Глава 3

Типы данных (часть 2)

3.1. Методы объекта

3.1.1. Методы объекта

У объекта можно определить произвольное свойство и установить в качестве значения этого свойства функцию. Такое свойство будет называться методом объекта.

Внутри этой функции будет доступно ключевое слово this, о котором подробнее речь пойдет позже в данном курсе. Ключевое слово this внутри метода объекта ссылается на сам этот объект, а значит с его помощью можно обращаться к вложенным свойствам этого объекта.

```
// Объект с предопределенным набором свойств
var tweet = {
  …
  likes: 16,
  getLikes: function() {
    return this.likes;
  },
  setLikes: function(value) {
    this.likes = parseInt(value) || 0;
    return this;
  },
  getAuthor: function() {
    return this.user.screenName;
  }
};
```

3.1.2. Цепочка вызовов методов

Метод, в том числе, может возвращать значение этого ключевого слова. Это часто используется на практике в методах, меняющих внутреннее состояние объекта. В примере выше таким методом является метод setLikes (задает количество лайков), а метод getLikes позволяет получить количество лайков:

```
tweet.getLikes(); // 16
```

Если функция, меняющая состояние объекта, после своего вызова возвращает сам объект, появляется возможность выстраивать цепочку вызовов других методов этого объекта:

```
tweet.setLikes(17) // { … }
     .getLikes();  // 17
```

В данном простом примере таким образом получается проверить, что внутренние состояние объекта действительно изменилось.

3.1.3. Геттеры/сеттеры свойств объекта

Сущствует альтернативный способ создания методов объектов с помощью так называемых геттеров/сеттеров. Для некоторого свойства могут быть определены:

- функция-геттер вызывается каждый раз при попытке прочитать свойство. Возвращаемое в ходе исполнения этой функции значение как раз и будет считанным значением свойства.
- функция-сеттер вызывается каждый раз при попытке изменить значение свойства, получает в качестве параметра устанавливаемое значение и может изменить внутренние переменные объекта.

Геттер определяется как функция, не принимающая параметров, определяемая с помощью ключевого слова get, имя которой является именем свойства, для которого определяется геттер. Сеттер же определяется с помощью ключевого слова set и принимает один параметр. Внутри сеттеров и геттеров свойств объекта можно использовать ключевое слово this для доступа к внутренним переменным.

В следующем примере определяются геттер и сеттер для свойства likes, которые соответственно возвращают и устанавливают значение внутренней переменной `_likes`:

```
// Объект с предопределенным набором свойств
var tweet = {
```

```
  …
  _likes: 16,
  get likes() {
    return this._likes;
  },
  set likes(value) {
    this._likes = parseInt(value) || 0;
  },
  getAuthor: function() {
    return this.user.screenName;
  }
};
```

При обращении к свойству объекта, для которого определен геттер, вызывается соответствующая функция, результат выполнения которой и будет считанным значением:

```
tweet.likes; // 16
```

При этом обращение к свойству объекта происходит аналогично тому, как происходило обращение к статичному свойству.

При попытке присвоить значение свойству, для которого определен сеттер, вызывается функция-обработчик, первым аргументом которой передается значение справа от оператора присваивания и которая уже устанавливает внутренние свойства объекта.

```
tweet.likes = 17;
tweet.likes; // 17
```

3.2. Обработка исключений

3.2.1. Основные подходы к обработке ошибок

Есть несколько шаблонов для работы с ошибками:

- Явная обработка нештатных ситуаций и граничных условий.
- Работать по так называемому контракту.

3.2.2. Обработка ошибок по контракту. Исключения

При работе по контракту устанавливаются ограничения на принимаемые функцией параметры. Если они не выполнены, функция «кидает исключение» какого-то типа. В свою очередь, код, который использует функцию, должен будет обработать эту исключительную ситуацию.

Можно переписать следующим образом пример выше:

```
// Объект с предопределенным набором свойств
var tweet = {
  …
  _likes: 16,
  get likes() {
    return this._likes;
  },
  set likes(value) {
    var likes = parseInt(value);

    if (isNaN(likes) || likes < 0) {
        throw new TypeError('Передано неверное значение');
    }

    this._likes = likes;
  },
  getAuthor: function() {
    return this.user.screenName;
  }
};
```

Здесь внутри сеттера явно производится проверка, что устанавливаемое значение является неотрицательным числом. В случае, если это не так, срабатывает исключение.

