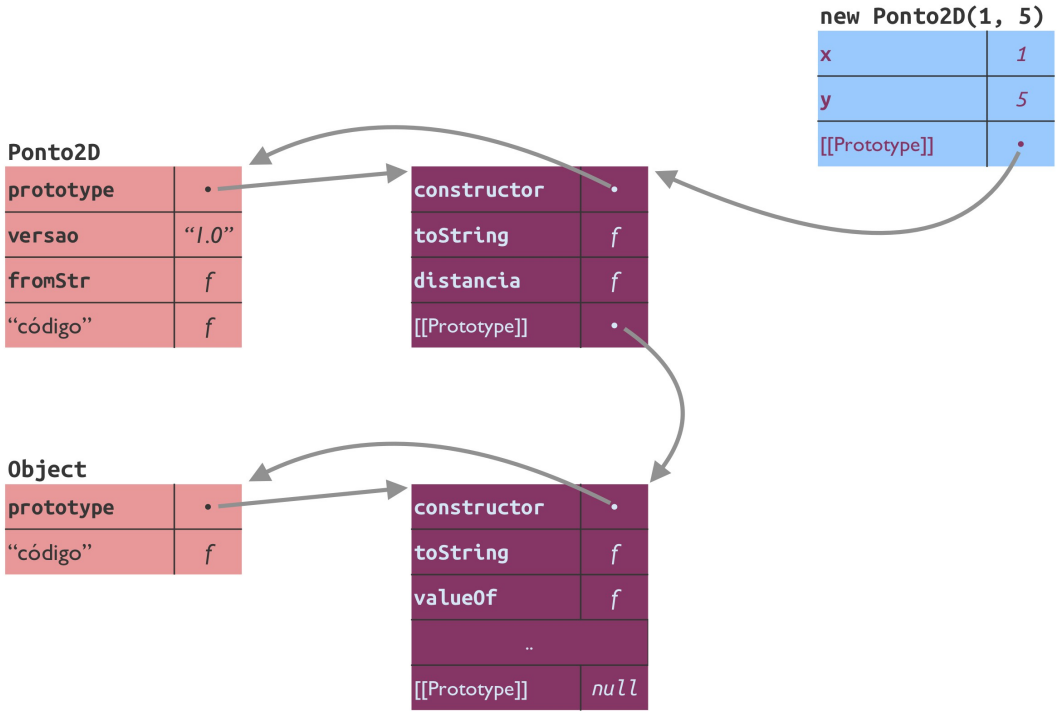
Quando os operados do operador === são objectos, este compara as referências dos objectos e não conteúdo. Ou seja, a comparação só dá true se os operandos se referirem ao mesmo objecto.
Temos de ser nós, programadores que estabelecemos a estrutura e operações dos nossos objectos, a definir como é que devem ser comparados. Uma estratégia habitual consiste em convertê-los para texto e depois comparar as strings resultantes. Quando não temos um método para converter os nossos objectos para strings (nem os pretendemos definir), podemos converter ambos os objectos para JSON e depois comparar as strings resultantes. JSON (JavaScript Object Notation) é um formato de dados para troca de informação entre processos. Actualmente (Julho de 2020) é o formato de dados mais utilizado em todo o mundo. Temos que ter em atenção, contudo, que nem todos os objectos podem ser transformados em JSON.
Consultar: https://www.json.org/json-en.html
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

Uma definição mais provável para o construtor `Ponto2D.fromString` seria a seguinte:

```
Ponto2D.fromString = (function() {
    const pontoStrRegex = /^<[\-+]?\d+(\.\d+)?,\s*[\-+]?\d+(\.\d+)?>$/;
    return function(pontoStr) {
        if (!pontoStrRegex.test(pontoStr)) {
            throw new Error(`Invalid string value for point ${pontoStr}`);
        }
        let [xStr, yStr] = pontoStr.split(',');
        xStr = xStr.trim().slice(1);
        yStr = yStr.trim().slice(1);
        return new Ponto2D(parseFloat(xStr), parseFloat(yStr));
    }
}());
```

Que "mecanismos", relacionados com programação funcional em JavaScript, são utilizados aqui? Quais as vantagens desta definição? Pista: qualquer expressão regular em JavaScript é compilada para código mais eficiente...

34. A imagem seguinte ilustra com um diagrama os objectos envolvidos na definição de `Ponto2D`. Os objectos roxos (c/ a propriedade `constructor`) são os protótipos, e os laranja são os construtores.



35. Vamos criar um `Ponto2DColorido` com as seguintes características:

1. É também um `Ponto2D`, logo vai herdar as propriedades de `Ponto2D`
2. Redefine o método `.toString()`, estendendo-o para indicar a cor
3. Tem duas novas propriedades:

   3.1 `cor`: que é um objecto literal com r, g e b

   3.2 `brilha`: método para modificar o brilho do ponto em percentagem do valor actual da cor

Introduza:

```
function Ponto2DColorido(x, y, cor) {
    Ponto2D.call(this, x, y);     // chama Ponto2D, sem operador new, e com
    this.cor = cor;               // this associado ao objecto criado com
}                                 // new Ponto2DColorido
```

*Como satisfazer o requisito 1? Noutras linguagens utilizamos **herança**. Existe herança neste modelo pseudoclássico? Sim: se substituirmos o protótipo original de um construtor por outro objecto (ver a seguir), então podemos herdar as propriedades desse objecto. Neste caso concreto, vamos **substituir o protótipo de** `Ponto2DColorido` (qualquer função "nasce" com um protótipo [quase] vazio) por um objecto criado a partir do protótipo de `Ponto2D`. Para isto funcionar, o construtor `Ponto2DColorido` deve inicializar a parte do objecto que está a ser construído que também é um `Ponto2D`, ou seja, tem que chamar o construtor `Ponto2D` mas para o objecto `Ponto2DColorido` que está a ser construído.*

