PROGRAMAÇÃO FRONT END II

Maikon Lucian Lenz

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Herança

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Descrever o `Object.prototype`.
- Reconhecer os métodos associados ao `Object.prototype`.
- Desenvolver aplicações com conceito de herança.

Introdução

Herança é um conceito fundamental da programação orientada a objetos porque, assim como o próprio conceito de objeto, é facilmente observada nas relações entre quaisquer entidades reais. Graças ao seu uso, é possível reaproveitar código, controlar atribuições de maior nível hierárquico e repassar características entre os objetos de maneira flexível, já que o JavaScript é fracamente tipado e admite mudanças estruturais em tempo de execução.

Neste capítulo, você conhecerá o objeto `Object.prototype`, que serve de base para os demais objetos. Para tanto, a linguagem se utiliza do conceito de encadeamento de protótipos, que garante ainda mais flexibilidade ao mecanismo de herança. Serão apresentados os métodos e as propriedades herdadas por cada novo objeto a partir do `Object.prototype` e o modo como a herança pode facilitar ou dificultar o desenvolvimento em alguns casos.

`Object.prototype`

Parte essencial de qualquer programação orientada a objetos é a relação entre eles, que pode ser implementada por meio de diferentes maneiras, com ou sem interações e dependências. Em geral, um objeto pode ser: dependente agregado, composto ou uma especialização de outro objeto (MACHADO; FRANCO; BERTAGNOLLI, 2016).

Esse último tipo de relação cria uma hierarquia entre objetos. Em outras palavras, estabelece uma relação de herança entre eles.

Se adequado, um modelo de objetos pode aproveitar as relações de herança entre os mesmos para reaproveitamento de código. Faz-se isso mediante o estabelecimento dos objetos que apresentam características em comum e podem delegar métodos e propriedades para um objeto de nível maior na hierarquia que poderão herdar simultaneamente.

O exemplo a seguir apresenta um pseudocódigo com dois objetos que compartilham métodos e propriedades similares.

Exemplo

Exemplo 1

```
Carro
{
    Modelo: "",
    Fabricante: "",
    Ano: 0,
    Cor: "",
    Abastecer()
}

Moto
{
    Modelo: "",
    Fabricante: "",
    Ano: 0,
    Cor: "",
    Abastecer()
}
```

Repare que todas as propriedades e os métodos existentes no objeto `Carro` também estão presentes no objeto `Moto`. Utilizando mecanismos de herança, seria possível criar um terceiro tipo de objeto que elencasse todos esses elementos e deixasse para os objetos individuais apenas as características únicas dos mesmos, como no exemplo a seguir.

Exemplo

Exemplo 2

```
Veiculo
{
    Modelo: "",
    Fabricante: "",
    Ano: 0,
    Cor: "",
    Abastecer()
}

Carro é filho de Veiculo
{
    Radio: false
}

Moto é filho de Veiculo { }
```

Observe a quantidade de código que foi reaproveitada. No caso do objeto `Carro`, além das características herdadas do objeto `Veiculo`, é declarada uma propriedade exclusiva do objeto que não existirá no objeto `motor (Radio)`.

Em JavaScript, propriedades e métodos são herdados a partir de um objeto protótipo. Na verdade, qualquer objeto em JavaScript está vinculado a um objeto protótipo (FLANAGAN, 2013).

Todo novo objeto, seja instanciado de maneira literal (utilizando `{}`) ou a partir de um construtor explícito (`new`), herda as propriedades e os métodos do objeto nativo `Object.prototype`, sendo este a origem de todos os demais, já que o objeto `Object.prototype` não herda nada de nenhum.

Outros protótipos podem ser criados e utilizados durante a criação de novos objetos. Nesse caso, a nova instância herdará elementos de ambos os protótipos, uma vez que o utilizado foi criado a partir de um `Object.prototype` também. Essa situação é definida como encadeamento de protótipos (FLANAGAN, 2013).

O padrão ECMAScript define desde a versão 5 uma função estática para criar objetos, definindo de maneira explicita o protótipo a ser utilizado. É a função `Object.create()` que recebe como argumentos o objeto que servirá de protótipo e, opcionalmente, descrições das propriedades como segundo argumento (ECMA INTERNATIONAL, 2015), conforme exemplo a seguir.

Exemplo

Exemplo 3:

```
var obj1 = {}                        // Nenhum dos objetos
criados
var obj2 = { valor1: 1, valor2: 2 } // possui propriedades
var obj3 = new Object();
var obj4 = Object.create({})
var obj5 = Object.create(null)
var obj6 = Object.create(Object.prototype)
```

Simplificando, Flanagan (2013) define a capacidade de criar um objeto especificando um protótipo como a capacidade de criar um herdeiro.

