Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.
Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу http://learn.javascript.ru.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru .

Глава: Создание графических компонентов

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

Введение

Здесь мы рассмотрим создание интерфейсных компонент. Более коротко их называют «виджетами» (widget).

Предполагается, что к этому разделу вы уже знакомы с основами JavaScript, DOM, CSS.

В этом разделе также понадобится библиотека jQuery. Мы используем её для оптимизации основных операций с DOM/CSS. Если вы освоили учебник до этого момента, то уже знаете, как работать с ними средствами обычного JavaScript. Поэтому вас не постигнет участь горе-разработчиков, знающих «только jQuery» и впадающих в ступор при необходимости минимального выхода за возможности этой библиотеки.

Поизучать jQuery можно, например, по книге jQuery. Подробное руководство по продвинутому JavaScript [1] (Бер Бибо, Иегуда Кац). В интернет есть и много других источников на эту тему.

jQuery чеклист

Мы будем использовать следующие возможности jQuery:

- Создание элементов: простая форма `$("<div/>")` и расширенная `$("<div/>", {html: ...})`.
- Поиск элементов: `$(".selector", context)` и `elem.find(".selector")`.
- Навигация по DOM: `closest`, `parent`...
- Размеры и метрики: `offset`, `width/height`, `outerWidth/outerHeight`, `scrollTop/scrollLeft`.

- События: обработчики и делегирование с jQuery.on [2], неймспейсы в событиях.
- Манипуляции с DOM/CSS: `append/prepend/и т.п.`, метод `css`.
- Для задач на анимацию: `animate` (таких задач мало, так что не обязательно).

Вёрстка графических компонентов

При создании графических компонент («виджетов») в первую очередь придумывается их HTML/CSS-структура.

Как будет выглядеть виджет в обычном состоянии? Как будет меняться в процессе взаимодействия с посетителем?

Чтобы разработка виджета была удобной, при вёрстке полезно соблюдать несколько простых, но очень важных соглашений.

Семантическая вёрстка

HTML-разметка и названия CSS-классов должны отражать смысл информации, а не ее оформление.

Например, сообщение об ошибке можно сверстать так:

```
<div style="color:red; border: 1px solid red">
  Плохая вёрстка сообщения об ошибке: атрибут style!
</div>
```

..Или так:

```
<div class="red red-border">
  Плохая вёрстка сообщения об ошибке: несемантический class!
</div>
```

В обоих случаях семантической вёрстки не видно. В первом случае — стиль, а во втором — класс содержат информацию об *оформлении*.

При семантической вёрстке класс отвечает на вопрос: «что это?», «в каком это состоянии?» (открыто, закрыто, отключено…) и т.п.

Например:

```
<div class="error">
  Сообщение об ошибке (error), правильная вёрстка!
</div>
```

У предупреждения будет класс `message` и так далее, по смыслу.

```
<div class="warning">
  Предупреждение  (warning), правильная вёрстка!
</div>
```

Семантическая верстка упрощает поддержку и развитие CSS, упрощает взаимодействие между членами команды.

Это касается, например, верстальщика и JavaScript-разработчика.

Такая верстка удобна для организации JS-кода. В коде мы просто ставим нужный класс, остальное делает CSS.

Состояние — на элементе

Зачастую компонент может иметь несколько состояний. Например, меню может быть открыто или закрыто.

Состояние должно добавляться CSS-классом на тот элемент, к которому «по смыслу» относится.

Например, меню в закрытом состоянии скрывает свой список элементов. Класс `menu-closed` нужно добавлять не к списку элементов, который скрывается-показывается, а к *корневому элементу* виджета, поскольку это состояние касается всего меню:

```
<div class="menu menu-closed">
  <div class="menu-title">Заголовок меню</div>
  <ul class="menu-items">
    <li>Список элементов</li>
  </ul>
</div>

<style>
  .menu-items {
    display: block; /* открыто по умолчанию */
  }

  .menu-closed .menu-items {
    display: none; /* закрыто */
  }
</style>
```

Префиксы виджета у классов

Посмотрите, пожалуйста, вёрстку для виджета диалогового окна.

```
<div class="dialog">
  <h2 class="title">Заголовок</h2>
  <div class="content">
    Содержимое. Имена классов в этой вёрстке опасны.
  </div>
</div>

<style>
  .dialog .title { стиль заголовка }
  .dialog .content { стиль содержимого окна }
</style>
```

В этой вёрстке есть серьёзная проблема, которая появится, если внутри `.content` будет текст с классом `.title`.

```
<div class="dialog">
  <h1 class="title">Заголовок</h1>
  <div class="content">

    <h2 class="title">Привет!</h2>

  </div>
</div>
```

Такое вполне возможно, ведь диалоговое окно может иметь любое содержимое.

