www.devmedia.com.br
[versão para impressão]
Link original: http://www.devmedia.com.br/articles/viewcomp.asp?comp=34215

Boas Práticas de programação em Java Script

Esse artigo traz uma série de boas práticas sobre o desenvolvimento de código usando JavaScript, bem como erros e confusões comuns que geralmente são cometidos em relação à linguagem.

Fique por dentro

Este artigo é útil por introduzir uma série de conceitos e boas práticas associadas ao desenvolvimento de código JavaScript.

Quando se inicia em uma nova linguagem de programação é essencial conhecer a fundo os detalhes da mesma, bem como as boas práticas que a comunidade construiu com base em experiências e no velho modelo tentativa-erro. Aqui, trataremos de expor alguns dos principais problemas associados ao mau uso do JavaScript, os quais o leitor poderá usar para embasar seus projetos futuros ou criar seus próprios frameworks front-end.

A linguagem de programação JavaScript foi concebida como uma linguagem de scripts que proporcionava não somente uma fácil integração entre o browser, o seu documento DOM e a HTML, como também formas de manter o código do cliente organizado, totalmente orientado a objetos (e nisto incluímos vários *design patterns*, boas práticas de codificação, APIs, etc.) e, sobretudo, reutilizável.

E isto é o que todos os desenvolvedores mais buscam nos últimos tempos: módulos reusáveis, baseados em componentes e que forneçam uma interface em comum para ser acessada por diferentes clientes e alimentada por diferentes servidores.

Não é raro encontrar pelos fóruns da web ou blogs questões sobre o melhor uso do JavaScript em relação às suas estruturas. A maioria dos desenvolvedores sequer tem conhecimentos aprofundados na mesma ou ao menos entende bem o que cada implementação básica que faz, como a de uma *function*, por exemplo.

Não obstante a isso, neste artigo trataremos de introduzir uma série de boas práticas que podem ser usadas para te ajudar a desenvolver código JavaScript mais performático, produtivo e limpo.

Erros comuns

Para entender melhor como tais práticas podem ser otimizadas, vamos explorar alguns dos principais tipos de erros mais comuns ao se desenvolver código JavaScript.

Essa lista é importante pois abraça conceitos tidos como ultrapassados na linguagem e que devem ser entendidos para evitar dores de cabeça futuras.

Não usar "var" para declarar suas variáveis

Quando não as declaramos com essa palavra reservada o código provavelmente irá funcionar, mas fará com que as mesmas sejam criadas em âmbito global, acumulando espaço em memória desnecessário e, possivelmente, culminando em erros nas camadas mais inferiores do código. Um dos problemas mais comuns associados a esse tipo de estratégia é chamado de "vazamento de variáveis globais". Por exemplo, considere o código a seguir:

```
function teste() { abc = 'abc'; }
teste();
console.log('Ainda consigo ver o valor de abc: ' + abc);
```

Como a variável foi declarada dentro do escopo de um bloco (uma função), pelos termos comuns das linguagens de programação a mesma deveria ser considerada local. Entretanto, a ausência da palavra reservada *var* torna a mesma global, mesmo que ela esteja dentro de um bloco qualquer. Neste caso, a solução consiste em fazer o devido uso da mesma possibilitando que sua visualização esteja restrita ao bloco em questão:

```
function teste() { var abc = 'abc'; }
teste();
console.log('Agora não consigo mais ver o valor de abc: ' + abc);
```

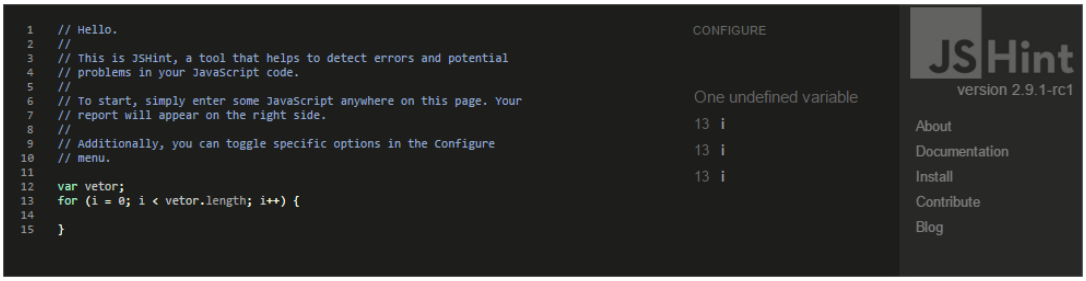
Outro exemplo clássico desse erro está na implementação da estrutura de repetição *for* do JavaScript:

```
for (i = 0; i < vetor.length; i++) { ... }
```

Neste exemplo, temos uma variável dentro de um bloco (o próprio *for*), porém como não usamos o *var* seu escopo será criado de forma global, logo se tentarmos acessar o valor final de *i* mesmo após finalizado o loop ainda conseguiremos fazê-lo. Uma simples declaração corrige todo o problema:

```
for (var i = 0; i < vetor.length; i++) { ... }
```

Para lidar melhor com este e os demais problemas que serão apresentados, o leitor pode fazer uso da ferramenta online **JSHint** (vide seção **Links**), que é uma ferramenta de análise semântica bem simples e intuitiva que te ajuda a identificar problemas mais óbvios nos seus códigos JavaScript. Veja na **Figura 1** as mensagens de validação do mesmo quanto ao problema do loop que apresentamos.



abrir imagem em nova janela

Figura 1. Exemplo de análise do loop com ferramenta JSHint.