Essa estrutura de encadeamento de protótipos também é utilizada durante atribuições e consultas de valor às propriedades de um dado objeto. Sempre que uma propriedade é acessada, é consultado o próprio objeto e seus protótipos em busca dela.

Se o objeto especificado apresenta a propriedade que se está tentando consultar, o valor dela será retornado. Do contrário, essa propriedade será buscada no protótipo do objeto. Caso ainda não seja encontrada, e o protótipo do objeto contenha um protótipo, este será o próximo alvo. Esse fenômeno se repete até que a propriedade desejada seja encontrada ou não existam mais protótipos daquele objeto para serem consultados (FLANAGAN, 2013).

Fique atento

Junto das propriedades, são herdados, também, seus atributos.

Durante atribuições, o processo é similar, porém, ao atribuir um valor a uma propriedade inexistente no objeto, mas presente em algum protótipo dele, é criada uma propriedade — mesmo que esse identificador já exista em níveis maiores de hierarquia. No exemplo seguinte, é possível constatar cada uma das situações.

Exemplo

Exemplo 4:

```
const obj1 = { a: 1 } // cria um objeto de 1 propriedade
obj1.b = 2                                    // cria uma nova
const obj2 = Object.create(obj1) // cria um herdeiro de obj1
console.log(obj2.a, obj2.b)  // imprime valores do protótipo
obj2.b = 3                   // cria propriedade b própria
console.log(obj2.a, obj2.b) // valores do protótipo e do objeto
console.log(obj1.a, obj1.b) // exibe 1 e 2, atribui-
ções no filho
                                  // não interferem o pai
```

Assim, é evidente a importância que a herança exerce na programação orientada a objetos e, principalmente, para JavaScript, que tem no objeto o seu ponto fundamental. A forma como JavaScript lida com a herança assemelha-se em muito às demais linguagens, exceto pela sua característica de não atribuir valores aos protótipos quando não encontrada a propriedade solicitada no objeto original. O que parece ser uma limitação é, na verdade, um recurso poderoso, que permite não apenas criar instâncias especializadas de um protótipo, como também anular propriedades herdadas de forma simplificada.

O protótipo de objeto (`Object.prototype`) contém métodos próprios que, junto dos construtores, garantem a flexibilidade dos demais objetos que o herdarão.

Métodos associados ao protótipo de objeto

Os métodos disponíveis em `Object.prototype` podem ser alteráveis, sobrescrevendo-os ou não. No primeiro caso, os métodos têm funções diretamente ligadas à estrutura de um objeto genérico, como enumeração de propriedades ou verificação de pertencimento a protótipos. Já no segundo caso, as funções garantem que o método, apesar de sempre presente, também possa ser alterado por objetos mais especializados – é o caso, por exemplo, dos métodos de apresentação textual de dados do objeto.

Método `hasOwnProperty`

Este método retorna `true`, se o parâmetro passado corresponde a uma propriedade do próprio objeto, ou `false`, se a propriedade não existe ou foi herdada de algum protótipo (ECMA INTERNATIONAL, 2015).

O parâmetro pode ser do tipo `string` ou `Symbol`, e o retorno sempre será do tipo `Boolean`.

É um método que se contrapõe ao operador in, que resulta em verdadeiro para qualquer propriedade acessível a partir do objeto testado, inclusive aquelas herdadas.

Exemplo

Exemplo 5:

```
const obj1 = { a: 1 }           // cria um objeto, 1 propriedade
const obj2 = Object.create(obj1)     // cria um herdeiro de obj1
obj2.b = 2               // cria uma propriedade exclusiva
                                                  // de obj2
console.log("a" in obj2)                  // retorna true
console.log("b" in obj2)                  // retorna true
console.log(obj2.hasOwnProperty("a")) // retorna false
console.log(obj2.hasOwnProperty("b")) // retorna true
```

O objeto `obj2` do exemplo anterior possui a propriedade `a` herdada de `obj1` e a propriedade `b` própria dele. Para o operador `in`, ambas as propriedades `a` e `b` podem ser acessadas no `obj2` e, portanto, o retorno é verdadeiro. Já para o método `hasOwnProperty()`, somente `b` retorna verdadeiro porque é a única propriedade definida pelo próprio objeto `obj2`.

Método `isPrototypeOf`

Retorna `true` ou `false` sempre que o objeto referenciado pertencer à cadeia de protótipos desse objeto, conforme exemplo a seguir (ECMA INTERNATIONAL, 2015).