36. E agora acrescente:

```
Ponto2DColorido.prototype =
Object.create(Ponto2D.prototype);
Ponto2DColorido.prototype.constructor = Ponto2DColorido;
```

São estas duas instruções que implementam ***herança prototipal*** *(ver caixa do lado). Também era possível substituir a chamada a* `Object.create` *por*

```
Ponto2DColorido.prototype = new Ponto2D();
```

Só que isto faria com que o construtor `Ponto2` *fosse executado e as propriedades* `x` *e* `y` *passariam para o protótipo.*

`Object.create` permite criar um objecto e especificar qual o protótipo desse mesmo objecto. Queremos um novo protótipo para `Ponto2DColorido`, cujo protótipo é o protótipo de `Ponto2D`. Temos, depois, de actualizar a propriedade `constructor` (está a apontar para `Ponto2D` quando deveria apontar para `Ponto2DColorido`)

37. Vamos redefinir o método herdado `Ponto2D.toString`:

```
Ponto2DColorido.prototype.toString = function() {
    // erro: inst. seguinte entra em "loop" recursivo
    // const txt = this.toString();
    const txt = Ponto2D.prototype.toString.call(this);
    const cor = this.cor;
    return txt + ` (r: ${cor.r} g: ${cor.g} b: ${cor.b})`;
}
```

*Note-se que queremos aproveitar o método existente na "**superclasse**" `Ponto2D`. Para invocarmos o "**supermétodo**" não podemos fazer `this.toString()` porque isso faria com que o método `Ponto2DColorido.prototype.toString` fosse recursivamente chamado. Para invocarmos o "supermétodo", e assim reutilizarmos o seu código, temos que chamar aceder ao método herdado através de `Ponto2D`.*

38. `Ponto2DColorido` vai estender `Ponto2D` com um novo método, o método `brilha`:

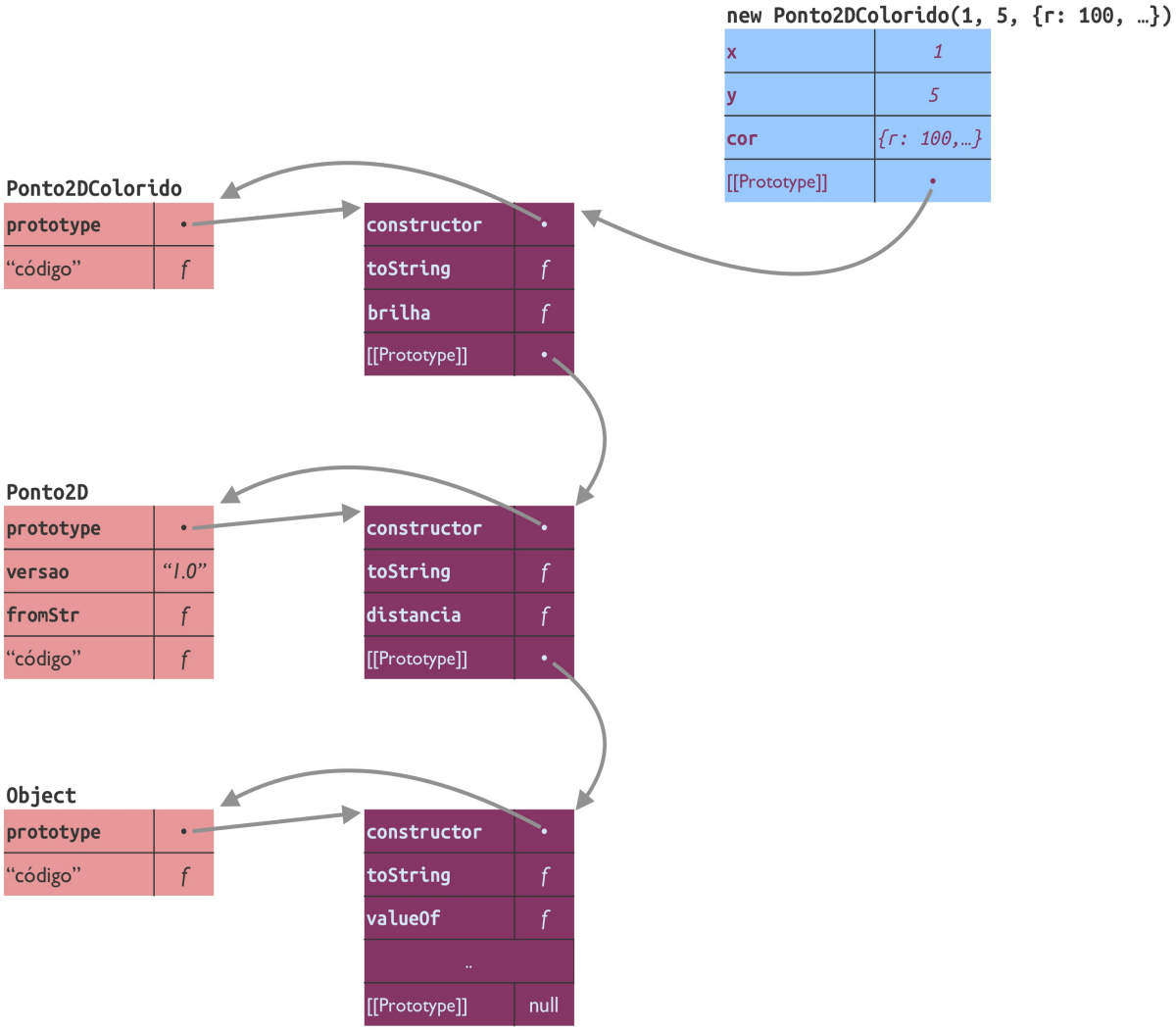
```
Ponto2DColorido.prototype.brilha = function(perc) {
    const [min, max, round] = [Math.min, Math.max, Math.round];
    const cor = this.cor;
    const factor = (1 + perc / 100);
    cor.r = max(0, min(255, round(cor.r * factor)));
    cor.g = max(0, min(255, round(cor.g * factor)));
    cor.b = max(0, min(255, round(cor.b * factor)));
    return cor;
}
```

39. E agora testamos tudo com:

```
const ponto3 = new Ponto2DColorido(40, -35, {r: 11, g: 201, b: 156});
const ponto4 = new Ponto2DColorido(12, 19, {r: 110, g: 101, b: 39});
console.log(ponto3);
console.log(ponto4);
console.log("ponto1 instanceof Ponto2D? ", ponto1 instanceof Ponto2D);
console.log("ponto3 instanceof Ponto2D? ", ponto3 instanceof Ponto2D);
console.log("ponto1 instanceof Ponto2DColorido? ", ponto1 instanceof Ponto2DColorido);
```

```
console.log("ponto3 instanceof Ponto2DColorido? ", ponto3 instanceof Ponto2DColorido);
console.log(ponto3.toString());
console.log(ponto4.toString());
console.log(ponto3.distancia(ponto4));
```

40. A figura seguinte detalha graficamente então a ligação entre os objectos principais:



41. Um problema: continuamos a não poder capturar uma referência para um método. Introduza:

```
const distancia = ponto2.distancia;
console.log(distancia(ponto2));     // erro
const brilha = ponto3.brilha;
console.log(brilha());              // erro
```

42. Pontos devem ser imutáveis. Não devemos poder mudar as coordenadas `x` e `y`. Podemos utilizar `defineProperty` para obter esse efeito de duas maneiras. A primeira torna as propriedades `x` e `y` imutáveis em modo strict (e apenas neste modo):

```
function Ponto2D(x, y) {
    const propertyDefinitions =  {
        writable: false,
        configurable: true,
        enumerable: true
    };
    Object.defineProperty(this, 'x', {value: x, ...propertyDefinitions});
    Object.defineProperty(this, 'y', {value: y, ...propertyDefinitions});
}

const ponto7 = new Ponto2D(5, -4);
console.log(ponto7.x, ponto7.y);
(function() {
    'use strict';
    ponto7.x = 40;
})();
```

De notar que, com o ES6, podemos definir getters e setters em objectos literais. Aqui vai um exemplo

```
let aluno = {
    nome: 'alberto antunes',
    get apelido() {
        const pos = this.nome.lastIndexOf(' ') + 1;
        return this.nome.slice(pos);
    },
    get dataNascimento() {
        return new Date('1997-10-10');
    },
    set dataNascimento(val) {
        throw new Error("As pessoas não mudam de data "
                        + "de nascimento!");
    }
};
```

43. Uma outra solução passa por definir um *setter* para lançar um erro quando a propriedade é chamada. Isto obriga-nos a definir um *getter* para aceder aos valores de `x` e `y`.

```
function Ponto2D(x, y) {
    const propertyDefinitions =  {
        set(val) {
            throw new Error('Imutable property!')
        },
        configurable: true,
        enumerable: true,
    };
    Object.defineProperty(this, 'x', {get()  {return x;},  ...propertyDefinitions});
    Object.defineProperty(this, 'y', {get() {return y;}, ...propertyDefinitions});
}
```

Consultar:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set

PARTE V – MODELO CLÁSSICO: POO COM CLASSES ES6

44. Com o ES6, veio a palavra-reservada ***class*** e todo um suporte sintático para trabalhar com classes "por cima" deste modelo prototipal. Vamos redefinir a hierarquia anterior de `Ponto`s com este novo mecanismo:

*A solução anterior para POO não é propriamente agradável de escrever. Ainda que forneça **encapsulamento**, no sentido em que atributos e métodos ficam "contidos" num mesmo objecto, o código fica disperso pelo ficheiro, e não está contido dentro de uma unidade lógica (de um espaço de nomes) (...)*

```
class Ponto2D {
    visivel = true;
    static versao = "2.0";
    static _pontoStrRegex =
       /^<[\-+]?\d+(\.\d+)?,\s*[\-+]?\d+(\.\d+)?>$/;

    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    static fromString(pontoStr) {
        if (!Ponto2D._pontoStrRegex.test(pontoStr)) {
            throw new Error(`Invalid string value for point ${pontoStr}`);
        }
        let [xStr, yStr] = pontoStr.split(',');
        xStr = xStr.trim().slice(1);
        yStr = yStr.trim().slice(0, -1);
        return new Ponto2D(parseFloat(xStr), parseFloat(yStr));
    }
    toString() {
        return `<${this.x}, ${this.y}>`;
    }
    distancia(p2) {
        const pow = Math.pow;
        const xDistSq = pow(this.x - p2.x, 2);
        const yDistSq = pow(this.y - p2.y, 2);
        return Math.sqrt(xDistSq + yDistSq);
    }
}
```

*(...) Com o ES6 veio a palavra-reservada **class** e com ela um suporte sintático para trabalhar com classes neste modelo prototipal. Trata-se meramente de "açúcar sintático", já que, na verdade, JavaScript continua a ser uma linguagem sem classes. (...)*

(...) Por um lado, se pretendemos programar neste modelo pseudoclássico, o "açúcar sintático" (*) é bem vindo já que torna o código mais claro. Por outro lado, ao recorrer a uma sintaxe inspirada na linguagem Java, pode dar a ideia errada de que JavaScript tem classes, quando não tem.

*(*): Algumas fontes (eg, https://javascript.info/class#not-just-a-syntactic-sugar) não consideram que classes são apenas "açúcar sintático". Não partilhamos dessa opinião porque a) classes ES6 não alteram nem acrescentam nada de fundamental ao mecanismo de programação com protótipos - os protótipos continuam lá, de tal maneira que é possível estender com **extends** uma "classe" definida ao jeito do modelo anterior - e, b) o que acrescentam, pode ser replicado facilmente com mecanismos já existentes na linguagem. Dito isto, não temos nada contra "açúcar sintático", nem contra mecanismos que facilitem a escrita do código, como é o caso das classes ES6, que simplificam e facilitam o desenvolvimento de código com protótipos em JavaScript.*

45. E agora teste com:

```
const ponto1 = new Ponto2D(10, -20);
const ponto2 = new Ponto2D(-1, 40);

// Cada objecto tem o seu x, y mas todos partilham toString
console.log(ponto1.toString === ponto2.toString)
console.log(ponto1, ponto2);
console.log(ponto1.toString(), ponto2.toString());
console.log(ponto1.constructor.name);
console.log(Object.getPrototypeOf(ponto1) === Ponto2D.prototype);
```

46. E agora temos a classe `Ponto2Colorido`:

*À semelhança de Java, utilizamos **`extends`** para indicar que uma classe herda de outra. E, tal como em Java, uma classe apenas pode herdar de uma outra classe. Ou seja, classes ES6 suportam apenas **herança simples** (por oposição a **herança múltipla**).*

```
class Ponto2DColorido extends Ponto2D {
    constructor(x, y, cor) {
        super(x, y);
        this.cor = cor;
    }
    toString() {
        const txt = super.toString();
        const cor = this.cor;
        return txt + ` (r: ${cor.r} g: ${cor.g} b: ${cor.b})`;
    }
    brilha(perc) {
        const [min, max, round, cor] = [Math.min, Math.max, Math.round, this.cor];
        const factor = (1 + perc / 100);
        cor.r = max(0, min(255, round(cor.r * factor)));
        cor.g = max(0, min(255, round(cor.g * factor)));
        cor.b = max(0, min(255, round(cor.b * factor)));
        return cor;
    }
}
```

47. Seguida do código que testa esta hierarquia de classes:

```
const ponto3 = new Ponto2DColorido(40, -35, {r: 11, g: 201, b: 156});
const ponto4 = new Ponto2DColorido(12, 19, {r: 110, g: 101, b: 39});
console.log(ponto3);
console.log(ponto4);
console.log("ponto1 instanceof Ponto2D? ", ponto1 instanceof Ponto2D);
console.log("ponto3 instanceof Ponto2D? ", ponto3 instanceof Ponto2D);
console.log("ponto1 instanceof Ponto2DColorido? ", ponto1 instanceof Ponto2DColorido);
console.log("ponto3 instanceof Ponto2DColorido? ", ponto3 instanceof Ponto2DColorido);
console.log(ponto3.toString());
console.log(ponto4.toString());
console.log(ponto3.distancia(ponto4));
console.log(Object.getPrototypeOf(ponto3) === Ponto2DColorido.prototype);
console.log(Object.getPrototypeOf(Object.getPrototypeOf(ponto3)) === Ponto2D.prototype);
```

48. Também aqui, uma referência para um método não traz consigo o objecto a partir do qual a

referência foi retirada.

```
const toStr = ponto3.toString;
console.log(toStr(s));          // erro
```

49. Este exemplo não exercita todas as funcionalidades, nem todo os mecanismos, que acompanham a definição de classes ES6. Sugere-se a leitura atenta das referências indicadas na última parte do laboratório.

50. Tarefa: torne os `Ponto2D` (e os `Ponto2DColorido`) imutáveis em termos das propriedades `x` e `y`.

PARTE VI – MODELO SEM CLASSES E FUNCIONAL: POO COM "FÁBRICAS DE OBJECTOS"

51. Antes de avançar, comece por rever muito bem o conceito de *closure* e IIFE. Já está?

As soluções anteriores sofrem de alguns problemas:

1. São complicadas, sendo necessário escrever muito código de infraestrutura (com protótipos), ou com muita sintaxe (class).
2. Não oferecem privacidade. À data de julho de 2020, existem propostas para adicionar propriedades privadas/protegidas a classes ES6.
3. Estimulam a utilização do modelo clássico para POO, modelo que tem sido alvo de muitas críticas em tempos recentes, especialmente o mecanismo de herança. Muita vezes, este modelo leva à criação hieraquias de objectos muito rígidas, difíceis de compreender, alterar e manter.
*4. Dependem da utilização de **this**, e a utilização desta propriedade é confusa.*
*5. Dependem da utilização de **new**, e este operador (não só aqui, mas em todas linguagens onde existe) levanta alguns problemas em termos de manutenção do código, nomeadamente, no facto de não podermos tratar um construtor como sendo apenas mais uma função. Além disso, no modelo pseudoclássico, se nos esquecermos de invocar o construtor com **new**, o JavaScript não alerta para o erro.*
6. Estimulam a utilização de objectos com muito estado (ie, variáveis) partilhado entre os diversos métodos. Corremos o risco de enfrentar os mesmos problemas que encontramos quando programamos com muitas variáveis globais.