В этом случае CSS-правило `.dialog .title` будет применено и к `h2` в содержимом с непредсказуемыми последствиями.

Конечно, можно попытаться бороться с этим. Например, нейтрализовать его действие, добавив дополнительное правило

`.dialog .content .title` или указав тег `h2`, но это скорее «заплатка», нежели полноценное решение проблемы.

Ещё один вариант — жёстко задать вложенность. А именно, использовать класс `.dialog > .title`. Это сработает в данном конкретном примере, но как быть в тех местах, где нужен более глубокий потомок?

Чтобы избежать возможных проблем, все классы внутри виджета начинают с его имени.

Правильный вариант:

```
<div class="dialog">
  <h1 class="dialog-title">Заголовок</h1>
  <div class="dialog-content">Содержимое</div>
</div>

<style>
  .dialog-title { стиль загловка }
  .dialog-content { стиль содержимого окна }
</style>
```

В этом случае внутрь `.dialog-content` можно смело помещать другие компоненты со своими классами `..-title`, `..-content` и т.п.

Кроме всего прочего, обработка такого CSS будет чуть-чуть быстрее  Так как один класс вместо каскада.

Когда префиксы не нужны?

Префиксы добавляют длины классам. Без них можно обойтись в тех случаях, когда внутри элемента заведомо не будет других компонент. То есть конфликты исключены.

С другой стороны, требования имеют свойство расти. Компоненты зачастую вставляются туда, где их не предполагалось. Ваш виджет, написанный для одной задачи или проекта, может быть потом использован совсем в другом месте, где потребуются вложенные компоненты. И тогда заранее предусмотренные префиксы сослужат хорошую службу.

Итого

- **Вёрстка должна быть семантической, использовать соответствующие смыслу информации теги и классы.**
- **Класс, описывающий состояние компоненты, нужно ставить на её верхнем элементе, а не на том, который нужно «украсить» в этом состоянии.**

  Например, есть индикатор загрузки:

  ```
  <div class="loader">
    <span class="loader-stream">Тут показывается прогресс</span>
  </div>
  ```

  Состояние `"loading"` может влиять только на показ внутреннего `span`, но ставить его нужно всё равно на внешний элемент, ведь это — состояние всей компоненты.

  ```
  <div class="loader loading">
    <span class="loader-stream">Тут показывается прогресс</span>
  </div>
  ```

Возможно и такое, что состояние относится к внутреннему элементу. Например, для дерева состояние открыт/закрыт относится к узлу, соответственно, класс должен быть на узле.

Например:

```
...
<li class="tree-closed">
  Закрытый узел дерева
</li>
...
```

- **Классы внутри компонента должны начинаться с префикса — имени компоненты.**

  Если говорить строго, то это нужно, чтобы избежать проблем в тех случаях, когда:

  - Компонент может содержать произвольный DOM, как например диалоговое окно с произвольным HTML-текстом.
  - В стилях используется селектор потомков `.dialog .title` — только в этом случае CSS для `.dialog .title` будет действовать на элементы с классом `.title` внутри содержимого.

  То есть, если мы абсолютно точно знаем внутреннюю структуру DOM — можно делать любые классы. Но что, если потом «внезапно» понадобится поместить что-то внутрь?

  Если селектор потомков не используется сейчас — это не гарантия, что он не будет использоваться в будущем.

  В принципе, вы можете не следовать этому правилу, если ситуация ясна. Но самый безопасный и расширяемый вариант — использовать префиксы для классов всегда.

Процедурное и объектное программирование

На протяжении долгого времени в программировании применялся процедурный подход [3] . При этом программа состоит из функций, вызывающих друг друга.

Гораздо позже появилось объектно-ориентированное программирование [4] , которое позволяет группировать функции и данные в единой сущности — «объекте».

Например, «пользователь», «меню», «элемент»… Чтобы ООП-подход «работал», объект должен представлять собой законченную, интуитивно понятную сущность.