3.2.3. Обработка ошибок по контракту. Оператор try-catch.

Код, который вызвал функцию, может обрабатывать возможные исключительные ситуации с помощью оператора try-catch:

```
try {
  tweet.likes = 'foo';
} catch (e) {
  if (e instanceof TypeError) {
    tweet.likes = 0;
  }
  console.error(e);
}

tweet.likes; // 0
```

Оператор try-catch состоит из двух частей:

- Блок try, в котором расположен код, который может бросить исключения.
- Блок catch, который принимает на вход исключение, если оно будет выброшено, и позволяет его обработать.

Внутри блока catch удобно использовать оператор instanceof для проверки типа исключения, которое было выброшено. Таким образом, появляется возможность отделить известные ошибки, которые были задокументированы разработчиком, от остальных.

3.2.4. Объект исключения. TypeError

Объект исключения содержит несколько полезных свойств:

- Свойство name, которое содержит имя типа ошибки:

```
e.name; // 'TypeError'
```

- Свойство message, которое содержит сообщение об ошибке:

```
e.message; // 'Передано неверное значение'
```

- Свойство stack, которое содержит стек вызовов функций, которые привели к ошибке:

```
e.stack;
// TypeError: Передано неверное значение
//     at Object.set likes [as likes]
 ↪  (<anonymous>:10:15)
//     at <anonymous>:18:15
```

  В stack также содержится имя и сообщение об ошибке.

3.3. Сравнение и приведение объектов

JavaScript — язык с динамической типизацией. Иногда возникают ситуации, когда функция возвращает значение произвольного типа, причем тип возвращаемого значения заранее не известен.

В этом разделе речь пойдет о том, как можно адаптировать различные типы данных так, чтобы их можно было друг с другом сравнивать.

3.3.1. Сравнение объектов

Пусть есть объект panda, который обладает методами valueOf и toString. Метод valueOf возвращает код unicode-символа с пандой, а метод toString возвращает HTML-entity с кодом для вставки панды в HTML страницу.

```
var panda = {
  valueOf: function() {
    return 128060;
  },
  toString: function() {
    return '\\&\#x1F43C;';
  }
}
```

Также есть код «поросенка» в unicode-таблице, заданный примитивом (целым числом):

```
var pigCode = 128055;
```

Ставится вопрос, каков будет результат сравнения объекта с пандой и числа:

```
panda == pigCode; // ???
```

Сравнение будет происходить следующим образом:

- К левому операнду применяется внутренний метод isPrimitive:

  ```
  isPrimitive(panda); // false
  ```

  Метод isPrimitive проверяет, является ли операнд примитивом. В данном случае panda — объект и не является примитивом.

- К правому операнду применяется внутренний метод isPrimitive:

  ```
  isPrimitive(pigCode); // true
  ```

  В данном случае pigCode является примитивом (числом).

- Проверяется, имеет ли операнд, который не является примитивом, метод valueOf:

  ```
  typeof panda.valueOf === 'function'; // true
  ```

  В данном случае этот метод действительно есть.

- После этого у не-примитива вызывается метод valueOf и результат сравнивается со значением примитива:

```
panda.valueOf() === pigCode; // false
```

По сути будут сравниваться два целых числа:

```
128060 === 128055; // false
```

Таким образом, результат сравнения объекта panda и кода pigCode:

```
panda == pigCode; // false
```

Полный алгоритм можно найти по ссылке: Абстрактный Алгоритм Эквивалентного Сравнения.

Важно понимать, что при сравнении двух переменных сложных типов (два объекта или два массива), которые передаются по ссылке, реально сравниваются всегда ссылки на области памяти. Таким образом, операция сравнения двух сложных типов вернет истину только в том случае, если внутренние ссылки обоих объектов ссылаются на один и тот же объект в памяти.

В частности, два идентичных (например, пустых), но разных объекта при сравнении друг с другом будут давать false:

```
{} == {}; // false
```

Если же сравнить объект panda с кодом панды:

```
var pandaCode = 128060
```

То получится, как в этом не сложно убедиться, true:

```
panda == pandaCode; // true
```

3.3.2. Приведение объекта к числу

```
var panda = {
  valueOf: function() { return 128060; },
  toString: function() { return '&#x1F43C;'; }
}
```

Способы приведения переменной к числу:

- Передача этой переменной внутрь конструктора Number:

  ```
  Number(panda); // 128060
  ```

  Это самый простой способ преобразования переменной к числу. В результате будет вызван метод valueOf объекта и получено целое число.

- Использование унарного сложения. Если поставить знак плюс перед переменной, будет вызвана принудительная конвертация переменной в целое число, а именно вызывается метод valueOf:

```
+panda; // 128060
```

  Не рекомендуется пользоваться таким методом на практике, так как такая конвертация очень неявна и очень легко спутать операцию конкатенацию или сложение нескольких переменных с операцией унарного сложения для преобразования типов данных.

- Двойной бинарный сдвиг — еще один способ преобразования переменной к числу:

```
~~panda; // 128060
```

- Операция сравнения с целым числом. При этом вызыватеся метод valueOf.

```
panda == 128060; // true
```

  Следует отметить, что при операции строгого сравнения (===), в отличии от нестрогого сравнения (==), переменные сравниваются всегда только в рамках одного типа данных.

```
panda === 128060; // false
```

- Функция parseInt/parseFloat пытается привести объект сначала к строке, а потом распарсить ее к целому числу/числу с плавающей точкой.

```
parseInt(panda); // NaN
parseFloat(panda); // NaN
```

  В обоих случаях получается результат Not A Number. Это связано с тем, что сначала происходит преобразование к строке, а в данном случае нет строчного представления, которое могло бы быть преобразовано к строке.

- Явный вызов метода valueOf:

```
panda.valueOf(); // 128060
```

3.3.3. Приведение объекта к строке

Способы приведения объекта к строке:

- Передача этой переменной внутрь конструктора String:

  ```
  String(panda); // '&#x1F43C;'
  ```

  В результате будет вызван метод toString объекта.

- Конкатенация переменной со строкой:

  ```
  '' + panda; // '128060'
  ```

  В результате вызывается метод valueOf, но преобразуется в строку.

- Преобразование к строке также происходит при нестрогом сравнении переменной со строкой.

  ```
  panda == '128060'; // true
  ```

  При этом также вызывается метод valueOf и его значение преобразуется к строке. Строгое сравнение объекта со строкой всегда вернет false:

  ```
  panda === '128060'; // false
  ```

- Явный вызов метода toString:

  ```
  panda.toString(); // '&#x1F43C;'
  ```

Следует отметить, что только два из представленных метода сконвертировали объект в строку согласно желаемой логике (используя toString). Чтобы избежать путаницы, рекомендуется не определять методы valueOf и toString одновременно.

Например, пусть дан объект (без метода valueOf):

```
var panda = {
  toString: function() { return '&#x1F43C;'; }
}
```

При использовании описанных выше способов приведения к строке:

- Передача этой переменной внутрь конструктора String:

  ```
  String(panda); // '&#x1F43C;'
  ```

- Конкатенация переменной со строкой:

```
'' + panda; // '&#x1F43C;'
```

- Преобразование при нестрогом сравнении со строкой.

```
panda == '&#x1F43C;'; // true
```

- Преобразование при строгом сравнении со строкой.

```
panda === '&#x1F43C;'; // false
```

- Явный вызов метода toString:

```
panda.toString(); // '&#x1F43C;'
```

Как видно из данного примера, по возможности следует определять только метод toString, так как его поведение более предсказуемо.

3.4. Скрытые методы

3.4.1. Понятие скрытого метода

В данном разделе будет рассмотрен еще один способ создания свойств у объектов, который позволяет более глубоко настраивать их поведение.

Объект «панда» из предыдущего примера:

```
var panda = {
  valueOf: function() { return 128060; },
  toString: function() { return '&#x1F43C;'; }
}
```

Если применить метод Object.keys к объекту «панда», будет выведен массив из имен ключей созданного объекта.

```
Object.keys(panda); // ['valueOf', 'toString']
```

Этот массив состоит из двух строк, как и ожидалось.

Если же создать пустой массив и точно также передать его в метод Object.keys, результатом будет пустой массив:

```
var emptyObject = {};
Object.keys(emptyObject); // []
```

Это вполне предсказуемо, ведь объект был создан пустой.

С помощью оператора typeof можно определить тип свойства panda.valueOf, чтобы лишний раз убедиться, что это метод и тип значение данного свойства — функция:

```
typeof panda.valueOf === 'function'; // true
```

Если же сделать такую же проверку для пустого объекта:

```
typeof emptyObject.valueOf === 'function'; // true
```

то окажется, что метод valueOf определен на пустом объекте, не видим в списке ключей объекта, но может быть вызван.

3.4.2. Метод Object.defineProperties

Метод Object.defineProperties позволяет объявлять свойства объекта и принимает три аргумента:

1. Объект, в котором нужно определить новое свойство.
2. Название свойства, которое будет создано.
3. Объект конфигурирования, который может содержать следующие ключи:

   value Значение, которое будет установлено в качестве свойства объекта.

   атрибут writable определяет, является ли свойство записываемым.

   атрибут enumerable определяет, видно ли свойство в списке ключей объекта, получаемом при помощи Object.keys.

   атрибут configurable определяет,

   Значения параметров writable, enumerable и configurable по умолчанию — false.

Одним из ключей объекта конфигурирования является value, по которому содержится значение, которое будет установлено в качестве свойства объекта.

3.4.3. Атрибут enumerable

В рассматриваемом примере с объектом «panda» создаются два метода, а значит в качестве value должна быть указана функция. Чтобы скрыть свойство из списка ключей объекта нужно указать значение атрибута enumerable равным false.

```
var panda = {};

Object.defineProperty(panda, 'valueOf', {
```

```
  value: function() {
    return 128060;
  },
  writable: true,
  enumerable: false,
  configurable: true
});

Object.defineProperty(panda, 'toString', {
  value: function() {
    return '&#x1F43C;';
  },
  writable: true,
  enumerable: false,
  configurable: true
});
```

3.4.4. Атрибут writable

Пусть дан пустой объект, в котором предполагается хранить твит:

```
var tweet = {};
```

Новое свойство text этого объекта можно определить при помощи метода defineProperty с значением writable равным false:

```
Object.defineProperty(tweet, 'text', {
    value: 'Я и IoT, пятый доклад на #wstdays в Питере',
    writable: false
})
```

Убедиться, что свойство создано можно при помощи метода getOwnPropertyDescrip

```
Object.getOwnPropertyDescriptor(tweet, 'text');
// { value: 'Я и IoT, пятый доклад на #wstdays в Питере',
//   writable: false,
//   enumerable: false,
//   configurable: false }
```

В результате свойство может быть прочитано, но не может быть изменено:

```
tweet.text; // 'Я и IoT, пятый доклад на #wstdays в Питере'
tweet.text = 'Вёрстка писем. Развенчиваем мифы. …
 ↪ #wstdays';
tweet.text; // 'Я и IoT, пятый доклад на #wstdays в Питере'
```

При этом (не в строгом режиме) такой код ошибку не вызывает, но и не меняет значение свойства также.

3.4.5. Атрибут configurable

```
var tweet = {};
Теперь можно определить свойство со значением configurable
 ↪ равным false:
Object.defineProperty(tweet, 'text', {
    value: 'Я и IoT, пятый доклад на #wstdays в Питере',
    configurable: false
});
При помощи метода getOwnPropertyDescriptor можно убедиться,
 ↪ что свойство создано:
Object.getOwnPropertyDescriptor(tweet, 'text');
// { value: 'Я и IoT, пятый доклад на #wstdays в Питере',
//   writable: false,
//   enumerable: false,
//   configurable: false }

tweet.text; // 'Я и IoT, пятый доклад на #wstdays в
 ↪ Питере'
```

Можно проверить, что свойство существует с помощью hasOwnProperty:

```
tweet.hasOwnProperty('text'); // true
```

Однако удалить свойство не получается:

```
delete tweet.text; // false
```

Оператор delete всегда возвращает результат, успешно ли произведено удаление. Удаление является частным случаем конфигурирования свойства, поэтому свойство не получилось удалить.

Более того, можно непосредственно проверить, что свойство все еще есть:

```
tweet.text; // 'Я и IoT, пятый доклад на #wstdays в
 ↪ Питере'

tweet.hasOwnProperty('text'); // true
```

3.4.6. Заморозка объекта

Пусть из внешнего API был получен объект:

```
var tweet = {
  …
  likes: 16,
  getLikes: function() {
    return this.likes;
  }
};
```

Этот объект может быть изменен в ходе исполнения кода. На практике же часто требуется обеспечить его неизменяемость. Именно для этой цели и служит заморозка.

Проверить, является ли объект замороженным, можно с помощью метода Object.isFrozen:

```
Object.isFrozen(tweet); // false
```

В данном случае объект не заморожен и может быть изменен.

В частности, если посмотреть дескриптор любого из его свойств:

```
Object.getOwnPropertyDescriptor(tweet, 'likes')
// { value: 16,
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

окажется, что оно может быть изменено (writable равно true) и удалено (configurable равно true).

Теперь можно применить заморозку при помощи Object.freeze:

```
Object.freeze(tweet);
```

Теперь объект является замороженным:

```
Object.isFrozen(tweet); // true
Object.getOwnPropertyDescriptor(tweet, 'likes')
// { value: 16,
//   writable: false,
//   enumerable: true,
//   configurable: false }
```

А его свойства стали неизменяемыми и неконфигурируемыми. Заморозка никак не повлияла на видимость свойств (не меняет атрибут enumerable).

В результате свойство likes не получается ни удалить, ни изменить:

```
tweet.likes = 17;
tweet.likes; // 16

delete tweet.likes; // false
```

3.5. Объект Даты

Часто на практике приходится работать с датами. В JavaScript существует специальный объект Date для работы с датами.

Создание объекта-даты с помощью конструктора Date:

- Вызов конструктора Date без параметров создает объект с текущей датой в системном часовом поясе:

```
new Date(); // Mon Oct 17 2016 09:37:20 GMT+0500
↪  (YEKT)
```

- Создание даты по строке с датой:

```
tweet.createdAt; // 'Sat Oct 01 12:01:08 +0000 2016'
// Пытаемся сконвертировать строку в дату
new Date(tweet.createdAt); // Sat Oct 01 2016
↪  17:01:08 GMT+0500 (YEKT)
```

- Создание даты из UNIX Timestamp:

```
new Date(1475323268000); // Sat Oct 01 2016 17:01:08
↪  GMT+0500 (YEKT)
```

- Создание даты из набора параметров:

```
new Date(2016, 9, 1, 17, 1, 8); // Sat Oct 01 2016
↪  17:01:08 GMT+0500 (YEKT)
```

  Количество параметров должно быть больше или равно двум и они должны быть целыми числами.

Объект Date в JavaScript был импортирован из языка Java образца 1995 года, когда JavaScript был создан, и с тех пор практически не менялся. Поэтому он содержит много странностей, которые следует просто запомнить. Например, второй параметр при создании даты из набора параметров, месяц, начинается с нуля, а третий (день) — с единицы.

Чтобы получить значение UNIX Timestamp из даты можно воспользоваться методом valueOf:

```
(new Date(2016, 9, 1, 17, 1, 8)).valueOf(); //
↪  1475323268000
```

Текущее значение UNIX Timestamp можно получить с помощью статического метода Date.now():

```
Date.now(); // 1476680054602
```

Полный список методов доступен по ссылке: Date

3.6. Библиотека математических функций и констант Math

Библиотека Math представляет собой множество математических функций и констант. В частности в Math определены следующие функции:

Math.random Генерирует случайное число от 0 до 1

```
Math.random(); // 0.4468546273336771
```

Math.min Принимает на вход неограниченное количество чисел и определяет меньшее из них:

```
Math.min(1, 5); // 1
```

Math.max Определяет большее из чисел:

```
Math.max(1, 5, 10); // 10
```

Math.round Округляет число до ближайшего целого:

```
Math.round(2.7); // 3

Math.round(2.3); // 2
```

Math.floor Округляет число до целого в меньшую сторону

```
Math.floor(2.7); // 2

Math.floor(2.3); // 2
```

Math.ceil Округляет число до целого в большую сторону

```
Math.ceil(2.7); // 3

Math.ceil(2.3); // 3
```

Math.log Возвращает натуральный (по основанию e) логарифм числа

```
Math.log(10); // 2.302585092994046
```

Math.pow Возвращает основание, возведённое в степень

```
Math.pow(2, 5); // 32
```

Math.sin Возвращает синус угла в радианах

```
Math.sin(1); // 0.8414709848078965
```

Math.tan Возвращает тангенс угла в радианах

```
Math.tan(1); // 1.5574077246549023
```

Полный список доступных математических функций и констант можно посмотреть по ссылке: Math.

3.7. Регулярные выражения

Регулярные выражения в JavaScript имеют стандартный PCRE-синтаксис. Подробнее прочитать про PCRE (Perl Compatible Regular Expressions) можно по ссылке. Также полезно прочесть руководство по регулярным выражениям применительно именно к JavaScript.

Рекомендуется прибегать к использованию регулярных выражений только в крайних случаях, поскольку написание регулярных выражений представляет собой достаточно трудную задачу, а также сами регулярные выражения очень тяжело читать, если не иметь в этом опыта.

С помощью регулярного выражения можно выделить все хеш-теги в твитах:

```
tweet.text; // 'Node.js, и модули, Джеймс о проблемах
 ↪  Node.js #nodejs #modules'
```

Хеш-теги могут быть как на русском языке, так и на английском. В любом случае хеш-тег начинается с символа #. Регулярное выражение будет иметь вид:

```
/#[a-z0-9]+/
```

Оно начинается с символа решетки, за которым следует более чем один символ, который может быть от a до z или числом.

Проверить, содержится ли указанное регулярное выражение в строке с помощью метода test регулярного выражения:

```
/#[a-z0-9]+/gi.test(tweet.text); // true
```

Флаг g обозначает глобальное сопоставление, а i — игнорирование регистра при сопоставлении.

В случае, если в твите нет хеш-тегов:

```
var tweetWithoutHashtag; // 'Я и IoT, пятый доклад на WSD
 ↪  в Питере'
```

Будет иметь место следующий результат:

```
/#[a-z0-9]+/gi.test(tweetWithoutHashtag); // false
```

Пусть дан объект со свойством text, которое все также содержит текст твита:

```
var tweet = {
  text: 'Node.js, и модули, Джеймс о проблемах Node.js
   ↪  #nodejs #modules #модули'
};
```

Требуется написать метод, который бы на лету оборачивал бы все хеш-теги твита в ссылки.

Для этого можно воспользоваться методом Object.defineProperty, чтобы определить геттер для свойства linkify:

```
Object.defineProperty(tweet, 'linkify', {
  get: function() {
    return this.text.replace(/#[a-z0-9]+/gi, '<a
     ↪  href="$1">$1</a>');
  }
});
```

Функция обработчик заменяет все вхождения регулярного выражения с помощью метода text.replace. Первым аргументом метода replace является регулярное выражение, а вторым — шаблон, согласно которому должна производиться замена.

Создание геттеров при помощи defineProperty более явно, нежели, чем с помощью ключевых слов get и set, а также может быть более тонко сконфигурировано.

При помощи метода Object.getOwnPropertyDescriptor можно получить текущие параметры свойства linkify:

```
Object.getOwnPropertyDescriptor(tweet, 'linkify');
// { get: [Function: get],
//   set: undefined,
//   enumerable: false,
//   configurable: false }
```

Геттер определен, но не сеттер — поэтому свойство linkify неизменяемо.

При попытке вызвать метод linkify можно заметить, что русские теги не обрабатываются:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="$1">$1</a> <a href="$1">$1</a> #модули'
```

Чтобы исправить это, можно дополнить регулярное выражение:

```
return this.text.replace(
  /#[a-z0-9а-я]+/gi,
  '<a href="$1">$1</a>'
);
```

Теперь все хеш-теги обрабатываются, но вместо ссылок на места хеш-тегов попадают ссылки с placeholder'ами $1 и так далее:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="$1">$1</a> <a href="$1">$1</a> <a
 ↪  href="$1">$1</a>'
```

Для того, чтобы решить эту проблему, регулярное выражение следует обернуть в круглые скобки.

```
return this.text.replace(
  /(#[a-z0-9а-я]+)/gi,
  '<a href="$1">$1</a>'
);
```

Таким образом определяется захватывающая группа, которая захватывает все символы, которые попали в скобки. После этого $1 в шаблоне заменяется на содержимое первой захватывающей группы:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="#nodejs">#nodejs</a> <a
 ↪  href="#modules">#modules</a> <a
 ↪  href="#модули">#модули</a>'
```

Если определены две захватывающие группы, вторая захватывающая группа доступна в шаблоне по $2:

```
return this.text.replace(
  /(#([a-z0-9а-я]+))/gi,
  '<a href="$2">$1</a>'
);
```

Получается:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="nodejs">#nodejs</a> <a
 ↪  href="modules">#modules</a> <a
 ↪  href="модули">#модули</a>'
```

Во многих справочниках по регулярным выражениям можно увидеть применение классификатора `\w`, который соотвествует любому цифробуквенному символу, включая нижнее подчеркивание. Эквивалентен `[A-Za-z0-9_]`. Судя по всему, буквами считаются только символы латинского алфавита.

```
return this.text.replace(
  /(#([\w]+))/gi,
  '<a href="$2">$1</a>'
);

tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="nodejs">#nodejs</a> <a
 ↪  href="modules">#modules</a> #модули'
```