Vamos ver uma solução mais simples, que representa uma outra abordagem à POO, baseada em closures e objectos literais, que consegue obter a maioria dos benefícios da POO.

Trata-se, na verdade, de um padrão de software muito utilizado em JavaScript, ainda que possam existir outros muito parecidos. De facto, existem várias "receitas" para chegar a uma solução similar a esta. Aqui está (*):

```
function constructor(spec) {
    let {member} = spec;
    let {other} = otherConstructor(spec);
                  // inheritance call
    return {
      method1: function(...) {
          // tem acesso a member, other, spec
      },
      // ...
      methodN: function(...) {
          // tem acesso a member, other, spec
      }
    };
}
```

spec representa um objecto de especificação. Aconselha-se a utilização de um destes objectos para transportar os argumentos de construtores, dado que, estes tendem a ter muitos parâmetros. As variáveis locais do construtor actuam como propriedades privadas do objecto devolvido, uma vez que são capturadas pelos métodos deste objecto, mas não são directamente acessíveis através dele. Se quisermos devolver um objecto imutável, ou seja, um objeto incorruptível e seguro, então podemos envolver o objecto devolvido em `Object.freeze`.

(*) - Baseado no modelo recomendado por Douglas Crockford

52. Implemente:

```
function ponto2D(x, y) {
    return {
        x,
        y,
        visivel: true,
        toString: function() {
            return `<${x}, ${y}>`;
        },
        distancia: function(p2) {
            const pow = Math.pow;
            const xDistSq = pow(x - p2.x, 2);
            const yDistSq = pow(y - p2.y, 2);
            return Math.sqrt(xDistSq + yDistSq);
        }
    };
}

ponto2D.versao = "3.0";

ponto2D.fromString = (function() {
    const pontoStrRegex = /^<[\-+]?\d+(\.\d+)?,\s*[\-+]?\d+(\.\d+)?>$/;
    return function(pontoStr) {
        if (!pontoStrRegex.test(pontoStr)) {
            throw new Error(`Invalid string value for point ${pontoStr}`);
        }
        let [xStr, yStr] = pontoStr.split(',');
        xStr = xStr.trim().slice(1);
        yStr = yStr.trim().slice(0, -1);
        return ponto2D(parseFloat(xStr), parseFloat(yStr));
    }
}());
```

*A função `ponto2D` é uma função que designamos de **função fábrica** (factory function). Como `ponto2D` está na raiz da nossa hierarquia de objectos, não necessita de chamar outros construtores.*

*De notar que aqui, e em `ponto2DColorido` (ver à frente), não utilizamos **`this`**. Também não precisamos de **`new`**. Trabalhamos com objectos literais e com variáveis locais às funções. Os métodos `toString` e `distancia` são **closures** que capturam as variáveis locais `x` e `y`.*

NOTA: Para manter a "compatibilidade" com os exemplos anteriores, aqui não utilizamos um objecto de especificação nos nossos construtores.

53. E agora teste com:

```
const ponto1 = ponto2D(10, -20);
const ponto2 = ponto2D(-1, 40);
console.log(ponto1.toString === ponto2.toString)
console.log(ponto1, ponto2);
console.log(ponto1.toString(), ponto2.toString());
```

*Em JavaScript moderno podemos definir getters, que são métodos acedidos como se fossem propriedades de dados. Ao contrário de Java, estes getters podem ser acrescentados aos nossos objectos após o código ter sido desenvolvido e estar em utilização, isto sem, colocar em causa a **abstração de dados** que pretendemos para os nossos objectos. (...)*

(...) Por exemplo, vamos supor que era mais eficiente guardar as coordenadas num array de números reais (o ES6 trouxe arrays tipificados a JavaScript). Podemos facilmente mudar a definição de `ponto2D`, sem que o resto do código tenha que ser alterado:

```
function ponto2D(x, y) {
  let coords = Float64Array.of(x, y);
  return {
      get x() {
          return coords[0];
      },
      get y() {
          return coords[1];
      },
      toString: function() {
          return `<${coords[0]}, ${coords[1]}>`;
      },
      distancia: function(p2) {
         const pow = Math.pow;
         const xDistSq = pow(coords[0] - p2.x, 2);
         const yDistSq = pow(coords[1] - p2.y, 2);
         return Math.sqrt(xDistSq + yDistSq);
      }
  };
}
```

De notar que, agora, as propriedades x e y são imutáveis uma vez que não definimos setters.