То есть, объект — это нечто большее, чем просто «группировка» функций и данных. В частности, встроенный объект Math [5] , хотя и содержит функции (`Math.sin`, `Math.pow`, …) и данные (`Math.PI`), не является объектом в смысле ООП. Это не более чем способ группировки: «пространство имён».

Меню, процедурный код

Например, пусть у нас есть HTML-элемент, представляющий собой «меню», вот такой:

```
<div id="food-menu" class="menu">
  <span class="menu-title">Продуктовое меню</span>
  <ul class="menu-items">
    <li>Сыр</li>
    <li>Колбаса</li>
    <li>Торт</li>
  </ul>
</div>
```

Оживим его в процедурном стиле — через создание функций:

```
var foodMenu = $("#food-menu");

function open() {
  foodMenu.addClass('menu-open');
}

function close() {
  foodMenu.removeClass('menu-open');
}

foodMenu.on('click', '.menu-title', function() {
  if (foodMenu.hasClass('menu-open')) {
    close();
  } else {
    open();
  }
});
```

Это ещё называют «простыня кода». Потому что чем больше такого кода — тем сложнее его поддерживать. Особенно это заметно при создании сложных интерфейсов.

Меню, объектный подход

При объектном подходе меню описывается в виде объекта, например:

```
/**
 * options -- объект с параметрами меню.
 *   elem -- элемент меню
*/
function Menu(options) {
  var elem = options.elem;

  this.open = function() {
    elem.addClass('menu-open');
  };

  this.close = function() {
    elem.removeClass('menu-open');
  };

  elem.on('click', '.menu-title', function() {
    if (elem.hasClass('menu-open')) {
      close();
    } else {
      open();
    }
  });

}
```

Теперь мы можем использовать его в любом месте кода:

```
var foodMenu = new Menu({
  elem: $('#food-menu')
});
// ...
foodMenu.open();
```

Сравнение: процедурный и объектный подходы

Нельзя сказать, что объектный подход всегда лучше процедурного.

Как и любой образ мыслей, объектно-ориентированный подход может быть удобен или не удобен, в зависимости от ситуации.

Бывают задачи, в которых сложно выделить сущность. Например, «алгоритм нахождения наибольшего общего делителя». В нём есть два числа и численные операции над ними. Или задача «разложить число на простые».

Для них замечательно подойдёт процедурный подход.

В эту же группу попадает большинство математических алгоритмов, в которых есть сложные вычисления без выделения дополнительных сущностей. Именно поэтому древние языки программирования, предназначенные для расчетов (Фортран, Алгол) не содержали объектных возможностей.

С другой стороны, при разработке интерфейсов мы имеем дело с ярко выраженными сущностями.

Посетитель открывает *окно браузера*, затем нажимает *кнопку*, при этом раскрывается *меню*. Выбор *пункта меню* приводит к *запросу на сервер* — все выделенные слова здесь являются сущностями.

Поэтому ООП отлично подходит для веб-разработки и виджетов. Какие-то сущности можно оформлять в объект, а какие-то — нет, это уже решает разработчик.

Мы продолжим разбор этого подхода в следующих главах.

Внутренний и внешний интерфейс

Отделение внутреннего интерфейса от внешнего — обязательная практика в разработке чего угодно сложнее hello world.

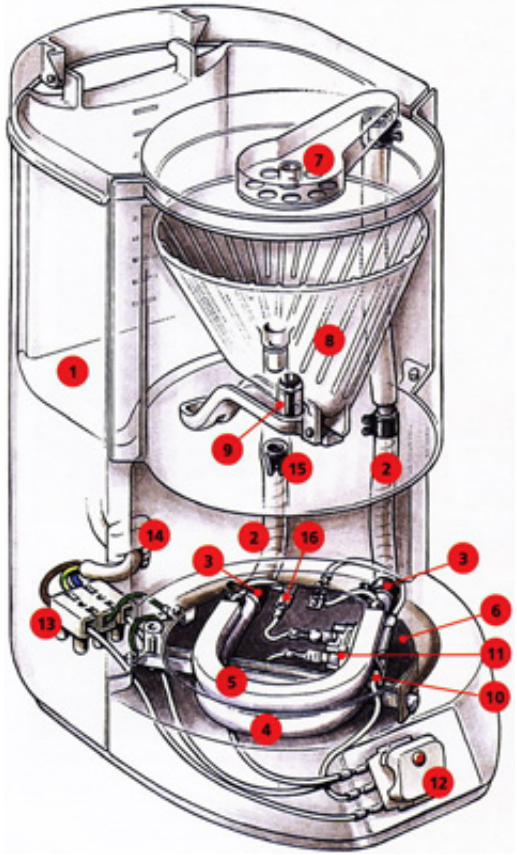
Чтобы это понять, отвлечемся от разработки и переведем взгляд на объекты реального мира.