Exemplo

Exemplo 6:

```
const obj1 = { }                          // cria um objeto
const obj2 = Object.create(obj1)      // cria um herdeiro de obj1
const obj3 = Object.create(obj2)      // cria um herdeiro de obj2
console.log(obj1.isPrototypeOf(obj2))      // retorna true
```

```
console.log(obj1.isPrototypeOf(obj3))      // retorna true
console.log(obj2.isPrototypeOf(obj3))      // retorna true
console.log(obj3.isPrototypeOf(obj2))     // retorna false
console.log(obj3.isPrototypeOf(obj1))     // retorna false
console.log(obj2.isPrototypeOf(obj1))     // retorna false
```

Método `propertyIsEnumerable`

Um dos atributos que as propriedades recebem é a capacidade de serem enumeradas (`isEnumerable`). Esse atributo permite controlar a varredura de um objeto por laços, como o `for...in`.

Caso a propriedade seja passível de enumeração, será encontrada durante uma varredura, e o método `propertyIsEnumerable` retornará verdadeiro ao testá-la, conforme o seguinte exemplo (ECMA INTERNATIONAL, 2015).

Exemplo

Exemplo 7:

```
const obj1 = { a: 1, b: 2 }                  // cria um objeto
console.log(obj1.propertyIsEnumerable("a")) // retorna true
console.log(obj1.propertyIsEnumerable("b"))  // retorna true
console.log(obj1.propertyIsEnumerable("length")) // re-
torna false
for (x in obj1)
{
console.log(x)                       // exibe a e b, ignora
                           // length e outras propriedades
                                        // não enumeráveis
}
```

Apesar de o objeto `obj1` herdar outras propriedades do protótipo, nenhuma delas é enumerável. Logo, somente as propriedades que foram criadas para o próprio objeto serão listadas.

Método `toString` e `toLocaleString`

O método `toString` retorna o objeto como uma `string`. A variação `toLocaleString`, por padrão, chama o método `toString`, mas é disponibilizada para permitir que, em determinadas situações, seja sobrescrita para personalizar o método para objetos específicos, como mostra o exemplo a seguir (ECMA INTERNATIONAL, 2015).

Exemplo

Exemplo 8:

```
var obj1 = { a: 0, b: 1, c: 2 }
console.log(obj1.toString())          // exibe [object Object]
console.log(obj1.toLocaleString())    // exibe [object Object]
```

Ambos os métodos não recebem parâmetros e, da forma como são implementados pelo `Object.prototype`, não permitem grande utilização, se não para verificar a classe do objeto em questão. No entanto, lembre-se de que esses métodos podem ser sobrescritos e especializados conforme a necessidade (FLANAGAN, 2013).

Método `valueOf`

Não recebe nenhum parâmetro e retorna o valor primitivo do objeto (ECMA INTERNATIONAL, 2015). É o mesmo que tentar exibir os valores do objeto de forma direta. Esse método é invocado pelo próprio JavaScript ao se tentar utilizar o objeto como um valor primitivo, como em `console.log(objeto)`.

Exemplo

Exemplo 9:

```
var obj1 = { a: 0, b: 1, c: 2 }
console.log(obj1)             // exibe { a: 0, b: 1, c: 2 }
console.log(obj1.valueOf())   // exibe { a: 0, b: 1, c: 2 }
console.log(obj1 + " 1")      // exibe [object Object] 1
console.log(obj1.valueOf() + " 2") // exibe [object Object] 2
Object.prototype.valueOf = () => "Sumiu";  // sobres-
creve o método
// valueOf()
console.log(obj1 + " 1")            //  exibe  Sumiu  1
console.log(obj1.valueOf() + " 2")  //  exibe  Sumiu  2
```

Conforme esse exemplo, esse método é personalizado para cada tipo de objeto e pode ser sobrescrito para customizar a conversão do objeto em valor primitivo. Independentemente do que estiver presente no objeto, o retorno de `valueOf` sempre será a `string` "`Sumiu`". Ao utilizar o operador + em conjunto com o objeto, JavaScript tenta converter o objeto em valor primitivo, usando `valueOf` que, agora, é modificado para exibir sempre a mesma `string`.

Além dos métodos descritos anteriormente, `Object.prototype` também carrega consigo a propriedade `constructor`, que aponta para a função utilizada na hora de instanciar novos objetos do mesmo protótipo. É comum estar implementada a propriedade `__proto__`, que aponta para o objeto do protótipo em si.

Aplicações com herança

Para que o programador amplie suas chances de sucesso em desenvolver um sistema adequado às necessidades, alguns cuidados especiais devem ser tomados ao se implementar mecanismos de herança.

Muitas vezes, um objeto pode ser mais bem aproveitado, aumentando sua dependência de outros objetos, mas o tornando mais flexível. Nesse caso, diz-se que o acoplamento é reduzido (MACHADO; FRANCO; BERTAGNOLLI, 2016).

