# JSDG 🦏 | Metaprogramming | Метапрограммирование | chapter 14

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

---

Многие из описанных здесь средств можно условно назвать "метапрограммированием": если нормальное программирование предусматривает написание кода для манипулирования данными, то метапрограммирование предполагает написание кода для манипулирования другим кодом.

В динамическом языке вроде JavaScript границы между программированием и метапрограммированием размыты - программисты, привыкшие к более статическим языкам, могут относить к метапрограммированию даже простую возможность прохода по свойствам объекта с помощью цикла for/in.

---

Расширяемость объектов

Атрибут extensible (расшияремый) объекта указывает, можно ли добавлять к объекту новые свойства. Обычные объекты JavaScript по умолчанию расширяемы, но вы можете изменять это с помощью функций

Чтобы определить, является ли объект расширяемым, его необходимо передать функции Object.isExtensible().

Чтобы сделать объект нерасширяемым, егонужно передать функции Object.preventExtensions ().

Обратите внимание, что после того, как вы сделали объект нерасширяемым, не существует способа сделать его снова расширяемым.

Также имейте в виду, что вызов Object.preventExtensions() оказывает воздействие только на расширяемость самого

объекта.

Если к прототипу нерасширяемого объекта добавляются новые свойства, то они будут унаследованы нерасширяемым объектом.

Атрибут extensible предназначен для того, чтобы иметь возможность запирать объекты в известном состоянии и предотвращать внешнее вмешательство.

Атрибут extensible объектов часто применяется в сочетании с атрибутами configurable и writable свойств, а в JavaScript определены функции, которые облегчают установку этих атрибутов вместе.

• Функция Object. seal() работает подобно
Object .preventExtensions (), но в дополнение к тому, что делает объект нерасширяемым, она также превращает все собственные свойства объекта в неконфигурируемые. Это означает, что к объекту нельзя добавлять новые свойства, а существующие свойства не удастся удалить или конфигурировать. Однако существующие свойства, которые являются записываемыми, по-прежнему можно устанавливать. Способа распечатать запечатанный объект не предусмотрено. Выяснить, запечатан ли объект, можно посредством функции Object.isSealed().

• Функция Object.freeze () запирает объект еще сильнее. Помимо того, что она делает объект нерасширяемым и его свойства неконфигурируемыми, Object.freeze () также превращает все собственные свойства данных объекта в допускающие только чтение. (Если обьект имеет свойства с методами установки, то они не затрагиваются и по-прежнему могут вызываться операциями присваивания свойствам.) Выяснить, заморожен ли объект, можно с помощью функции Object.isFrozen ().

___

**Атрибут prototype**

Атрибут prototype  объекта указывает объект, из которого наследуются свойства.

Данный атрибут настолько важен, что мы обычно говорим просто “прототип объекта o”, а не “атрибут prototype объекта o”.

Вспомните также, что когда слово protopype представлено с применением шрифта кода, оно ссылается на обыкновенное свойство объекта, не на атрибут prototype: в главе 9 объяснялось, что свойство prototype функции конструктора указывает атрибут prototype объектов, созданных с помощью этого конструктора.

Атрибут prototype устанавливается, когда объект создается. Объекты, созданные из объектных литералов, используют в качестве своих прототипов Object.prototype. Объекты, созданные посредством new, применяют для своих прототипов значение свойства prototype функции конструктора. Объекты, созданные с помощью Object.create(), используют в качестве своих прототипов первый аргумент данной функции (который может быть null).

===

Чтобы запросить прототип любого объекта, нужно передать объект функции **Object.getPrototypeOf()**:

```
Object.getPrototypeOf({}) / => Object.prototype
Object.getPrototypeOf([]) // => Array.prototype
Object.getPrototypeOf(() =>{}) // => Function.prototype
```

В разделе 14.6 описана очень похожая функция, Reflect.getPrototypeof().

```
let p = (x: 1}; // Определить объект прототипа.
let o = Object. create (p); // Создать объект с этим прототипом
p.isPrototypeOf (o) // => true: o наследуется из P
Object.prototype.isPrototype0f (p) // => true: p наследуется из Object.prototype
Object.prototype.isPrototype0f (o) // => true: тоже
```

Обратите внимание, что isPrototype0f () выполняет функцию, похожую на операцию instanceof (см. подраздел 4.9.4).

Атрибут prototype объекта устанавливается, когда объект создается, и по обыкновению остается фиксированным.

Тем не менее, с использованием **Object.setPrototype0f ()** прототип объекта можно изменять:

```
let o = {x: 1};
let p = (y: 2};
Object.setPrototypeOf(o, p); // Установить прототип объекта o в p.
o.y // => 2: o теперь наследует свойство y
let a = [1, 2, 3];
Object.setPrototype0f (a, p); // Установить прототип массива a в p
a.join // => undefined: a больше не имеет метода join ()
```

В некоторых ранних браузерных реализациях JavaScript атрибут prototype объекта предоставлялся через свойство proto_ (содержащее два подчеркивания в начале и в конце). Указанное свойство давно устарело, но достаточно много существующего кода в веб-сети зависит от _proto и потому стандарт ECMAScript делает его обязательным для всех реализаций JavaScript, которые выполняются в веб-браузерах. (В Node оно тоже поддерживается, хотя стандарт не требует его для Node.) .В современном JavaScript свойство __proto__ допускает чтение и запись, и вы можете (но не должны) применять его как альтернативу использованию функций Object.getPrototype0f () и Object.setPrototype0f (). Однако одно интересное применение proto. связано с определением прототипа объектного литерала:

```
let p = { z: 3 }:
let o = {
      x: 1,
      y: 2,
      __proto__ : p
};

o.z // => 3: o унаследован от p
```

___

**Хорошо известные объекты Symbol**

**Symbol.iterator** — лучше всех известный пример “хорошо известных объятое **Symbol”.** Существует набор значений **Symbol,** хранящихся в виде свойств.

фабричной функции Symbol(), которые используются для того, чтобы позволить коду JavaScript управлять определенными аспектами поведения объектов и классов.
В последующих подразделах будут описаны все хорошо известные объекты Symbol и продемонстрированы способы их применения.

**Symbol.iterator и Symbol. asyncIterator**

**Symbol.iterator** и **Symbol.**asyncIterator позволяют объектам или классам делать себя итерируемыми или асинхронно итерируемыми.

**Symbol.hasInstance**

```
// Определить объект как "гип", который можно использовать
// с операцией instanceof.
let uint8 = {
      [Symbol.hasInstance] (x) 1
            return Number.isInteger (x) && x >= 0 && x <= 255;
      }
}

128 instanceof uint8 // => true
256 instanceof uint8 //=> false: слишком большой
Math.PI instanceof uint8 // => false: не целый
```

Было бы столь же легко - и яснее для читателей кода - написать функцию isUint8 () вместо того, чтобы полагаться на такое поведение Symbol.has Instance.

——

**Symbol.toStringTag**

Если вы вызовите метод toString () базового объекта JavaScript, то получите строку // "[Object Object]";

```
{}.toString() // => " [object Object]"
```

Вызвав ту же самую функцию Object.prototype.toString () как

метод экземпляра встроенных типов, вы получите более интересные результаты:

```
object.prototype.toString.cal1([]) // => " (object Array]"
object.prototype.toString.cal1(/./) // "[object RegExp]"
Object.prototype.toString.cal1(()=>{}) //  "'(object Function]"
Object.prototype.toString.cal1("") // "[object String]"
Object.prototype.toString.cal1(0) // "[object Number]"
Object.prototype. toString.call(false) // "[object Boolean]"
```

Следующая функция classof () возможно полезнее, чем операция typeof, которая не делает никаких различий между типами объектов:

```
function classof (o) {
      return Object.prototype.toString.call (o).slice (8,-1);
}

classof (null) / => "Null"
classof (undefined) / => "Undefined"
classof (1) 1/ => "Number"'
classof (10n**100n) / => "BigInt"
classof ("") / => "'String"
classof (false) // => "Boolean"
classof (Symbol ()) 11 => "Symbol"
classof ({}) // => "Object"
classof ( []) / => "Array"
classof (/./) // => "RegExp"
classof (()=>{}) / => "Function"
classof (new Map () ) / => "Map"
classof (new Set () ) [=> "Set"
classof (new Date () ) / => "Date"
```

До версии ES6 такое особое поведение метода Object .prototype.toString () было доступным только для экземпляров встроенных типов, и вызов этой функции classof () с экземпляром класса, который вы определили самостоятельно, возвращал просто "Object". Тем не менее, в ES6 функция Object.prototype.toString () ищет в своем аргументе свойство с символьным именем Symbo1. toStringTag и в случае существования такого свойства применяет в выводе его значение. Это означает, что если вы определяете собственный

класс, то можете легко заставить его работать с функциями, подобными classof ():

```
class Range {
      get [Symbol.toStringTag] () { return "Range"; }
            // остальной код класса не показан
}

let r = new Range (1, 10);
Object.prototype.toString.call (r) // => "[object Range]"
classof (r) / => "Range"
```

———

**Symbol.species**

До версии ES6 в JavaScript не предлагалось ни одного реального способа создания надежных подклассов встроенных классов вроде Array. Однако в ES6 вы можете расширить любой встроенный класс, просто используя ключевые слова class и extends. Прием демонстрировался в подразделе 9.5.2 на простом подклассе класса Array

```
// Тривиальный подкласс Array, который добавляет методы получения
// для первого и последнего элементов.
class EZArray extends Array (
      get first () I return this [0]; J
      get last () { return this [this.length-1]; }
}

let e = new EZArray (1, 2, 3) ;
let f = e.map (x => x * x) :

e.last // => 3: последний элемент массива EZArra по имени e
f.last // => 9: f - также объект EZArray со свойством last
```

В ES6 и последующих версиях конструктор Array () имеет свойство с символьным именем Symbol.species.

Обратите внимание, что этот объект Symbol применяется как имя свойства в функции конструктора. Большинство остальных

описанных здесь хорошо известных объектов Symbol используются в качестве имен методов обьекта-прототипа.

Методы вроде map () и slice (), которые создают и возвращают новые массивы, слегка подкорректированы в ES6 и последующих версиях.

Вместо того чтобы создавать обыкновенный объект Array, для создания нового массива они (в действительности) вызывают new this.constructor [Symbol.species] ()

Причина в том, что Array [Symbol.species] представляет собой свойство сметодами доступа, допускающее только чтение, функция получения которого просто возвращает this . Конструкторы подклассов наследуют эту функцию получения, т.е. конструктор каждого подкласса по умолчанию является собственным “видом”.

Самым простым вариантом, вероятно, будет явное определение собственного метода получения Symbo1. species при создании класса:

```
class EZArray extends Array 1
      static get [Symbol.species] () I return Array; )
      get first () { return this [01; }
      get last () { return this [this.length-1];
}

let e = new EZArray (1, 2,3) ;
let f = e.map (x => x - 1) ;
e.last // => 3
f.last // => undefined: f - нормальный массив без метода доступа last
```

Создание полезных подклассов Array было главным сценарием использования, который мотивировал введение Symbol.species, но это не единственное место, где применяется данный хорошо известный объект Symbol,

—

**Symbol.isConcatSpreadable**

```
let arraylike = {
      length: 1,
      0: 1,
      [Symbol.isConcatSpreadable]: true
};
[].concat (arraylike) // => [1): (результат был бы [[1]], если не распространять)
```

Для каждо строкового метода match(), matchAll(), search (), replace () и split () имеется соответствующий хорошо известный объект Symbol: Symbol.match, Symbol.search и т.д.

**[Symbol.toPrimitive]** - позволяет переопределять такое стандартное поведение преобразования объектов в элементарные значения и предоставляет вам полный контроль над тем, образом экземпляры ваших классов будут преобразовываться в элементарные значения.

Если вы хотите, чтобы экземпляры вашего класса поддерживали сравнение и сортировку с помощью < и >, тогда у вас есть веская причина определить метод
[Symbol.toPrimitive].

**Symbol.unscopables**

Вспомните, что оператор with принимает объект и выполняет свое тело, как если бы оно находилось в области видимости, где свойства этого объекта были переменными. Когда в класс Array были добавлены новые методы, возникли проблемы с совместимостью, из-за которых перестала работать часть существующего кода. Результатом стало появление объекта Symbol.unscopables. В ES6 и последующих версиях оператор with был слегка модифицирован. При использовании с объектом о оператор with вычисляет Object.keys(o[Symbol.unscopables] II {}) иигнори рует свойства, имена которых присутствуют в результирующем массиве, когда создает искусственную область видимости, где выполняется его тело. Такое решение применяется в ES6 для добавления новых методов к Array.prototype

без нарушения работы существующего кода в веб-сети.

let newArrayMethods =
Object.keys(Array.prototype[Symbol.unscopables]);

---

Теги шаблонов

Строки внутри обратных кавычек известны как "шаблонные литералы" и были раскрыты в подразделе 3.3.4. Когда за выражением, значением которого является функция, следует шаблонный литерал, выражение превращается в вызов функции, называемый "тегированным шаблонным литералом" (tagged template literal).

Первый аргумент представляет собой массив строк и за ним могут быть указаны ноль или большее количество дополнительных аргументов, имеющие значения любого типа.