Как правило, устройства, с которыми мы имеем дело, весьма сложны. Но *разделение интерфейса на внешний и внутренний* позволяет использовать их без малейших проблем.

Пример из жизни

Например, кофеварка. Простая снаружи: кнопка, индикатор, отверстия,… И, конечно, результат — кофе

Но внутри… (картинка из пособия по ремонту)



Масса деталей. Но мы можем пользоваться ей, совершенно не зная об этом.

Кофеварки — довольно-таки надежны, не правда ли? Можно пользоваться годами, и только когда что-то пойдет не так — придется нести к мастеру.

Секрет надежности и простоты кофеварки — в том, что все детали отлажены и *спрятаны* внутри. Если снять с кофеварки защитный кожух, то использование её будет более сложным (куда нажимать?) и опасным (током ударить может). Здесь, как мы увидим, **объекты очень схожи с кофеварками**.

Внутренний и внешний интерфейс

- *Внутренний интерфейс* — это свойства и методы, доступ к которым может быть осуществлен только из других методов объекта.
- *Внешний интерфейс* — это свойства и методы, доступные снаружи объекта.

Все, что спрятано внутри кофеварки: трубка кипятильника, нагревательный элемент, тепловой предохранитель и так далее — это её внутренний интерфейс.

Внутренний интерфейс используется для обеспечения работоспособности объекта, его детали используют друг друга. Например, трубка кипятильника подключена к нагревательному элементу.

Но снаружи кофеварка закрыта специальным кожухом, чтобы никто к ним не подобрался. Детали скрыты и недоступны. Виден лишь внешний интерфейс.

Внутренний интерфейс кофеварки	Внешний интерфейс кофеварки

Резервуар Клапан Гибкая трубка Тепловой предохранитель Хомут гибкой трубки Термостат Трубка кипятильника Нагревательный элемент Контактная колодка Пластина подогрева основания Прижимная планка шнура Обратный клапан Держатель фильтра ...	Отверстие для горячей воды Выключатель Разъем питания Индикатор наполненности Фильтр с молотым кофе ...

Все, что нужно для пользования объектом — это внешний интерфейс. О внутреннем знать не обязательно.

Интерфейсы для `Menu`

Переведём взгляд обратно, на программирование. Посмотрим на меню из предыдущей главы:

```
/**
 * options -- объект с параметрами меню.
 *   elem -- элемент меню
*/
function Menu(options) {
  var elem = options.elem;

  this.open = function() {
    elem.addClass('menu-open');
  };

  this.close = function() {
    elem.removeClass('menu-open');
  };

  elem.on('click', '.menu-title', function() {
    if (elem.hasClass('menu-open')) {
      close();
    } else {
      open();
    }
  });

}
```

Для нашего меню внешним интерфейсом является конструктор `new Menu(options)` и методы `open` и `close`. Их достаточно для использования объекта.

А свойство `elem` является внутренним.

Внутренний интерфейс `Menu`	**Внешний интерфейс `Menu`**
`elem`	`new Menu(options)` `open()`

	`close()`

Защита внутреннего интерфейса

Внутренний интерфейс должен быть закрыт от доступа снаружи.

В терминологии ООП это называется инкапсуляция [6] . Объект может содержать собственные данные и методы, которые закрыты от обращения снаружи. Из других методов объекта к ним можно обратиться, а извне объекта — нельзя.

Для такого разграничения доступа есть несколько причин:

Защита разработчиков.

Представьте, команда разработчиков пользуется кофеваркой. Кофеварка создана фирмой «Лучшие Кофеварки» и, в общем, работает хорошо, но с неё сняли защитный кожух и, таким образом, внутренний интерфейс стал доступен.

Все разработчики цивилизованны — и пользуются кофеваркой как обычно. Но хитрый Вася решил, что он самый умный, и подкрутил кое-что внутри кофеварки, чтобы кофе заваривался покрепче. Вася не знал, что те изменения, которые он произвел, приведут к тому, что кофеварка испортится через два дня.