Chamar funções de callback antes do previsto

Algumas APIs ou bibliotecas de browsers esperam receber funções como argumentos para seus principais métodos. Mas acontece que a maioria dos programadores acaba chamando a função antes que ela seja de fato enviada ao método como uma callback. Por exemplo, considere o código a seguir:

```
$(document).ready(inicializarComponentes());
```

O código mostra a chamada à função **ready()** do framework jQuery que é executada logo após a completa inicialização do documento DOM na página em questão. Esta função é largamente usada em substituição à função *onload()* do próprio documento HTML, em desuso pela comunidade. Veja que dentro dela chamamos uma função de nome *inicializarComponentes* que, por sua vez, será executada antes que a função ready() a reconheça.

O grande segredo nesse tipo de implementação está nos parênteses, pois são eles os responsáveis por disparar a execução de qualquer função no JavaScript. No mesmo exemplo, suponha que a implementação da função *inicializarComponentes* seja a demonstrada a seguir:

```
function inicializarComponentes() {
   console.log('Inicializando...');
}
```

Desta forma, sempre que desejarmos efetuar a execução do código interno a esta função basta chamar o seu nome acrescido dos parênteses, em qualquer lugar do seu código:

```
inicializarComponentes();
```

Já uma função de callback segue o conhecido princípio de *"Don't call me. I'll call you"* ("Não me chame. Eu o chamarei."). Em outras palavras, as funções ditas de callback não são executadas diretamente, logo não recebem parênteses nos locais onde são chamadas. O mesmo código de inicialização do jQuery deve ficar assim, portanto:

```
$(document).ready(inicializarComponentes);
```

A esse tipo de problema damos o nome de "Invocação Prematura". Ainda é possível também passar as funções de callback em formato de *string* em vez de *functions*. Vejamos um segundo exemplo:

```
setTimeout("inicializarComponentes()", 2000);
```

A função *setTimeout()* é responsável por executar uma determinada função (primeiro parâmetro) ao final de um certo limite de tempo estipulado (segundo parâmetro). Ao passar a função de callback entre as aspas duplas possibilitamos que a mesma fosse traduzida automaticamente pelo JavaScript para uma *function* e, sobretudo, não fosse executada imediatamente (como aconteceria caso as aspas não fossem usadas).

Todavia, esse tipo de implementação tem suas limitações: ela só funciona para funções de callback que são esperadas como parâmetros de funções. Se precisarmos configurar uma função associada a um atributo de um objeto, o mesmo procedimento não funcionará. Por exemplo, considere que a função *inicializarComponentes()* deva ser associada ao evento *onload* do objeto *window* do DOM:

```
window.onload = "inicializarComponentes()";
```

O referido código não funcionará porque atributos esperam sempre um valor (tipo primitivo, objeto ou função). Se passarmos uma string, a mesma será ignorada por não atender à regra. Para este mesmo caso, a solução seria remover os parênteses e deixar que a *onload* se encarregue de executar a callback na hora certa.

O escopo "this" pode não ser o que você espera

Geralmente, quando criamos uma função de callback dentro de outra função, o valor que está associado ao operador **this** (ou o contexto) muda. Esse tipo de implementação requer um conhecimento mais aprofundo no gerenciamento de contextos pelo JavaScript por parte do desenvolvedor, principalmente no que remete ao uso de cadeias de funções aninhadas.

De todo modo, a melhor estratégia é sempre salvar seus escopos *this* em variáveis locais dentro da função principal e fora da função de callback.

O grande segredo é saber quem o *this* está referenciando, observando bem quando uma *function() {}* está dentro de outra. Tomemos como exemplo as funções *ready()* e *setTimeout()* vistas há pouco que, quando unidas, podem ser representadas da seguinte maneira:

```
$(document).ready(function() {
   setTimeout(function() { this.getElementBy('teste') }, 2000);
});
```

Veja que no exemplo estamos tentando recuperar um elemento de id "teste" no documento HTML que selecionamos através do seletor jQuery *$(document)*.

Todavia, o acesso à variável *this* foi feito de dentro da função *setTimeout()*, logo é a esta que o *this* está referenciando e, portanto, não temos acesso ao *this* externo da função *ready()*.

Uma possível solução seria migrar o trecho de código para antes da função *setTimeout()*, mas e se quisermos acessar essa propriedade dentro desta função, isto é, após um determinado *timeout* finalizar? Para isso, basta salvar o valor da variável em uma outra variável local, tal como fizemos na **Listagem 1**.

Listagem 1. Exemplo com a variável this.

```
$(document).ready(function() {
     var teste = this;
     setTimeout(function() {
  teste.getElementBy('teste')
    }, 2000);
});
```

Existem algumas situações mais específicas que podem gerar confusão quando fizermos uso do *this*. Por exemplo, considere o código apresentado na **Listagem 2** onde temos a declaração de um objeto *pessoa* com somente um atributo: *nome*.

Em seguida, associamos uma função ao mesmo objeto chamada *imprimeNome()* que, quando executada, imprime uma mensagem de alerta com o nome do referido objeto. Veja que para este exemplo fazemos uso do operador *this* através de um processo que chamamos de *binding*, ou associação. Basicamente, o que o JavaScript faz é converter o *this* para *pessoa* no exemplo permitindo assim o devido acesso às suas propriedades.

Listagem 2. Exemplo de função simples com o this.

```
var pessoa = {nome: 'Fabricio'};

pessoa.imprimeNome = function() {
  alert('Olá, meu nome é ' + this.nome);
}

pessoa.imprimeNome(); // Abre uma janela de alerta com o texto "Olá, meu nome é Fabricio"
```

Contudo, em alguns casos essa implementação pode não trazer o resultado esperado. Se você chamar a mesma função por si só, por exemplo, seu contexto não será associado ao objeto *pessoa*, mas sim ao objeto global *window*:

```
var boasvindas = pessoa.imprimeNome;
boasvindas(); // Abre uma janela com o texto "Olá, meu nome é undefined"
```

O objeto *window*, por sua vez, não tem nenhum atributo "nome" em sua constituição o que fará com que a mensagem receba o *undefined* junto. Para solucionar esse tipo de problema podemos fazer uso das funções *call()* e *apply()* do JavaScript que permitem configurar qual contexto deve ser usado para cada chamada de função, seja ela de callback ou não. Dessa forma, podemos garantir que as chamadas ao *this* estarão associadas aos devidos objetos. Vejamos:

```
var pessoa = { nome: 'Fabricio' };
imprimeNome.call(pessoa); // Abre janela com o texto "Olá, meu nome é Fabricio"
imprimeNome.apply(pessoa); // Abre janela com o texto "Olá, meu nome é Fabricio"
```

Ambos os métodos estão disponíveis para todas as funções, recebendo o primeiro argumento com o contexto a ser usado. Para os casos em que a nossa função receber algum parâmetro, a chamada a estas funções deve passar o mesmo logo em sequência ao primeiro, como podemos ver na **Listagem 3**.