Если шаблонный литерал - просто постоянная строка, не содержащая интерполяций, тогда теговая функция будет вызвана с массивом из одной строки и без дополнительных аргументов

В общем случае, если шаблонный литерал имеет n интерполированных значений, тогда теговая функция будет вызвана с n+1 аргументов. В первом аргументе передается массив из n+1 строк, а в остальных -- n интерполированных значений в порядке, в котором они следуют в шаблонном литерале.

```
function html (strings, values) {
    // Преобразовать каждое значение в строку
    // и отменить специальные символы HTML.
    let escaped = values.map (v => String (v)
            . replace ("&", "damp; ")
            •replace ("<" "&lt;")
            .replace (">" "&gt;")
            • replace ( '" , "&quot;")
            •replace ("'", "&#39; ") ) ;

    // Возвратить объединенные строки и отмененные значения.
```

```
    let result = strings [0];
    for (let i = 0; i < escaped. length; i++) {
        result += escaped [i] + strings (it1];
    }

    return result;
}

let operator = "<";

html'<b>x $(operator) y</b>' // => "<b>x &lt; y</b>"

let kind = "game", name = "DaD";

html'<div class="$ kind'">$ {name\</div>' // =>'<div
class="game">D&amp; D</div>
```

Одним из средств, мимоходом упомянутых в подразделе 3.3.4, была теговая функция String .raw", которая возвращает строку в ее "необработанной" форме, не интерпретируя любые управляющие последовательности с обратной косой чертой. Она реализована с применением особенности вызова теговых функций, которую мы еще не обсуждали. Мы видели, что когда теговая функция вызывается, в ее первом аргументе передается массив строк. Но этот массив также имеет свойство по имени raw, значением которого является еще один массив строк с тем же самым количеством элементов. Массив в аргументе содержит строки, в которых управляющие последовательности были интерпретированы как обычно, а массив гаш включает строки, где управляющие последовательности не интерпретировались. Такая малоизвестная особенность важна, если вы хотите определить DSL с грамматикой, которая использует обратные косые черты. Скажем, если нужно, чтобы теговая функция glob"* поддерживала сопоставление с образцом для путей в стиле Windows (в которых применяются обратные косые черты, а не прямые) и нежелательно заставлять пользователей дублировать каждую обратную черту, то функцию glob () можно переписать с целью использования strings.raw [] вместо strings []. Недостаток, конечно же, в том, что мы больше не сможем применять в шаблонных литералах управляющие последовательности вроде \u!

---

**API-интерфейс Reflect**

Объект Reflect не является классом; подобно объекту Math его свойства просто определяют коллекцию связанных функций. Такие функции, добавленные в ES6, определяют API-интерфейс для выполнения "рефлексии" объектов и их свойств. Новой функциональности здесь немного: объект Reflect определяет удобный набор функций, все в единственном пространстве имен, которые имитируют поведение синтаксиса базового языка и дублируют возможности разнообразных существующих функций Object.

Хотя функции Reflect не предлагают никаких новых возможностей, они группируют их вместе в одном удобном AP-интерфейсе. И, что важнее, набор функций Reflect имеет отображение один к одному с набором методов обработчиков Proxy, которые мы рассмотрим в разделе 14.7.

---

**Объекты Proxy**

Класс посредника Proxy, доступный в ES6 и последующих версиях, является наиболее мощным средством метапрограммирования в JavaScript.

Он позволяет писать код, который изменяет фундаментальное поведение объектов JavaScript.
Когда мы создаем объект Proxy, то указываем два других объекта - объект цели (target) и обьект обработчиков (handlers):

let proxy = new Proxy (target, handlers);

Однако прозрачные оболочки могут быть полезны, когда создаются как "аннулируемые посредники".

Вместо создания объекта Proxy с помощью конструктора Proxy () вы можете применить фабричную функцию Proxy. revocable (), возвращающую объект, который включает объект Proxy и также

функцию revoke (). Как только вы вызовите функцию revoke (), посредник немедленно прекращает работу:

function accessTheDatabase () { /* реализация опущена * / return 42; }

let { proxy, revoke } = Proxy. revocable (accessTheDatabase, {} );

proxy () // => 42: Посредник предоставляет доступ к внутренней функции цели.

revoke (); / / Но при желании этот доступ можно отключить в любое время.

proxy (); // !TypeError: мы больше не можем вызывать эту функцию.

Еще одна методика, используемая при написании посредников, предусматривает определение методов обработчиков, которые перехватывают операции над объектом, но по-прежнему делегируют их выполнение объекту цели. Функции API-интерфейса Reflect (см. раздел 14.6) имеют в точности те же сигнатуры, что и методы обработчиков, поэтому они облегчают такое делегирование.

```
// Мы можем автоматически сгенерировать остальные обработчики.
// Метапрограммирование ведет к победе!
Reflect.ownKeys(Reflect).forEach (handlerName => {
    if (! (handlerName in handlers)) {
        handlers [handlerName] = function (target, ...args) (
        // Заретистрировать операцию.
        console.log ('Handler $ (handlerName\ ($ (objnamel, $ (args})");
        // Делегировать выполнение операции.
        return Reflect [handlerName] (target, ...args);
    };
}):

Reflect.isExtensible ()
```

Reflect.getOwnPropertyDescriptor ()