Imagine que você precisa implementar um objeto `Moto` novamente, mas que, ao contrário do exemplo 2, a moto seja movida a eletricidade, e não a combustão. O exemplo apresentado não seria o mais adequado, uma vez que o objeto `Moto` pode ter o método `Recarregar()` implementado, mas herdaria o método `Abastecer()` também, podendo criar conflitos e erros de execução, além de ocupar memória e tempo de processamento com itens dispensáveis.

Para escapar a essa situação, pode-se sobrecarregar o método ou reorganizar a estrutura dos objetos criando um objeto `Tanque` e um objeto `Bateria` independentes de `Carro` e `Moto`. Na primeira situação, de qualquer forma, a implementação do método original deve ser pensada e construída de forma a garantir coesão com eventuais sobrecarregamentos.

Assim, é evidente que a herança é muito mais relevante para fornecer soluções genéricas entre objetos do que na resolução de problemas específicos, situações nas quais o método de composição costuma ser mais flexível e adequado.

De qualquer forma, em algumas situações, o mecanismo de herança pode solucionar grandes problemas de maneira rápida e simples. Por exemplo, se for necessário adicionar um método para todos os demais objetos ao invés de fazer isso individualmente, objeto por objeto, a herança baseada em protótipos permite que o programador adicione o novo método apenas ao protótipo de maior nível hierárquico. Consequentemente, todos os demais terão acesso ao mesmo.

O tipo de situação descrita anteriormente é possível não apenas ao encadeamento de protótipos da linguagem JavaScript, mas também devido à própria linguagem ser fracamente tipada, de forma que classes e objetos possam ser expandidos e modificados em tempo de execução sem qualquer problema.

Uma situação como essa foi apresentada no exemplo 9, em que o método `valueOf` era modificado no protótipo. Assim, todos objetos que herdam propriedades e métodos deste receberão automaticamente acesso a essa função.

Na prática, o que determina se o mais adequado é o uso ou não de herança é a experiência do programador e sua capacidade de abstração. Quanto maior for a habilidade dele em determinar a relação de dependência entre os elementos que compõem o seu sistema, melhor será a sua tomada de decisão.

É provável que inúmeras soluções possam ser imaginadas para um mesmo problema. Porém, na maior parte do tempo, as dificuldades envolvendo as relações de dependência entre objetos, se tomadas de maneira adequada ou não, revelar-se-ão somente à medida que o projeto se aproxime da sua conclusão, quando pequenas alterações podem ser excessivamente custosas devido ao mau planejamento inicial da estrutura do mesmo.

Em outras situações, decisões equivocadas podem apresentar-se limitantes apenas em uma segunda geração do seu aplicativo, quando, ao tentar criar atualizações – sejam com novas funcionalidades, sejam com correções de *bugs* –, o programador se depare com a necessidade de alterar grandes parcelas de código para modificar a estrutura dos objetos.

Como visto, a herança não é um conceito exclusivo de JavaScript, mas tem suas peculiaridades por se basear em um modelo de encadeamento de protótipos, do qual o objeto primordial é o `Object.prototype` que expõe os métodos e as propriedades mínimas necessárias para o bom funcionamento de todos os demais.

Essa relação de herança tem grande flexibilidade, assim como demais conceitos de JavaScript, e, da mesma forma, traz consigo a necessidade do planejamento claro da estrutura do programa.

A possibilidade de se sobrescrever métodos do protótipo pode ser uma solução fácil e brilhante em determinadas situações. Já em outras, pode significar a interferência inadequada em outras funcionalidades do sistema ou, até mesmo, em outros sistemas que compartilham a mesma instância do interpretador em determinados momentos – situação muito comum no desenvolvimento de páginas Web.

Referências

ECMA INTERNATIONAL. *ECMAScript® 2015 Language Specification*. 6. ed. Geneva: Ecma, 2015. 545 p. Disponível em: https://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf. Acesso em: 24 out. 2019.

FLANAGAN, D. *JavaScript*: o guia definitivo. 6. ed. Porto Alegre: Bookman, 2013. 1080 p.

MACHADO, R. P.; FRANCO, M. H. I.; BERTAGNOLLI, S. C. *Desenvolvimento de software III*: programação de sistemas web orientada a objetos em Java. Porto Alegre: Bookman, 2016. 220 p. (Série Tekne; Eixo Informação e Comunicação).

Leituras recomendadas

JAVASCRIPT. *MDN Web Docs*, Mountain View, 2019. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Acesso em: 24 out. 2019.

SANDERS, B. *Smashing HTML5*: técnicas para a nova geração da web. Porto Alegre: Bookman, 212. 368 p.

SIMPSON, K. *JavaScript and HTML5 now*: a new paradigma for the open web. Sebastopol: O'Reilly, 2012. 12 p.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS