Поделиться

Редактировать на GitHub

Объект функции, NFE

Как мы уже знаем, в JavaScript функция – это значение.

Каждое значение в JavaScript имеет свой тип. А функция – это какой тип?

В JavaScript функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но и использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

Свойство «name»

Объект функции содержит несколько полезных свойств.

Например, имя функции нам доступно как свойство «name»:

```
function sayHi() {
  alert("Hi");
}

alert(sayHi.name); // sayHi
```

Что довольно забавно, логика назначения `name` весьма умная. Она присваивает корректное имя даже в случае, когда функция создаётся без имени и тут же присваивается, вот так:

```
let sayHi = function() {
  alert("Hi");
};

alert(sayHi.name); // sayHi (есть имя!)
```

Это работает даже в случае присваивания значения по умолчанию:

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (работает!)
}

f();
```

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается определить его из контекста.

Также имена имеют и методы объекта:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }

}
```

```
13  alert(user.sayHi.name); // sayHi
14  alert(user.sayBye.name); // sayBye
```


Поделиться

Редактировать на GitHub

В этом нет никакой магии. Бывает, что корректное имя определить невозможно. В таких случаях свойство name имеет пустое значение. Например:

```
1  // функция объявлена внутри массива
2  let arr = [function() {}];
3
4  alert( arr[0].name ); // <пустая строка>
5  // здесь отсутствует возможность определить имя, поэтом
```

Впрочем, на практике такое бывает редко, обычно функции имеют `name`.

Свойство «length»

Ещё одно встроенное свойство «length» содержит количество параметров функции в её объявлении. Например:

```
1  function f1(a) {}
2  function f2(a, b) {}
3  function many(a, b, ...more) {}
4
5  alert(f1.length); // 1
6  alert(f2.length); // 2
7  alert(many.length); // 2
```

Как мы видим, троеточие, обозначающее «остаточные параметры», здесь как бы «не считается»

Свойство `length` иногда используется для интроспекций в функциях, которые работают с другими функциями.

Например, в коде ниже функция `ask` принимает в качестве параметров вопрос `question` и произвольное количество функций-обработчиков ответа `handler`.

Когда пользователь отвечает на вопрос, функция вызывает обработчики. Мы можем передать два типа обработчиков:

- Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
- Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Чтобы вызвать обработчик `handler` правильно, будем проверять свойство `handler.length`.

Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

```
1  function ask(question, ...handlers) {
2    let isYes = confirm(question);
3
4    for(let handler of handlers) {
5      if (handler.length == 0) {
6        if (isYes) handler();
7      } else {
8        handler(isYes);
9      }
10   }
11
12 }
13
14 // для положительных ответов вызываются оба типа обрабо
15
```

```
// для отрицательных - только второго типа
ask("Вопрос?", () => alert('Вы ответили да'), result =>
```


Это частный случай так называемого Ad-hoc-полиморфизма – обработка аргументов в зависимости от их типа или, как в нашем случае – от значения `length`. Эта идея имеет применение в библиотеках JavaScript.

Пользовательские свойства

Мы также можем добавить свои собственные свойства.

Давайте добавим свойство `counter` для отслеживания общего количества вызовов:

```
function sayHi() {
  alert("Hi");

  // давайте посчитаем, сколько вызовов мы сделали
  sayHi.counter++;
}
sayHi.counter = 0; // начальное значение

sayHi(); // Hi
sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2
```

> **Свойство не есть переменная**
>
> Свойство функции, назначенное как `sayHi.counter = 0`, не *объявляет* локальную переменную `counter` внутри неё. Другими словами, свойство `counter` и переменная `let counter` – это две независимые вещи.
>
> Мы можем использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот. Это два параллельных мира.

Иногда свойства функции могут использоваться вместо замыканий. Например, мы можем переписать функцию-счётчик из главы Замыкание, используя её свойство:

```
function makeCounter() {
  // вместо
  // let count = 0

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

Свойство `count` теперь хранится прямо в функции, а не в её внешнем лексическом окружении.

Это хуже или лучше, чем использовать замыкание?

Основное отличие в том, что если значение `count` живёт во внешней переменной, то оно не доступно для внешнего кода. Изменить его могут

только вложенные функции. А если оно присвоено как свойство функции, то мы можем его получить:

```
function makeCounter() {

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

Поэтому выбор реализации зависит от наших целей.

Named Function Expression

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя.