Nela a função *imprimeNome()* agora recebe uma parâmetro *qtde* referente à quantidade de vezes que a mensagem de boas-vindas deve ser impressa no *alert*. Para a função *apply()*, especificamente,

precisamos passar a lista de parâmetros do método em um vetor, na ordem em que forem definidos na assinatura original da função.

Listagem 3. Exemplo de uso do call/apply com função que recebe parâmetro.

```
var pessoa = {nome: 'Fabricio'};

pessoa.imprimeNome = function(qtde) {
   for (var i = 0; i < qtde; i++) {
      alert('Olá, meu nome é ' + this.nome);
   }
};

imprimeNome.call(pessoa, 2); // Exibe a mensagem duas vezes
imprimeNome.apply(pessoa, [4]) // Exibe a mensagem quatro vezes
```

Uso indevido de callbacks em loops

A criação de funções de callback como *event handlers* (ouvintes de eventos de click, mouse, etc.) dentro de um loop pode não funcionar apropriadamente dependendo da variável contadora do mesmo.

A solução para estes casos quase sempre está em criar uma segunda função e fazê-la funcionar como uma *closure*. Por exemplo, suponha que queremos registrar um evento de click para cada um dos botões que tivermos numa página (veja o código da **Listagem 4** para isso).

Listagem 4. Função que associa eventos de click para botões na página.

```
var botoes = document.getElementsByTagName('button')
for (var i = 0, len = botoes.length; i < len; i++) {
    botoes[i].addEventListener('click', function(e) {
        console.log('Você clicou no botão de nº ' + i);
    }, false);
}
```

O código tem como objetivo vasculhar todos os elementos <*button*> da página e adicionar uma função anônima a cada evento de click dos mesmos que, por sua vez, imprimirá uma mensagem no console do browser informando o id do botão clicado.

A lógica da implementação está correta, todavia seu resultado final não atenderá à mesma. Isso porque cada função de callback que criamos (uma para cada iteração no loop) referencia a mesma variável *i*.

Em outras palavras, não será criada uma nova variável "var i" para cada iteração do *for*, logo a mesma será usada para associar às mensagens do console. Dessa forma, o resultado será a impressão da mensagem "Você clicou no botão de nº 4" em todas as iterações do loop (para os cenários onde temos quatro botões na página de teste).

Uma possível solução para esse problema é envolver o nosso código interno do loop em uma *outer function*, ou função externa. Uma *outer function* é executada assim que for criada e, então, é descartada. Dessa forma, o JavaScript é forçado a criar um novo escopo de variável *i* e a inicializar com o valor atual do loop.

Vejamos como fazer isso na **Listagem 5**. Note que estamos passando a variável *i* agora como um

parâmetro para a *outer function* no final da mesma, além de mudando o nome do argumento interno para *j* evitando assim confusões de nomenclatura.

Listagem 5. Loop de eventos dentro de uma outer function.

```
var botoes = document.getElementsByTagName('button');
for (var i = 0, len = botoes.length; i < len; i++){
    !function outer(j){
        botoes[i].addEventListener('click', function inner(e){
            console.log('Você clicou no botão de nº ' + j);
        }, false);
    }(i);
}
```

Processamento de tarefas intensivas sem bloqueio do browser

Mais cedo ou mais tarde, casos em que precisamos processar uma grande quantidade de dados via JavaScript aparecerão e exigirão um esforço grande do browser que, por sua vez, pode travar ou não apresentar um tempo de resposta muito bom.

Entretanto, ao contrário de outras linguagens de programação *server side*, o JavaScript não tem uma função equivalente à *sleep(1000)* para pausar a execução por um certo período de tempo. A solução, neste caso, consiste em criar filas de tarefas em conjunto com a programação assíncrona via função *setTimeout*. Os primeiros passos consistem em:

· Otimizar o loop para que o mesmo execute em menos de 100 milissegundos: o tempo máximo de resposta para não afetar a experiência do usuário;

· Deixar a maior parte do processamento a cargo do servidor;

· Usar *webworkers*, uma forma simples de executar scripts em threads de fundo sem interferir na interface do usuário.

A vantagem é que eles permitem, inclusive, efetuar operações de I/O usando o objeto XMLHttpRequest e se comunicar com o código JavaScript que o criou via mensageria. Porém, nem todos os browsers oferecem suporte a este tipo de recurso, além de o mesmo não estar habilitado a acessar propriedades do objeto DOM;

· Ponha códigos de espera dentro do corpo do loop de forma a permitir que o mesmo respire a cada iteração.

Com base nisso, uma possível solução para o problema seria:

```
for (var i = 0; i < vetor.length; i++) {
   setTimeout(processarItens(vetor[i]), 20);
}
```

Há dois problemas nessa implementação: o primeiro se refere às chamadas de callbacks dentro de loops conforme vimos antes; e o segundo se refere ao intervalo de tempo estipulado entre a execução de cada função, isto é, em vez de configurar um intervalo real entre cada chamada, nosso código irá agendar uma lista de *jobs* (um para cada item) para serem executados na fila de eventos todos de uma vez.

