15 августа 2022 г.

Методы примитивов

JavaScript позволяет нам работать с примитивными типами данных – строками, числами и т.д., как будто они являются объектами. У них есть и методы. Мы изучим их позже, а сначала разберём, как это всё работает, потому что, конечно, примитивы – не объекты.

Давайте взглянем на ключевые различия между примитивами и объектами.

Примитив

- Это – значение «примитивного» типа.
- Есть 7 примитивных типов: `string`, `number`, `boolean`, `symbol`, `null`, `undefined` и `bigint`.

Объект

- Может хранить множество значений как свойства.
- Объявляется при помощи фигурных скобок `{}`, например: `{name: "Рома", age: 30}`. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что мы можем хранить функцию как одно из свойств объекта.

```
let roma = {
  name: "Рома",
  sayHi: function() {
    alert("Привет, дружище!");
  }
};

roma.sayHi(); // Привет, дружище!
```

Здесь мы создали объект `roma` с методом `sayHi`.

Существует множество встроенных объектов. Например, те, которые работают с датами, ошибками, HTML-элементами и т.д. Они имеют различные свойства и методы.

Однако у этих возможностей есть обратная сторона!

Объекты «тяжелее» примитивов. Они нуждаются в дополнительных ресурсах для поддержания внутренней структуры.

Примитив как объект

Вот парадокс, с которым столкнулся создатель JavaScript:

- Есть много всего, что хотелось бы сделать с примитивами, такими как строка или число. Было бы замечательно, если бы мы могли обращаться к ним при помощи методов.
- Примитивы должны быть лёгкими и быстрыми насколько это возможно.

Выбранное решение, хотя выглядит оно немного неуклюже:

1. Примитивы остаются примитивами. Одно значение, как и хотелось.
2. Язык позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов.
3. Чтобы это работало, при таком доступе создаётся специальный «объект-обёртка», который предоставляет нужную функциональность, а после удаляется.

Каждый примитив имеет свой собственный «объект-обёртку», которые называются: `String`, `Number`, `Boolean`, `Symbol` и `BigInt`. Таким образом, они имеют разный набор методов.

К примеру, существует метод str.toUpperCase(), который возвращает строку в верхнем регистре.

Вот, как он работает:

```
let str = "Привет";

alert( str.toUpperCase() ); // ПРИВЕТ
```

Очень просто, не правда ли? Вот, что на самом деле происходит в `str.toUpperCase()`:

1. Строка `str` – примитив. В момент обращения к его свойству, создаётся специальный объект, который знает значение строки и имеет такие полезные методы, как `toUpperCase()`.
2. Этот метод запускается и возвращает новую строку (показывается в `alert`).
3. Специальный объект удаляется, оставляя только примитив `str`.

Получается, что примитивы могут предоставлять методы, и в то же время оставаться «лёгкими».

Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта. Однако, он всё же должен придерживаться спецификаций и работать так, как будто он его создаёт.

Число имеет собственный набор методов. Например, toFixed(n) округляет число до n знаков после запятой.

```
let num = 1.23456;

alert( num.toFixed(2) ); // 1.23
```

Более подробно с различными свойствами и методами мы познакомимся в главах Числа и Строки.

Конструкторы `String/Number/Boolean` предназначены только для внутреннего пользования

Некоторые языки, такие как Java, позволяют явное создание «объектов-обёрток» для примитивов при помощи такого синтаксиса как `new Number(1)` или `new Boolean(false)`.

В JavaScript, это тоже возможно по историческим причинам, но очень **не рекомендуется**. В некоторых местах последствия могут быть катастрофическими.

Например:

```
alert( typeof 0 ); // "число"

alert( typeof new Number(0) ); // "object"!
```

Объекты в `if` всегда дают `true`, так что в нижеприведённом примере будет показан `alert`:

```
let zero = new Number(0);

if (zero) {
  // zero возвращает "true", так как является объектом
  alert( "zero имеет «истинное» значение?!?" );
}
```

С другой стороны, использование функций `String/Number/Boolean` без оператора `new` – вполне разумно и полезно. Они превращают значение в соответствующий примитивный тип: в строку, в число, в булевый тип.

К примеру, следующее вполне допустимо:

```
let num = Number("123"); // превращает строку в число
```

null/undefined не имеют методов

Особенные примитивы `null` и `undefined` являются исключениями. У них нет соответствующих «объектов-обёрток», и они не имеют никаких методов. В некотором смысле, они «самые примитивные».

Попытка доступа к свойствам такого значения возвратит ошибку:

```
alert(null.test); // ошибка
```

Итого

- Все примитивы, кроме `null` и `undefined`, предоставляют множество полезных методов. Мы познакомимся с ними поближе в следующих главах.
- Формально эти методы работают с помощью временных объектов, но движки JavaScript внутренне очень хорошо оптимизируют этот процесс, так что их вызов не требует много ресурсов.

Задачи

Можно ли добавить свойство строке?

важность: 5

Взгляните на следующий код:

```
let str = "Привет";

str.test = 5;

alert(str.test);
```

Как вы думаете, это сработает? Что выведется на экран?

решение

Попробуйте запустить код:

```
let str = "Привет";

str.test = 5; // (*)

alert(str.test);
```

В зависимости от того, используете ли вы строгий режим (`use strict`) или нет, результат может быть:

1. `undefined` (без strict)
2. Ошибка (strict mode)

Почему? Давайте посмотрим что происходит в строке кода, отмеченной `(*)` :

1. В момент обращения к свойству `str` создаётся «объект-обёртка».
2. В строгом режиме, попытка изменения этого объекта выдаёт ошибку.
3. Без строгого режима, операция продолжается, объект получает свойство `test` , но после этого он удаляется, так что на последней линии `str` больше не имеет свойства `test` .

Данный пример наглядно показывает, что примитивы не являются объектами.

Они не могут хранить дополнительные данные.

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>` , для нескольких строк кода — тег `<pre>` , если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen…)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Джун на фронте
5 дней назад

👋 Привет, друзья!

Хочу поделиться с вами своим путем в IT-сферу. Ровно год назад я мечтал стать разработчиком, но не знал, как начать. И тогда я узнал об этом сайте. 🤓

И вот, я так вдохновился этим обучением, что создал **телеграм-канал "Джун на фронте"**. Где я записываю свои успехи и делюсь опытом.

Если вы тоже мечтаете о карьере в разработке или дизайне, то присоединяйтесь. Вместе мы победим. 🚀

KingNut Academy
месяц назад

Разбор задач, тестов и всего что поможет при прохождении собеседования на должность frontend developer (html, css, js, ts, react). Заходи в ТГ @interview_masters

Crash
6 месяцев назад

Кто из Казани, го дружить @sobaka35 - тг

Anna
6 месяцев назад

Очень понравилась статья. Интересно было узнать такие нюансы

Flyress
8 месяцев назад edited

Знающие люди, подскажите насчёт моментов "удаления" обьекта из памяти. Немножко оффтоп, но я считаю что это касается и создаваемых в этой статье неявных обьектов-обёрток:

```
...
someMethod() { // метод тоже по факту является обьектом
console.log("some output");
},
}

let savedMethod = anyObj.someMethod;
anyObj = null;
savedMethod();
```

В главе про сборщик мусора было сказано, что память очищается если доступ к обьекту из корня отсутствует. Следовательно, я делаю вывод, что при таком коде у нас отсутствует доступ к anyObj-

показать больше

0 0 Ответить • Поделиться ›

Flyress Flyress
8 месяцев назад edited

Теперь попробую ответить на свой вопрос...

```
let num = 123.456;
let primitiveMethod = num.toFixed;
```

После 2 строчки обьекта в памяти уже нет. Потому что его свойство .toFixed на самом деле хранило адрес на адрес на метод toFixed. И удалив его, сам метод toFixed как обьект в памяти остаётся достижим, а обьект-обёртка уже не достижим. А достижим обьект функции потому, что было присвоено значение адреса на этот обьект переменной primitiveMethod, а также собственно присвоен адрес на этот адрес.
Если всё примерно так, то спасибо за помощь, я прям преисполнился в понимании того, что происходит. Вот только как теперь воспринимать переменные и значения как раньше, по-простому, не знаю)))
P.S.
Я так понимаю в JS нет отдельного типа "адрес",чтобы абстрагироваться от этого вот всего, по крайней мере на уровне кода

0 0 Ответить • Поделиться ›

Maxim Flyress
8 месяцев назад edited

Когда значение впервые создается, под нее выделяется память и создается адрес доступа к этому значению, затем переменная получает этот адрес к значению (если происходила операция создания переменной и присвоение ей значения). Когда переменная хочет получить значение чтобы использовать где-нибудь, она обращается по этому адресу и получает это значение.

Итак вы можете объявить переменную с какой-нибудь строкой, а потом создать еще несколько переменных и указать в качестве их значения - значение первой переменной. Теперь все переменные ссылаются на одно и тоже значение.

Итого в памяти физически только одна строка, на которую ссылаются несколько переменных. Отсюда

делается вывод, пока хоть одна переменная будет иметь ссылку на эту строчку эта строка в памяти будет висеть.

P.S Почитайте что-нибудь по C, вам сразу откроется понимание как многие языки работают с памятью.

2 0 Ответить • Поделиться ›

Flyress → Maxim
8 месяцев назад

Насчёт обьектов, кажись я понял что происходит:

```
let obj1 = {
prop1: 1,
prop2: 2,
method() {
console.log("some output");
},
};
```

Переменная obj1 после присваивания обьекта в качестве значения хранит не адрес этого обьекта, а хранит адрес на адрес этого обьекта :D. То есть если сделать так:

```
let obj2 = obj1;
```

Или так:

```
function usingObj(obj) {
```

показать больше

0 0 Ответить • Поделиться ›

Maxim → Flyress
8 месяцев назад

Переменная хранит именно что адрес на этот объект, а не адрес на адрес объекта.

Вы уже привыкли, что если несколько переменных ссылаются на один объект, а потом этот объект изменяется то при изменении - вы это увидите у других переменных.

Но когда дело касается скалярных данных, то механика немного другая. Я приводил примером, что несколько переменных ссылаются на одно и то же скалярное значение. Но что если мы изменим у одной из переменных эту строку? Во первых выделится память под новые данные и будет получена ссылка на них, которая будет передана переменной. Остальные переменные ссылавшиеся на старую строчку так и останутся.

P.S Если вам интересно все что связано с адресацией и выделением памяти - настоятельно советую почитать что-то по C.

0 0 Ответить • Поделиться ›

Flyress → Maxim
8 месяцев назад edited

Если я вас правильно понял, то происходит вот как:

```
let num1 = 12345;
let num2 = 12345;
```

В данном 1 случае создаётся 2 разные ячейки памяти для 12345, а num1 и num2 сохраняют 2 разных адреса на эти ячейки. При этом вся конструкция как выражение без let возвращает значение 12345.

```
let num1 = 12345;
let num2 = num1;
```

В данном 2 случае создаётся 1 ячейка памяти для 12345 и num1 сохраняет адрес на неё. Затем переменная num2 получает значение 12345 по адресу переменной num1, но кроме значения num1, сохраняет ещё и адрес на ту же ячейку, не создавая новый адрес.
Но если после 2 случая сделать например так:
num1 = 5;
То создаётся опять таки новая ячейка в памяти под значение 5, и num1 переприсваивает адрес на неё, а старый теряется. При этом num2 всё ещё ссылается на ту самую ячейку с 12345, и поэтому она

показать больше

0 0 Ответить • Поделиться ›

Maxim Flyress

8 месяцев назад edited

С первым примером может произойти оптимизация, при анализе скрипта интерпретатор может взять 1234 и выделить память только для одного значения. Что приведет вас практически к поведению когда две переменные ссылаются на один адрес.

Вторая переменная не ссылается на первую переменную, иначе если первая переменная изменилась то вторая автоматически показывала другое значение независимо от того на какой тип ссылалась первая переменная изначально.

Объекты в js это структуры из C, а примитивы это скаляры из C.

Если вот прям вам так очень интересна тема, почитайте как C работает со своими данными. Js ведь написан на C++. Но я не рекомендую себя мучать этими вещами в js, вы не управляете памятью, вы можете только предотвратить ненужное хранение объектов в памяти, о чем и было уже написано в одной из статей на этом сайте.

1 0 Ответить • Поделиться ›

Комментарий был удален.

3 **Запрещаю писать даты** Guest

8 месяцев назад

О это та стремная баба с рыжими волосами, жрущая свой палец. Чего аватарку поменяла, думала мы тебя не помним? Виктора загнобили за даты и тебя загнобим

0 1 Ответить • Поделиться ›

3 **Запрещаю писать даты** Запрещаю писать даты

8 месяцев назад

И так будет с каждым кто напишет дату

1 0 Ответить • Поделиться ›

1 0 Ответить • Поделиться ›

Этот комментарий ожидает проверки. Показать комментарий.

ЗП

Этот комментарий ожидает проверки. Показать комментарий.

johndoe Запрещаю писать даты

8 месяцев назад

Ребята, сбоку от коммента есть менюшка. Дружно помечаем как спам и по желанию баним. Тогда останутся нормальные содержательные комменты, а мусор отправится на своё место.

2 0 Ответить • Поделиться ›

Н

Николай Кривошеев Запрещаю писать даты

9 месяцев назад

=DD

0 0 Ответить • Поделиться ›

Дима Кононов

9 месяцев назад

Зачем пишете дату?

1 0 Ответить • Поделиться ›

З

Запрещаю писать даты

9 месяцев назад

Кто напишет дату - тот 4ёрт

7 0 Ответить • Поделиться ›

Загрузить ещё комментарии