Например, давайте объявим Function Expression:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

И присвоим ему имя:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

Чего мы здесь достигли? Какова цель этого дополнительного имени `func`?

Для начала заметим, что функция всё ещё задана как Function Expression. Добавление `"func"` после `function` не превращает объявление в Function Declaration, потому что оно все ещё является частью выражения присваивания.

Добавление такого имени ничего не ломает.

Функция все ещё доступна как `sayHi()`:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

sayHi("John"); // Hello, John
```

Есть две важные особенности имени `func`, ради которого оно даётся:

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

Например, ниже функция `sayHi` вызывает себя с `"Guest"`, если не передан параметр `who`:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
```

```
  } else {
    func("Guest"); // использует func, чтобы снова вызв
  }
};

sayHi(); // Hello, Guest

// А вот так - не сработает:
func(); // Ошибка, func не определена (недоступна вне ф
```

Почему мы используем `func`? Почему просто не использовать `sayHi` для вложенного вызова?

Вообще, обычно мы можем так поступить:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

Однако, у этого кода есть проблема, которая заключается в том, что значение `sayHi` может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнёт выдавать ошибки:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Ошибка: sayHi не является функци
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Ошибка, вложенный вызов sayHi больше не р
```

Так происходит, потому что функция берёт `sayHi` из внешнего лексического окружения. Так как локальная переменная `sayHi` отсутствует, используется внешняя. И на момент вызова эта внешняя `sayHi` равна `null`.

Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы.

Давайте используем его, чтобы исправить наш код:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // Теперь всё в порядке
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (вложенный вызов работает)
```

Теперь всё работает, потому что имя `"func"` локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию.

Поделиться

Редактировать на GitHub

Внешний код все ещё содержит переменные `sayHi` и `welcome`, но теперь `func` – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Это не работает с Function Declaration

Трюк с «внутренним» именем, описанный выше, работает только для Function Expression и *не* работает для Function Declaration. Для Function Declaration синтаксис не предусматривает возможность объявить дополнительное «внутреннее» имя.

Зачастую, когда нам нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

Итого

Функции – это объекты.

Их свойства:

- `name` – имя функции. Обычно берётся из объявления функции, но если там нет – JavaScript пытается понять его из контекста.
- `length` – количество аргументов в объявлении функции. Троеточие («остаточные параметры») не считается.

Если функция объявлена как Function Expression (вне основного потока кода) и имеет имя, тогда это называется Named Function Expression (Именованным Функциональным Выражением). Это имя может быть использовано для ссылки на себя же, для рекурсивных вызовов и т.п.

Также функции могут содержать дополнительные свойства. Многие известные JavaScript-библиотеки искусно используют эту возможность.

Они создают «основную» функцию и добавляют множество «вспомогательных» функций внутрь первой. Например, библиотека jQuery создаёт функцию с именем `$`. Библиотека lodash создаёт функцию `_`, а потом добавляет в неё `_.clone`, `_.keyBy` и другие свойства (чтобы узнать о ней побольше см. документацию). Они делают это, чтобы уменьшить засорение глобального пространства имён посредством того, что одна библиотека предоставляет только одну глобальную переменную, уменьшая вероятность конфликта имён.

Таким образом, функция может не только делать что-то сама по себе, но также и предоставлять полезную функциональность через свои свойства.

Задачи

Установка и уменьшение значения счётчика

важность: 5

Измените код `makeCounter()` так, чтобы счётчик мог увеличивать и устанавливать значение:

- `counter()` должен возвращать следующее значение (как и раньше).
- `counter.set(value)` должен устанавливать счётчику значение `value`.
- `counter.decrease()` должен уменьшать значение счётчика на 1.

Посмотрите код из песочницы с полным примером использования.

P.S. Для того, чтобы сохранить текущее значение счётчика, можно воспользоваться как замыканием, так и свойством функции. Или сделать два варианта решения: и так, и так.

Открыть песочницу с тестами для задачи.

решение

Сумма с произвольным количеством скобок

Поделиться

Редактировать на GitHub

важность: 2

Напишите функцию `sum`, которая бы работала следующим образом:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.S. Подсказка: возможно вам стоит сделать особый метод преобразования в примитив для функции.

решение

Проводим курсы по JavaScript и фреймворкам.

Комментарии

перед тем как писать...

© 2007—2020 Илья Кантор | о проекте | связаться с нами | пользовательское соглашение | политика конфи