Uma variação do padrão apresentado anteriormente (e que levou à definição de `ponto2D`) reaproveita o objecto devolvido pelo "super-construtor" (caso se utilize um):

```
function constructor(spec) {
    let {member} = spec;
    let base = otherConstructor(spec);
    base.method1 = function(...) {
        ...
    };
    // ...
    base.method1 = function(...) {
        ...
    };
    return base;
}
```

Vamos utilizar este padrão para implementar `ponto2DColorido`. O construtor `ponto2D` vai actuar como `otherConstructor`.
Se pretendermos devolver um objecto imutável, podemos recorrer a `Object.freeze`, fazendo, por exemplo:

```
return Object.freeze(base);
```

54. E agora `ponto2DColorido`, que vai "herdar" de `ponto2D`:

```
function ponto2DColorido(x, y, cor) {
    let base = ponto2D(x, y);
    let baseToString = base.toString;

    base.toString = function() {
        const txt = baseToString();
        return txt + ` (r: ${cor.r} g: ${cor.g} b: ${cor.b})`;
    };

    base.brilha = function() {
        const [min, max, round] = [Math.min, Math.max, Math.round];
        const factor = (1 + perc / 100);
        cor.r = max(0, min(255, round(cor.r * factor)));
        cor.g = max(0, min(255, round(cor.g * factor)));
        cor.b = max(0, min(255, round(cor.b * factor)));
        return cor;
    };
    return base;
}
```

55. Introduza, agora:

```
const ponto3 = ponto2DColorido(40, -35, {r: 11, g: 201, b: 156});
const ponto4 = ponto2DColorido(12, 19, {r: 110, g: 101, b: 39});
console.log(ponto3.toString());
console.log(ponto4.toString());
console.log(ponto3.distancia(ponto4));
```

56. Como não usamos ***this***, podemos passar uma referência para um método sem termos que nos preocupar com a associação de ***this***:

```
let distancia = ponto3.distancia;
console.log(distancia(ponto2));
```

VANTAGENS *deste modelo:*

1. Closure garante ***privacidade****.*

2 . ***Mais simples*** *porque não exige sintaxe acrescida nem manipulação de manual de protótipos.*

3. ***Não usa new nem this****, o que em JavaScript é um bónus: não há o risco de acidentalmente alterar o objecto global, não é preciso fazer "rebinding" de* ***this****, etc.*

4 . ***Não utiliza herança*** *nem leva a hierarquias de objectos muito profundas e rígidas, à semelhança do que acontece em Java, C++ e C#. Nestas linguagens, e em JavaScript, quando utilizamos os outros modelos, em particular o modelo clássico, utiliza-se herança para partilha de código (ie, de comportamento). Ora, a herança de classes, apesar de ser um mecanismo interessante para reutilização de código, tende a aumentar a dependência entre classes. A um ponto que, a dada altura, torna-se muito difícil alterar uma classe de base sem quebrar a compatibilidade com as classes derivadas.*

5. Em qualquer instante podemos ***alterar a composição*** *dos objectos sem que isso implique alterações no código cliente, tal como quando se utiliza* ***new*** *(quando fazemos* ***new*** `Xpto(...)` *os objectos serão sempre do tipo* `Xpto`*, mesmo que, em dada ocasião, precisemos de devolver um objecto do tipo* `Ypto`*).*

6 . ***Mais rápido*** *localizar um método numa "hierarquia" de objectos. Os métodos estão logo "ali", no objecto, e não é necessário percorrer uma cadeia de protótipos para lá chegar.*

DESVANTAGENS *deste modelo:*

1. Necessita de ***mais memória*** *por cada instância porque cada* `ponto2D` *tem o seu* `.toString`*, o seu* `.distancia`*, o seu* `.brilha`*, etc. Apesar do código de cada um desses métodos ser partilhado, é necessário um objecto função por método e por objecto* `ponto2D`*. Ao passo que, com protótipos/classes, todos os* `Ponto2D` *partilham os mesmos objectos função. No entanto, a memória extra consumida por cada objecto* `ponto2D`*, comparativamente com o que sucede com essas soluções, é reduzida, e isto só tem impacto quando lidamos com muitos milhões de objectos. Neste caso, se calhar é melhor utilizar outra linguagem de programação.*

2. ***Não tipificado****:* ***instanceof*** *não funciona. Porém, não é difícil acrescentar suporte adhoc para tipos, além de que* ***instanceof*** *e* ***typeof*** *respondem à pergunta "quem és tu?" quando, se calhar, apenas precisamos de perguntar "o que é que tu consegues fazer?".*

3. Não é permitido enriquecer um protótipo com métodos e, com isso, "injectar" nova funcionalidade em todos os objectos que herdam desse protótipo. Ou seja, ***não permite herança retroactiva****. É, no entanto, possível alterar ou estender dinamicamente os super-construtores através de um decorador (*)*

(*) Um ***decorador*** *é uma função que recebe uma função e devolve outra, que chama a recebida, mas que pode acrescentar funcionalidade a essa função recebida.*

57. Se pretendermos uma identificação de tipos, compatível com tipos primitivos e objectos *built-in*,

podemos utilizar `Object.prototype.toString` e a propriedade `[Symbol.toStringTag]`. Primeiro, acrescente esta propriedade a `ponto2D` e a `ponto2DColorido`:

```
function ponto2D(x, y) {
    return {
        // ...
        [Symbol.toStringTag]: 'ponto2D'
    };
}
function ponto2DColorido(x, y, cor) {
    // ...
    base[Symbol.toStringTag] = 'ponto2DColorido';
    return base;
}
```

O método `Object.prototype.toString` identifica os objectos exibindo uma string do tipo `'[object T]'` em que `T` é o valor da propriedade `[Symbol.toStringTag]` do objecto, caso esta propriedade exista. Caso não exista, `Object.prototype.toString` exibe `'Object'` no lugar de `T`. Atenção que `Object.prototype.toString` é um método de instância e, logo, deve ser chamado como `obj.toString()`. Mas, como os nossos objectos redefinem `toString`, não podemos chamar este método desta forma.

Queremos chamar a versão de base do método, aquela foi definida no protótipo de `Object`. Temos (pelo menos) duas hipóteses, a longa e a curta:

1. `Object.prototype.toString(ponto1)`
2. `{}.toString.call(ponto1)`

NOTA: *Poderá ter que envolver `{}` num par de parênteses. Consultar:*
https://javascript.info/instanceof
https://javascript.info/symbol

58. E agora podemos verificar o tipo de dados dos objectos com:

```
console.log({}.toString.call(ponto1));
console.log({}.toString.call(ponto4));
```

59. Tarefa: torne os `ponto2D` (e os `ponto2DColorido`) imutáveis em termos das propriedades `x` e `y`.

PARTE VII – ACOMPANHAMENTO

60. Como complemente aos tópicos abordados neste laboratório, estude a seguinte documentação de apoio, em particular as duas primeiras referências.

[1]: Ilya Cantor, *"Objects: The Basics/Classes/Prototypes, inheritance - The Modern JavaScript Tutorial"*: https:javascript.info/object-basics https:javascript.info/classes https:javascript.info/prototypes

[2]: *"Object.create Examples", MDN Web Docs*:
https:developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#Examples

[3]: Marijn Haverbeke, *"Eloquent JavaScript, 3rd Ed. - Chapter 06: The Secret Life of Objects"*, 2018, No Starch Press, https:eloquentjavascript.net/06_object.html

[4]: Charles Scalfani, *"Goodbye, Object Oriented Programming"*, Jul. 2016, Medium:
https:medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53

[5]: Dan Abramov, *"How to Use Classes and Sleep at Night"* , Out. 2015, Medium:
https://medium.com/@dan_abramov/how-to-use-classes-and-sleep-at-night-9af8de78ccb4

61. Estude também os exemplos que acompanham este laboratório.

EXERCÍCIOS DE REVISÃO

1. No contexto de JavaScript, quais são os mecanismos principais de JavaScript utilizados para representar uma "classe" e respectivas definições de atributos e métodos?

Modelo Pseudoclássico	:	*Funções (construtores), propriedade* ***this****, operador* ***new****, protótipos*
Modelo Clássico	:	
Modelo Sem Classes e Funcional	:	

2. Uma classe ES6 é também um objecto? Se sim, que tipo de objecto? (outra forma de fazer esta última pergunta é: Qual é a classe de uma classe?)

3. Analise os seguintes blocos de código e execute as tarefas pedidas.

3.1 Leia o bloco de código seguinte e desenhe um diagrama a ilustrar a cadeia de protótipos, com todos os objectos envolvidos (incluindo `Object`), atributos e métodos relevantes:

```
function C(a) {
    this.a = a;
}
```

```
C.prototype.met1 = function met1(x) {
    return this.a + x;
};
```

3.2 Faça o mesmo para o bloco de código seguinte (inclua `C` e `Object` no diagrama):

```
function D(a, b) {
    C.call(this, a);
    this.b = b;
}

D.prototype = Object.create(C.prototype);
D.prototype.constructor = D;
```

```
C.prototype.met1 = function met1(x) {
    return this.a + x;
};
```

3.3 Atendendo a que

```
const obj1 = new C(10);
const obj2 = new D(10, 20);
```

o que é exibido pelas seguintes instruções:

```
console.log(obj1.constructor.name);
console.log(obj1 instanceof C);
console.log(obj1 instanceof D);
```

```
console.log(obj2.constructor.name);
console.log(obj2 instanceof C);
console.log(obj2 instanceof D);
```

```
console.log(Object.getPrototypeOf(Object.getPrototypeOf(obj2)).constructor.name);
console.log(Object.getPrototypeOf(Object.getPrototypeOf(obj1)).constructor.name);
```

```
console.log(obj1.met1(10));
console.log(obj1.met2(10));
console.log(obj2.met2(10));
```

```
console.log(obj2.toString());
console.log({}.toString(obj2));
console.log(JSON.stringify(obj2));
```

3.4 Refaça o exemplo completo utilizando classes ES6 (modelo clássico).

3.5 Refaça o exemplo completo utilizando funções fábrica (modelo sem classes).

4. Considere o seguinte bloco de código:

```
class Pessoa {
    constructor(nome) {
        this.nome = nome
    }
    apresenteSe() {
        return "Eu sou o/a "
               + this.obtemTitulo()
               + " " + this.nome + ".";
    }
    obtemTitulo() {
        return "";
    }
}

class PessoaFormal extends Pessoa {
    constructor(nome, titulo) {
        this.titulo = titulo;
    }
    apresenteSe() {
        return super.apresenteSe() + ". Ao seu dispor.";
    }

}
```

```
function teste() {
    let p = new Pessoa("Alberto");
    console.log(p.apresenteSe());

    p = new PessoaFormal("Armando",
                         "Doutor");
    console.log(p.apresenteSe());
}
```

Dentro do estilo de organização do código em cima exibido, introduza as alterações mínimas necessárias de modo a função `teste` exiba na consola:

```
Eu sou o/a  Alberto.
Eu sou o/a Doutor Armando.. Ao seu dispor.
```