Assim, nossa fila de eventos terá de trabalhar duro do mesmo modo que teria se não tivéssemos usado a *setTimeout*. Vejamos a solução exibida na **Listagem 6**. O segredo está em remover a estratégia do loop com o *for* e focar no estilo assíncrono da implementação.

Na primeira linha criamos uma cópia do vetor uma vez que iremos modificá-lo. Em seguida, dentro da função *processarProxItem()* recuperamos um por um os elementos do mesmo vetor via método *shift()* dos *arrays* em JavaScript. Se ele for diferente de null, o enviamos para ser processado pela função específica e chamamos, em seguida, o método *setTimeout* para dar ao browser uma lacuna de tempo de 10 milissegundos, assim o efeito todo é feito até o fim usando somente recursividade.

Listagem 6. Solução para loop sem o for.

```
var fila = items.slice(0);

function processarProxItem(){
    var proxItem = fila.shift();
    if (proxItem){
        processarItem(proxItem);
        setTimeout(processarProxItem, 10);
    }
}
processarProxItem();
```

A abordagem melhorou, já que agora o browser não irá mais travar e o usuário pode interagir com outras ações da página normalmente.

Porém, ela será 10 vezes mais demorada que a abordagem anterior, pois adiciona um timeout de dez milissegundos ao tempo total de cada execução. Uma forma de melhorar esse tempo de processamento é fazendo uso dos recursos dos *batches* que, por sua vez, executam várias tarefas ao mesmo tempo em blocos aumentando assim a performance do código.

Vejamos o código da **Listagem 7**. Note que a estrutura não muda muito, com exceção do loop *while* que configuramos. Sua execução será baseada no tempo de início do processamento que tomará no máximo os 100 milissegundos que definimos antes para resposta máxima ao usuário.

Assim, configuramos a execução de vários blocos ao mesmo tempo em vez de somente um por vez.

Listagem 7. Solução para loop com recurso de batches.

```
var fila = items.slice(0);

function processarProxBatch(){
    var proxItem, tempoInicio = +new Date;
    while(tempoInicio + 100 >= +new Date) {
        proxItem = fila.shift();
        if (!proxItem) return;
        processarItem(proxItem);
    }
    setTimeout(processarProxBatch, 10);
```

```
}
processarProxBatch();
```

Programando sem um compilador

Para os desenvolvedores que estão acostumados com linguagens de programação compiláveis como Java, C# ou C++, a natureza fracamente tipificada do JavaScript pode trazer muitas questões e eventuais frustrações. Dentre as várias reclamações sobre a mesma, temos:

· O JavaScript não captura erros óbvios como incompatibilidade de tipos em parâmetros de funções ou nomes de variáveis/funções digitados de forma incorreta;

· Algumas bibliotecas populares, como o jQuery, silenciosamente ignoram entradas de dados erradas em vez de lançar exceções/erros.

A grande questão é saber que linguagens dinâmicas têm abordagens diferentes das linguagens compiladas e seus usuários tendem a usar diferentes padrões de uso, em consequência. Se estiver programando JavaScript do mesmo modo que fazia com Java, então provavelmente você estará fazendo isso errado.

Apesar do processo de compilação ser deveras mais lento, ele pode encontrar erros usando a checagem de tipos, por exemplo, e irá parar a execução do código neste exato momento, com uma mensagem intuitiva explicando o porquê do erro e como o desenvolvedor pode resolvê-lo.

As linguagens dinâmicas como o JavaScript, por outro lado, são interpretadas, isto é, precisam ter seu código executado para verificar os erros do programa. Essa é uma desvantagem em relação às linguagens com sintaxe de erros bem definida e processada em tempo de compilação, já que se assegura que os erros não escapem e atinjam a interface do usuário (UI).

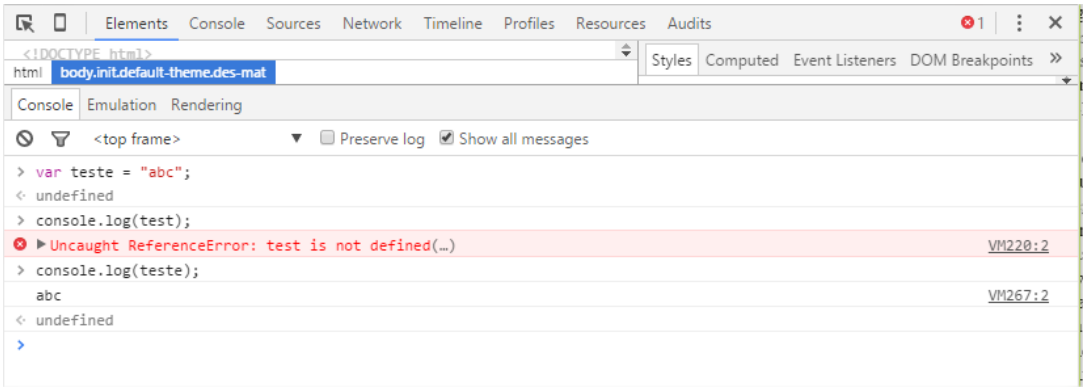
A melhor forma de lidar com isso é através dos Consoles Interativos, comumente chamados de Consoles REPL (de *read-eval-print-loop*), que nada mais são que aqueles disponibilizados nos browsers através de ferramentas do desenvolvedor (no Firefox temos o FireBug, no Chrome a ferramenta proprietária, etc.).

Elas funcionam como ferramentas de linha de comando para Unix ou Windows: você digita o comando, pressiona *enter*, ela executa o comando, imprime o resultado e então você faz tudo novamente.

Além disso, elas são ótimas para analisar bits de código em pequenas quantidades que apresentam dificuldade para funcionar devidamente, como em expressões regulares ou para examinar o DOM na página usando a API DOM. Também funcionam muito bem para trabalhar com bibliotecas de terceiros como jQuery, por exemplo.

Veja na **Figura 2** um exemplo de debug feito na Ferramenta do Desenvolvedor do Google Chrome (via atalho F12), onde criamos uma variável de nome "teste" e depois tentamos imprimir no console com um nome diferente.

A ferramenta captura a exceção e imprime uma mensagem intuitiva explicando o que ocorreu: *test is not defined*. Em seguida, efetuamos a impressão com o nome correto e a operação acontece sem erros.



abrir imagem em nova janela

Figura 2. Exemplo de debug com Console Interativo no Chrome.

Objetos e Herança

Enquanto alguns usuários de JavaScript nunca conhecerão sobre protypes ou a natureza orientada a objetos da linguagem, aqueles que vieram do tradicional modelo OO de programação certamente ficarão confusos quanto à forma como implementamos herança no JavaScript.

Talvez a maior confusão venha por parte dos frameworks JS que têm seus próprios *helpers* para escrever código baseado no modelo de classes e, assim, chegamos a seguinte conclusão: não existe apenas um jeito de fazer isso. Além disso, para complementar, os desenvolvedores não entendem por completo os conceitos tão a fundo quanto deveriam.

A essência da herança via prototypes é bem simples e se baseia em alguns conceitos também básicos:

· Um objeto *a* pode herdar de um outro objeto *b*. *b*, por sua vez, é dito protótipo de *a*.

· *a* herda todas as propriedades de *b*, isto é, se o valor *b.nome* for igual a "devmedia", então *a.nome* terá o mesmo valor automaticamente.

· As propriedades nativas de *a* devem sobrescrever as de *b*.

Consideremos um exemplo de objeto anônimo de tipo Pessoa e que contenha dois atributos: um *nome* e um *sobrenome*, tal como na **Listagem 8**.

Veja que definimos o mesmo atributo de nome para ambos os objetos, porém o segundo não consta de nenhum sobrenome. Todavia, estamos estendendo o segundo objeto do primeiro via operador *__proto__*, propriedade interna a todos os objetos no JavaScript que serve exatamente para definir a herança dos mesmos. Assim, quando acessarmos a propriedade sobrenome do segundo objeto o valor "Galdino" estará automaticamente associado à mesma.

Listagem 8. Exemplo de herança via prototypes.

```
var fabricio = {
          nome: 'Fabricio',
          sobrenome: 'Galdino'
}
var fabricio_filha = {nome: 'Joana'}
fabricio_filha.__proto__ = fabricio;
```

Supondo que, no exemplo, Joana se case e receba um novo sobrenome. Logo, teríamos o seguinte trecho de código:

```
fabricio_filha.sobrenome = 'Souza';
```

E o valor seria então sobrescrito no objeto *fabricio_filha*. Caso Joana se divorcie e deseje remover o seu sobrenome para voltar ao antigo, basta remover a propriedade e a herança se encarregará de fazer a associação automática com o elemento pai novamente:

```
delete fabricio_filha.sobrenome;
```

A implementação parece bem simples, mas temos um problema em relação a ela: não podemos usar o operador *__proto__*, ao menos não poderemos em um futuro breve. Isso porque esse recurso não é suportado no Internet Explorer e sequer está presente na especificação da ECMAScript.

O navegador Firefox já considera, inclusive, a sua remoção em versões futuras. Bem, então qual estratégia usar, já que o JavaScript não tem nenhum recurso de classes que vemos em outras linguagens? A solução são os **prototypes**.

Em termos de programação genérica, um protótipo é um objeto que fornece um comportamento base para um segundo objeto. O segundo objeto pode então estender esse comportamento base para formar sua própria especialização.

Este processo, também conhecido como *herança diferencial*, difere da clássica herança na medida em que não exige uma tipificação explícita (estática ou dinâmica) ou tenta definir formalmente um tipo em função de outro. Enquanto a herança clássica tem reutilização planejada, a verdadeira herança prototípica é oportunista.

Em JavaScript, cada objeto faz referência a um objeto protótipo a partir do qual ele pode herdar propriedades. Protótipos JavaScript são excelentes instrumentos para reuso: uma única instância de protótipo pode definir propriedades para um número infinito de instâncias dependentes. Protótipos também podem herdar de outros protótipos, formando cadeias de protótipos.

Porém, em comparação com a emulação no Java, o JavaScript amarra a propriedade *prototype* ao construtor. Como consequência, mais frequentemente do que não, objetos de vários níveis de herança são alcançados através do encadeamento de protótipos baseados em construtores.

No fim, o encadeamento de protótipos baseados em construtores requer planejamento inicial e resulta em estruturas que se assemelham mais de perto às hierarquias tradicionais de linguagens clássicas: construtores representam tipos (classes), cada tipo é definido como um subtipo de um (e apenas um) supertipo e todas as propriedades são herdadas através desse tipo de cadeia.

A palavra-chave *class* meramente formaliza a semântica existente. Deixando de lado todas essas caraterísticas sintáxicas, o JavaScript tradicional é claramente menos prototipável do que alguns afirmam.

Numa tentativa de oferecer um maior suporte aos protótipos, a especificação ES5 da ECMAScript introduziu o *Obejct.create*. Este método permite que um protótipo seja atribuído a um objeto diretamente e, portanto, liberta os protótipos JavaScript de construtores de modo que, em teoria, um objeto possa adquirir comportamento a partir de qualquer outro objeto arbitrário e estar livre das restrições do *typecasting* (conversão dos tipos). Vejamos na **Listagem 9** um exemplo de objeto clássico de círculo com suas respectivas funções.

Listagem 9. Exemplo de objeto via Object.create().

```
var circulo = Object.create({
          area: function() {
                    return Math.PI * this.radius * this.radius;
          },
          crescer: function() {
                    this.radius++;
          },
          encolher: function() {
                    this.radius--;
          }
});
```

O *Object.create* aceita um segundo argumento opcional que representa o objeto a ser herdado. Infelizmente, em vez de aceitar o objeto em si (na forma de um literal, variável ou argumento), o método espera uma definição completa da propriedade *meta*. Por exemplo, se quisermos definir a propriedade *radius* do exemplo anterior no objeto pai, precisaríamos instanciar o mesmo da forma como está definida na **Listagem 10**.

Listagem 10. Exemplo de herança via Object.create().

```
var circulo = Object.create({
          area: function() {
                    return Math.PI * this.radius * this.radius;
          },
          crescer: function() {
                    this.radius++;
          },
          encolher: function() {
                    this.radius--;
          }, {
                    radius: {
                       writable:true, configurable:true, value: 7
                    }
});
```

Partindo do princípio de que ninguém realmente usa esse tipo de código no mundo real, tudo o que resta é atribuir manualmente as propriedades para a instância depois de ter sido criada. Mesmo assim, a sintaxe *Object.create* só permite que um objeto possa herdar as propriedades de um protótipo. Em cenários reais, muitas vezes queremos adquirir o comportamento do protótipo a partir de múltiplo objetos: por exemplo, uma pessoa pode ser um empregado e um gerente ao mesmo tempo.

Mixins

Felizmente, o JavaScript oferece alternativas viáveis para o encadeamento de heranças. Em contraste aos objetos das linguagens mais rigidamente estruturadas, os objetos JavaScript podem chamar qualquer propriedade das funções, independentemente da linhagem. Em outras palavras, as funções JavaScript não precisam ser hereditárias para serem visíveis.

A abordagem mais básica para o reuso de funções é a delegação manual – qualquer função pública pode ser chamada diretamente através das funções *call* ou *apply* que vimos antes. É um recurso poderoso e facilmente esquecido.

Tradicionalmente, um **mixin** é uma classe que define um conjunto de funções que seriam, em outros cenários, definidas por uma entidade concreta (uma pessoa, um círculo, um observador).

Entretanto, as classes mixin são consideradas abstratas já que não serão instanciadas por si sós, em vez disso suas funções são copiadas (ou emprestadas) por classes concretas como uma forma de herdar comportamento sem ter de entrar em um relacionamento formal com o fornecedor do mesmo comportamento. No fim, temos uma funcionalidade interessante que nos permite usar objetos (instâncias) que oferecem clareza e flexibilidade: nosso mixin pode ser um objeto regular, um protótipo, uma função, etc. seja qual for, o processo torna-se transparente e óbvio.

Tomando como referência o mesmo exemplo de classe Círculo que criamos, vejamos como transformar nosso código de herança em um mixin (**Listagem 11**). Isto é o mais próximo que podemos chegar ao modelo de classes no JavaScript. Para tanto, o uso da propriedade *prototype* se faz essencial para permitir a definição da estrutura hierárquica no referido objeto.

Listagem 11. Exemplo de herança com círculo via mixin.

```
var Circulo = function() {};
Circulo.prototype = {
          area: function() {
                    return Math.PI * this.radius * this.radius;
          },
          crescer: function() {
                    this.radius++;
          },
          encolher: function() {
                    this.radius--;
          }
});
```

Também é possível simplificar essa chamada simplesmente declarando o objeto como uma variável via palavra-chave *var*. Assim, nosso código pode ficar ainda mais simples. Veja na **Listagem 12** um exemplo de classe que mapeia os possíveis eventos de um botão.

Listagem 12. Exemplo de mixin simplificado.

```
var clickableFunctions = {
          hover: function() {
                    console.log('hover');
          },
          pressionar: function() {
                    console.log('pressionando botão');
          },
          soltar: function() {
```

```
                        console.log('soltando botão');
            },
            disparar: function() {
                        this.action.fire();
            }
};
```

Como é possível então que um objeto mixin se integre ao seu objeto? Por meio de uma função *extend* que implementa a herança associando-a aos dois objetos em questão. Normalmente, a função *extend* simplesmente copia (não clona) as funções mixin para o objeto receptor.

Existem algumas pequenas variações nesta implementação dependendo do desenvolvedor/empresa que a emprega. Por exemplo, o framework *Prototype.js* omite uma checagem à propriedade *hasOwn* do objeto *Property* sugerindo que o mixin não tem propriedades enumeráveis em sua cadeia de protótipos), enquanto outras versões supõem que o desenvolvedor deseja copiar somente os mixin de protótipo do objeto.

Vejamos na **Listagem 13** um exemplo comum dessa função. Note que passamos ambos os objetos de destino e fonte (este último deve ser um *array* em vista da quantidade de objetos filhos que desejam herdar do objeto pai) como parâmetros à função que, por sua vez, se encarrega de iterar sobre a lista de fontes verificando se cada uma tem a propriedade de destino interna ao seu conteúdo (via função JavaScript *hasOwnProperty*) e adicionado a mesma em caso positivo.

Listagem 13. Exemplo da função de mixin extend().

```
function extend(destino, fonte) {
 for (var chave in fonte) {
    if (fonte.hasOwnProperty(chave)) {
        destino[chave] = fonte[chave];
    }
 }
 return destino;
}
```

Agora podemos estender um protótipo base a partir dos dois mixins que criamos anteriormente para criar um botão redondo, por exemplo. Vejamos o código da **Listagem 14**. Note que criamos apenas um objeto simples com duas propriedades: uma dimensão de raio do botão e uma label, em seguida, chamamos a função *extend* para associar as funcionalidades implementadas na Circulo e clickableFunctions, respectivamente. No fim, fazemos um teste básico instanciando um objeto de mesmo tipo e chamando alguns de seus métodos, de forma muito semelhante a que temos em outras linguagens OO.

Listagem 14. Exemplo de herança com dois mixins criados.

```
var BotaoCircular = function(radius, label) {
this.radius = radius;
this.label = label;
};

extend(BotaoCircular.prototype, Circulo);
extend(BotaoCircular.prototype, clickableFunctions);

var botaoCircular = new BotaoCircular(3, 'Enviar');
```

```
botaoCircular.crescer(); // Aumenta em 1 o valor da propriedade radius
botaoCircular.disparar(); // Dispara o evento no botão
```

Padrões e Modelos

Os frameworks MVC – ou MVW (*Model, View, "Whatever"*) – existem nas mais diversas formas e tipos pela comunidade front-end. Todavia, apesar de duas diferentes abordagens que dificultam a componentização e o aproveitamento das estruturas modulares, eles nos fornecem componentes fundamentais para um código padronizado: os **modelos**, que "modelam" os dados associados à aplicação. Nas aplicações web baseadas em um cliente, tais modelos geralmente são representados por um objeto de banco de dados hospedado no servidor.

Um bom exemplo disso é o framework MVC minimalista Backbone.js que, apesar de constantemente criticado pela sua camada de visão nada sofisticada, fornece um excelente gerenciamento interno dos modelos.

Por exemplo, na **Listagem 15** vemos um exemplo simples de modelo no Backbone. Veja que ele aplica o mesmo conceito de função *extend* que vimos anteriormente, além de usar uma notação de objetos igual à do JSON. Podemos ver também a distribuição uniforme dos atributos com seus respectivos valores iniciais, assim como o atributo externo *idAtributo* que contém o valor do id desse elemento no DOM. A função *nomeCompleto*, por sua vez, retorna os valores concatenados do nome e sobrenome do usuário via variável *this*, conforme também havíamos visto.

Listagem 15. Exemplo de modelo definindo no framework Backbone.

```
var Usuario = Backbone.Model.extend({
    defaults: {
      usuario: '',
      nome: '',
      sobrenome: ''
    },
      idAtributo: 'usuario',
      nomeCompleto: function () {
          return this.get('nome') + this.get('sobrenome');
     }
});
```

Na **Listagem 16** vemos um código de exemplo que faz uso desse modelo, inicializando-o, além de mostrar como a sua instância deve ser usada na aplicação. Veja que estamos apenas criando um novo objeto de tipo *Usuario* e inicializando suas propriedades para, em seguida, modificar uma delas e analisar como o framework reage a isso.

Apesar de simples, esse exemplo é o suficiente para entender como os modelos *cliente-side* funcionam e como eles interagem com os moldes de aplicações MVC.

Listagem 16. Exemplo de uso do modelo Backbone definido.

```
var usuario = new Usuario({
 usuario: 'fabricio_galdino',
 nome: 'Fabricio',
 sobrenome: 'Galdino'
```

```
});

usuario.nomeCompleto(); // Retorna "Fabricio Galdino"
usuario.set('nome', 'João');
usuario.save(); // Envia as mudanças para o endpoint no servidor
```

Adicionalmente, o Backbone fornece as chamadas classes "collection", as quais auxiliam os desenvolvedores a manipular facilmente conjuntos de instâncias de modelos comuns. Podemos pensar nelas como espécies de *arrays* superdotados, carregadas de funções utilitárias, tal como exibido na **Listagem 17**. Veja como é simples criar um tipo de dado a partir de outro, herdando automaticamente todas as suas características.

Como se trata de um modelo Backbone baseado no servidor, precisamos referenciar tanto o modelo de entidade (*Usuario*) quanto a URL onde o mesmo está hospedado. O método *fetch()* se encarrega de buscar todos os dados via *request* HTTP, enquanto o método *get()* busca um usuário em específico pelo seu atributo id.

Listagem 17. Exemplo de uso das collections do Backbone.

```
var UsuarioCollection = Backbone.Collection.extend({
  model: Usuario,
  url: '/usuarios'
});

var usuarios = new UsuarioCollection();
usuarios.fetch(); // Carrega os dados do usuário via HTTP
var fabricio = usuarios.get('fabricio_galdino'); // Encontra pelo idAtributo
```

Nem todos os frameworks MVC implementam uma classe *Collection* como o Backbone. Por exemplo, o Ember.js define uma classe *CollectionView* que similarmente mantém um conjunto de modelos comuns, mas amarra a manipulação ao objeto DOM. Independente do framework adotado, é sabido que precisamos manipular não somente objetos, classes e suas heranças, como também coleções destes mesmos objetos. E para isso, o uso de um framework adequado pode facilitar muito esse tipo de implementação.

Quando trabalhamos com aplicações de porte grande ou até médio, é comum ter múltiplas instâncias de modelos representando alguns objetos de banco do servidor. Isso geralmente acontece quando você tem múltiplas *views* de algum dado, de modo que um modelo aparece em duas ou mais *views*.

Considere o seguinte exemplo, que introduz duas novas coleções de usuários: *Seguidores*, para usuários que estão seguindo um dado usuário (em uma rede social, por exemplo) e *Seguidos*, para os usuários que estão sendo seguidos por outros. Um usuário que é tanto seguidor quanto está sendo seguido aparecerá em ambas as coleções, neste caso teremos instâncias duplicadas do mesmo modelo. Vejamos o código na **Listagem 18**.

Listagem 18. Exemplo Seguidores/Seguidos no Backbone.

```
var SeguidosCollection = UserCollection.extend({
          url: '/seguindo'
});

var SeguidoresCollection = UserCollection.extend({
```

```
          url: '/seguidores'
});

var seguindo = new SeguidosCollection();
var seguidores = new SeguidoresCollection();

seguindo.fetch();
seguidores.fetch();

var usuario1 = seguindo.get('fabriciogaldino');
var usuario2 = seguidores.get('fabriciogaldino');
usuario1 === usuario2; // false
```

Ter múltiplas instâncias de um mesmo modelo traz duas desvantagens principais. A primeira é que assim estamos usando memória adicional para representar o mesmo objeto. Dependendo da complexidade do modelo e do tamanho dos atributos que o mesmo contiver, pode não ser interessante consumir kilobytes de memória extra para tal.

Se as instâncias forem duplicadas dezenas ou centenas de vezes (um cenário bem possível tratando-se de uma aplicação com alta escalabilidade) elas podem rapidamente consumir toda a memória disponível.

A segunda é que se os usuários modificarem um destes modelos no cliente, outras instâncias dos mesmos serão dessincronizadas. Isso pode ser feito de várias formas: como através do usuário mudando o estado do objeto via UI, ou via *update* criado por outro usuário e enviado ao cliente via serviço em tempo real:

```
usuario1.set('nome', 'João');
usuario2.get('nome'); // ainda será João
```

Nesse mesmo exemplo onde o mesmo usuário aparece em duas coleções diferentes, torna-se trivial a atualização de ambas as instâncias de forma manual com a nova propriedade. Entretanto, em aplicações reais com vários usuários ao mesmo tempo, é inimaginável a quantidade de vezes que esse mesmo objeto de usuário possa aparecer em dezenas ou centenas de coleções diferentes.

Em função disso, uma solução comum para lidar com a duplicata de instâncias é fazer uso de uma função de fábrica quando criarmos uma nova instância do modelo. Se a fábrica detectar que uma instância do modelo já existe, ela a retornará em vez de criar uma nova.

Vejamos o código da **Listagem 19**. Note como é simples definir um objeto em nível global de cache e salvar todas as novas instâncias no mesmo, assim sempre que formos criar um novo objeto do tipo Usuario, chamamos a classe UsuarioFabrica para lidar com a verificação do cache e respectivo retorno do objeto cacheado ou criação de um novo se for o caso. Toda a checagem gira em torno do atributo *usuario* que é passado por parâmetro ao construtor da fábrica.

Listagem 19. Exemplo de fábrica de usuários.

```
var cacheUsuario = {};

function UsuarioFabrica(attrs, options) {
          var usuario = attrs.usuario;
          return cacheUsuario[usuario] ? cacheUsuario[usuario] : new Usuario(attrs, options);
}
```

```
var usuario1 = UsuarioFabrica({ usuario: 'fabriciogaldino' });
var usuario2 = UsuarioFabrica({ usuario: 'fabriciogaldino '});
usuario1 === usuario2; // true
```

Para fazer um uso eficiente deste padrão é sempre aconselhado usar a função de fábrica em conjunto para criar novas instâncias. Todavia, esse tipo de abordagem é dificultado quando precisamos implementá-la em bases de código que não são de nossa responsabilidade, como em bibliotecas de terceiros e plugins, por exemplo.

Considere, por exemplo, a função *Collection.prototype._prepareModel* do código fonte do Backbone. Ele usa esta função para "preparar" e, no fim, criar uma nova instância do modelo para adicionar à coleção. Ela é invocada para atender a uma série de necessidades, como quando precisamos popular uma coleção com modelos retornados de um recurso HTTP e seu código é apresentado na **Listagem 20**.

Listagem 20. Exemplo de uso da função prepareModel do Backbone.

```
// Prepara o hash de atributos (ou outros modelos) para serem adicionados a esta collection.
Backbone.Collection.prototype._prepareModel = function(attrs, options) {
          if (attrs instanceof Model) {
                    if (!attrs.collection) attrs.collection = this;
                    return attrs;
          }

          options || (options = {});
          options.collection = this;
          var modelo = new this.model(attrs, options);
          if (!modelo._validate(attrs, options)) {
                    this.trigger('invalid', this, attrs, options);
                    return false;
          }
          return modelo;
};
```

Atente para o código que faz uso do operador *new* para criar um novo modelo passando os *atributos* e *options* como parâmetros. É ele o responsável por criar a nova instância e associar a mesma à coleção. **this.model** é uma referência ao construtor da classe de modelo que a coleção encapsula. Ele é especificado quando definimos uma nova classe de coleção, tal como:

```
var UsuarioCollection = Backbone.Collection.extend({
 model: Usuario, url: '/usuarios'
});
```

O mais interessante nessa abordagem é que em vez de passarmos a classe Usuario à definição da coleção, podemos passar a classe UsuarioFabrica (a nossa função de fábrica que retorna instâncias de modelo únicas):

```
var UsuarioCollection = Backbone.Collection.extend({
model: UsuarioFabrica, url: '/usuarios'
});
```

Dessa forma, a UsuarioFabrica será atribuída ao *this.model* e será invocada pelo operador *new* quando a collection criar uma nova instância:

```
var modelo = new this.model(attrs, options); // this.model é a UserFactory
```

Várias são as técnicas e estratégias que o leitor pode seguir para implementar código em JavaScript. A maioria dos desenvolvedores seguem a sua própria especificação e ignoram muitos conceitos já explanados por outros profissionais que otimizam o uso da linguagem como um todo.

A melhor maneira de se aprofundar nestes conceitos é entendendo como o JavaScript funciona de fato, seus principais conceitos e como a orientação a objetos se aplica nessa linguagem em contraste às demais.

A especificação da ECMAScript (marca registrada que especifica linguagens para *client-scripting* na web como JavaScript, JScript e ActionScript) também traz uma série de informações importantes que, além de instruir sobre as melhores práticas associadas ao JavaScript, te manterá atualizado sobre as últimas novidades, padrões e regras da linguagem. O resto é experiência traduzida em prática.

Links

Página da ferramenta JSHint.
http://jshint.com/

Página oficial da especificação ECMAScript.
http://www.ecma-international.org/publications/standards/Ecma-262.htm

Fabricio Galdino