Виноват, разумеется, тот, кто снял защитный кожух с кофеварки, и тем самым позволил Васе проводить манипуляции.

В программировании — то же самое. Если разработчик при подключении меню будет менять то, что не рассчитано на изменение снаружи — последствия могут быть непредсказуемыми. Лучше защитить Васю от самого себя и от гнева других членов команды, когда они обнаружат, что меню сломалось.

Удобство в поддержке

Ситуация в программировании сложнее, чем с кофеваркой, т.к. кофеварку один раз купили и все, а меню может улучшаться и дорабатываться.

При наличии четко выделенного внешнего интерфейса, разработчик меню может свободно менять внутренние свойства и методы, без оглядки на коллег. Гораздо легче разрабатывать, если знаешь, что ряд методов (все внутренние) можно переименовывать, менять их параметры, и вообще, переписать как угодно, и это напрямую повлияет лишь на другие методы объекта.

Ближайшая аналогия в реальной жизни — это когда мы приносим домой «новую версию» кофеварки, которая работает гораздо лучше. Внутри всё переделано, но пользоваться ей по-прежнему просто (внешний интерфейс сохранён).

Безопасность пользователей

Представим на минуту, что «внутренний интерфейс» открыт (кожуха нет). Кто-нибудь мог бы использовать его для своих нужд особым образом. Скажем, Вася обнаружил увеличение мощности, которое возникает, если контакты замкнуть отверткой. А в новой версии эти же контакты будут означать совсем другое, и Вася может пострадать.

При программировании — пострадает тот код, который написан с использованием внутреннего интерфейса. Придётся искать, что изменилось и его исправлять. Если внутренний интерфейс изначально закрыт, то такая проблема исключена.

Управление сложностью.

Люди обожают пользоваться вещами, которые просты с виду. А что внутри — дело десятое.

Программисты здесь не исключение. Всегда удобно, когда детали реализации скрыты, и доступен простой, понятно документированный внешний интерфейс. Пусть сложность внутреннего интерфейса будет скрыта внутри.

Приватные свойства и методы

В ООП разделяют два основных вида свойств/методов:

- *Приватное свойство* — это такое, которое доступно только изнутри самого объекта.
- *Публичное свойство* — доступно как изнутри, так и из внешнего кода.

Бывают и промежуточные варианты между «приватный» (полностью закрыт для доступа) и «публичный» (полностью открыт).

- В языке C++ можно открыть доступ к приватным переменным одного класса — другому, объявив его «дружественным».
- В языке Java можно объявлять переменные, которые доступны всем классам внутри «пакета».

Термины «приватное свойство/метод», «публичное свойство/метод» относятся к общей теории ООП.

А их конкретная реализация в языке программирования может быть различной. В некоторых языках есть для этого специальные средства, но не в JavaScript.

В JavaScript для приватных свойств можно использовать замыкание.

В частности, в примере меню `var elem` является *приватным свойством*:

```
function Menu(options) {
  var elem = options.elem; // доступно только из других функций Menu

  this.open = function() {
    elem.addClass('menu-open');
  };

  ...
}
```

***Приватный метод* объявляется как локальная функция.**

Например, выделим обработчик клика в приватный метод `onTitleClick`:

```
function Menu(options) {
  var self = this;

  var elem = options.elem;

  elem.on('click', '.menu-title', onTitleClick);

  function onTitleClick(e) {
    if (elem.hasClass('menu-open')) {
      self.close();
    } else {
      self.open();
    }
  }

  self.open = function() {
    elem.addClass('open');
  };

  this.close = function() {
    elem.removeClass('open');
  };

}
```

Функция-обработчик `onTitleClick` вызывается в контексте элемента.

Поэтому, чтобы функция могла обратиться к объекту, ссылка на него копируется в замыкание вызовом `var self = this`. Конечно, вместо `"self"` можно выбрать любое другое имя переменной, но `"self"` используют для этого чаще всего.

Везде `self` вместо `this`

Приватные методы вызываются как функции, не как свойства объекта. Поэтому в них вместо `this` нужно использовать `self`.

```
function Menu() {
  var self = this;

  ...

  function privateMethod() {
    доступ к объекту через self;
  }

  this.close = function() {
    privateMethod(); // this при таком вызове передано не будет
  }
}
```

Даже более того! Наш код — живой. Мы добавляем новые функции, перерабатываем существующие. При этом часть кода может перемещаться из одного метода в другой или выделяться в новый метод.