5. Consulte o ficheiro `cliente.js` e resolva ou responda às seguintes questões:

5.1 Indique todos os construtores.

5.2 Quais as propriedades de instância de `ClienteEspecial`?

5.3 Neste contexto, qual a (aparente) vantagem do parâmetro `...args`?

5.4 Em `ClienteEspecial`, porque é que invocação a `Cliente` não utiliza **`new`**?

5.5 Quais os métodos de classe (ie, estáticos)?

5.6 Quais as duas instruções que implementam herança prototipal?

5.7 Indique um exemplo de polimorfismo.

5.8 Indique um exemplo de redefinição.

5.9 Termine a definição de `.toString` para os clientes especiais.

5.10 Qual o resultado de `cli4 instanceof Cliente` e como é que JavaScript obtém essa resposta?

5.11 Reimplemente este exemplo com classes ES6.

5.12 Reimplemente este exemplo sem classes e com fábricas de objectos.

6. Analise a o seguinte código:

```
class Colaborador {
    constructor(salBase) {
        this.salBase = salBase;
    }
    vencimento() {
        return this.salBase + this.obtemBonus();
    }
    obtemBonus() {
        return 50;
    }
}

class ColaboradorSenior extends Colaborador {
    constructor(salBase) {
         super(salBase);
    }
    obtemBonus() {
        return 200;
    }
}
```

```
function teste1() {
    let c = new Colaborador(1000);
    console.log(c.vencimento());

    c = new ColaboradorSenior(1000);
    console.log(c.vencimento());
}
```

6.1 O que é exibido por `teste1`?

6.2 É suposto o `ColaboradorSenior` receber o bónus dado aos `Colaborador`es (ou seja, ele deveria ter um bónus de 250). Neste sentido, quais as alterações (mínimas) a introduzir no programa?

6.3 O código apresentado tem um método a mais. Qual e porquê?

EXERCÍCIOS DE PROGRAMAÇÃO

7. Uma dada empresa de telecomunicações utiliza uma aplicação em JavaScript para gestão de clientes. Assim um `Cliente` possui primeiro nome, apelidos intermédios e apelido. Possui também idade, morada, código postal, NIF (número de identificação fiscal), e código de cliente. O nome completo deverá ser determinado a partir dos nomes todos. Deverá desenvolver uma operação para apresentar todos os dados de um cliente. Utilize

8. A mesma empresa guarda informação sobre as contas do cliente. Assim, cada conta possui código de conta (código alfanumérico único), morada, código postal, data de criação e uma referência para o cliente a quem a conta pertence. O valor da última factura também é guardado em cada conta. A esse valor acresce IVA que deverá ser calculado (utilize 23%).

9. Analise o exemplo fornecido sobre produtos, livros, jogos e encomendas. Implemente este exemplo com fábricas de objectos, e mecanismos associados, tal como vimos a respeito do modelo sem classes.

10. Uma aplicação de gestão necessita de lidar com informação sobre os seus colaboradores. Utilizando uma solução puramente funcional, baseada em fábricas de objectos, defina a classe `Colaborador` que deve possuir as seguintes propriedades:

. `primNome`: primeiro nome

. `apelido`

. `salarioAnual`: vencimento anual bruto (assuma que desconta 15% de IRS e 11% de TSU)

. `numMesesVenc`: quantos meses de vencimento (apenas dois valores são aceites: 12 ou 14)

. `nomeCompleto`: método para devolver o nome completo do colaborador

. `salarioAnualLiq`: método para devolver salário anual líquido

. `salarioMensalLiq`: método para devolver o salário líquido mensal

Utilizando objectos de especificação, desenvolva um construtor com todos os parâmetros

obrigatórios para inicializar os campos de cada objecto. Elabore código para testar esta classe, criando alguns objectos do tipo `Colaborador`.

11. Vamos supor que pretendemos implementar um sistema de informação para um banco. Em particular, pretendemos apenas representar as estruturas para lidar com contas bancárias: Eis os requisitos:

- Existem quatro tipos de contas: `Ordem`, `Ordenado`, `Prazo` e `PoupancaHabitacao`
- Para já, apenas estamos interessados em três operações bancárias básicas: obter saldo, levantar e depositar
- Todas as contas são associaciadas a um cliente através de um número de cliente. Têm, também, um número de conta (que é o *id* de uma conta), uma data de abertura, um saldo e uma duração (decorrida). Opcionalmente, podem receber um número de conta exterior que, no caso de não ser utilizado, é gerado por um mecanismo de auto-numeração.
- A conta a prazo possui taxa de juro, duração mínima e saldo mínimo. Se a duração mínima ainda não tiver sido atingida, o saldo não deverá contemplar a aplicação de juros.
- A conta poupança-habitação é um tipo de conta a prazo, variando apenas nos valores da duração e saldo mínimos.
- Para evitar complexidades adicionais 1) os juros são simples e não-compostos; 2) a contabilização dos juros é feita sempre do início, independentemente do número de depósitos feitos (isto é incorrecto para contas com juros, mas obrigaria a guardar uma lista de depósitos para calcular os valores correctos dos juros ou a uma actualização do juro total desde o início).
- A conta ordenado é um tipo de conta à ordem onde é possível ter um saldo negativo desde que não seja superior ao valor negativo do ordenado (ou seja, `saldo >= -ordenado`).
- Cada conta possui um estado com quatro valores: aberta, encerrada, bloqueada e inactiva.
- O método `toString` deverá devolver uma string com os valores de todos os atributos em formato CSV; o método `mostra` deverá devolver uma string com uma representação visual dos objectos.

Represente este domínio utilizando classes ES6 primeiro e depois com fábricas de objectos. Utilize objectos de especificação na lista de parâmetros dos construtores.

12. Pretende implementar um catálogo de livros. Para tal, estude o código fornecido à parte e desenvolva uma classe (eg, `Catalogo` ou `ListaLivros`) especializada em gerir uma lista de livros. Para armazenar os livros em memória utilize uma das colecções do JavaScript (eg, um `Map` ou um `Array`). Desenvolva os seguinte métodos:

. adicionar um livro não repetido

. pesquisas por ID, ISBN, título e género

. remover um livro do catálogo