Представьте себе, что будет, если код, который использует `this`, переместить в функцию, для которой нужен `self` (приватную)?

Нужно будет переименовать все обращения к `this`, обязательно что-то будет забыто, ошибки.. Но есть рецепт.

Поэтому вместо `this` лучше использовать `self`, во всех методах!

Геттеры

Допустим, внешнему коду понадобилось узнать текущее состояние меню: открыто оно или нет. Как это сделать?

Свойства, которые получают информацию из виджета, будем называть *геттерами* (от англ. getter).

В простейшем случае геттер `isOpen` просто возвращает локальную переменную:

```
01 function Menu(menuId) {
02   var self = this;
03
04   var isOpen = false;  // open/close будут менять эту переменную
05   ...
06   this.isOpen = function() {
07     return isOpen;
08   };
09   ...
10 }
```

В нашем случае такой переменной нет, поэтому реализация — проверкой класса:

```
this.isOpen = function() {
  return elem.hasClass('menu-open');
};
```

Если в будущем мы захотим избавиться от переменной `isOpen`, переименовать класс или хранить состояние меню другим способом — это можно будет сделать совершенно незаметно для внешнего кода.

Таким образом, геттеры позволяют скрыть внутреннюю реализацию, как работает открытие/закрытие меню, и выдать наружу простой интерфейс.

Геттер не имеет права производить какие-либо изменения в виджете и его переменных. Он занимается только выдачей информации. Без побочных эффектов.

Название метода-геттера принято начинать `is...`, если возвращается логическое значение (`isOpen`), и с `get..` в остальных случаях (`getValue`).

Сеттеры

Если внешний код хочет явно поставить состояние меню, то это тоже можно обеспечить.

Для установки состояния и данных используются *методы-сеттеры* (setter).

Их название традиционно начинается со слова `set...`.

Простейший метод-сеттер занимается только изменением переменной, и выглядит примерно так:

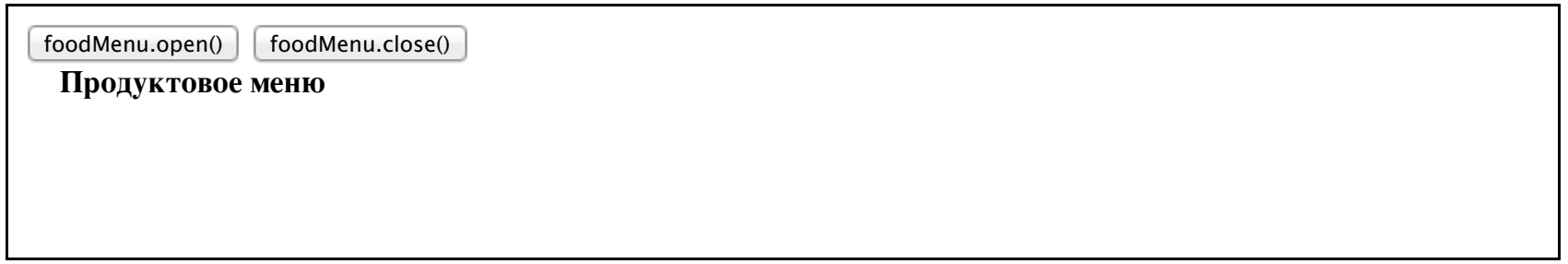
```
this.setOpen = function(newIsOpen) {
  isOpen = newIsOpen;
};
```

..Но в нашем случае при изменении состояния необходимо вызвать соответствующие методы:

```
this.setOpen = function(newIsOpen) {
  if (newIsOpen) this.open();
  else this.close();
};
```

Внутренний интерфейс можно легко менять, главное поддерживать рабочий сеттер.

Итоговый вариант меню с геттером `isOpen` и сеттером `setOpen`:



Геттеры и сеттеры в ES5

Современный стандарт JavaScript предоставляет встроенные языковые свойства для создания геттеров и сеттеров, см. Дескриптор свойства, геттеры и сеттеры [7].

Они доступны во всех современных браузерах, кроме IE<9.

Геттеры и сеттеры позволяют не только скрыть внутренний интерфейс, но и гибко задать режим доступа к свойству:

- Только чтение: есть геттер. Бывает нужно, когда состояние может быть изменено только из самого виджета.
- Только запись: есть сеттер. Например, функции работы с событиями `addEventListener/removeEventListener` позволяют поставить/убрать обработчик, но нет «геттера», который возвращал бы события на элементе.
- Чтение и запись: есть и геттер и сеттер.

Единый геттер-сеттер

Есть альтернативный принцип именования геттеров-сеттеров, который заключается в том, что вместо методов `getData` и `setData` делается один метод с названием `data`. При этом предполагается, что вызов `data()` без аргументов будет возвращать значение, а с аргументом `data(value)` — устанавливать его, если возможно.

Такой подход удобен тем, что название метода становится короче. С другой стороны, может возникнуть путаница между обычными свойствами, вида `this.prop` и свойствами, оформленными в геттер-сеттер `this.prop()`.

Кроме того, смешивать два подхода к именованиям в одном проекте нежелательно. Так получилось, что этот подход менее распространён, чем предыдущий.

Инкапсуляция DOM компоненты

Принцип разделения интерфейсов распространяется и на DOM. Несмотря на то, что документ один и, технически, всё доступно всем..

DOM-структура и классы компонента — его личное дело. Внешний код не имеет права без чрезвычайно уважительной и

продуманной 5 раз причины туда лезть.

Доступ и модификация DOM для меню осуществляется исключительно объектом меню.
Внешний код не имеет права даже узнать, есть ли класс `menu-open` на элементе. Он должен вызвать метод `menu.isOpen()`, который вернёт состояние.

Таким образом гарантируется возможность изменять DOM, добавлять или удалять элементы, классы.

Итого

- При создании объекта нужно делать внешне доступным только тот интерфейс, который действительно необходим для его использования.
- Методы и свойства, которые служат для внутренней реализации, желательно спрятать, чтобы их снаружи даже видно не было.

  В терминологии ООП свойства и методы, которые доступны только внутри объекта, называются «приватными». В некоторых языках есть специальный синтаксис для таких свойств. В JavaScript мы можем использовать локальные переменные:

```
function Menu(options) {
  var self = this; // для обращения к объекту из методов

  this.property = публичное свойство Menu

  var property = приватное свойство Menu

  this.method = function() { публичный метод Menu }

  function method(args) { приватный метод Menu }
}
```

  Параметр конструктора `options` также автоматически является приватным.

Цели, которые мы достигаем, защищая внутренний интерфейс:

- Явная индикация, куда можно лазать, а куда — нет. Защита объекта от внутренних изменений, которые могут «сломать» его.
- Удобство при разработке и доработке. Всегда понятно, какие методы могли быть вызваны снаружи, а какие — нет. Если метод внутренний — его можно рефакторить как угодно.
- Управление сложностью. Разработчик, который хочет использовать объект, должен видеть только простой внешний интерфейс, без перегрузки деталями его функционирования.

Особым случаем является создание виджетов.

Меню должно быть «черным ящиком», всё взаимодействие с которым осуществляется через объект.

DOM-структура графической компоненты должна, по возможности, быть изолирована от внешнего воздействия.

Технически этого добиться сложно, т.к. DOM доступен всем, но тут уже вступают в дело «правила хорошего тона». Есть объект меню — делаем всё через него.

Для того, чтобы разрешить только получение значения — создают методы-геттеры. Чтобы разрешить только изменение — методы-сеттеры.

В современном стандарте JavaScript для этого предусмотрен специальный синтаксис [8].

Соглашения в коде виджета

Здесь мы посмотрим, как лучше структурировать код, чтобы компоненту было удобно разрабатывать и поддерживать.

Пример: меню

Разберём полезные соглашения на примере меню из задачи про выпадающее меню [9].

```
// Виджет получает объект с параметрами options
function Menu(options) {
  var self = this;

  // 1. Инициализация, назначение обработчиков
  var elem = options.elem;

  // 2. Обработчики - приватные методы, не анонимные функции
  // 3. Назначаются через делегирование
  elem.on('click', '.menu-title', onTitleClick);
  elem.on('click', '.menu-items a', onItemClick);

  // 4. Задача обработчика -- запустить нужный метод
  function onTitleClick() {
    if (elem.hasClass('menu-open')) {
      self.close();
    } else {
      self.open();
    }
  }

  // 5. Обработчик не передаёт объект события в вызванный метод
  function onItemClick(e) {
    selectItem( $(e.target) );
    return false;
  }

  // 6. Метод получает только необходимые ему данные
  function selectItem(item) {
    alert( item.html() );
  }

  this.open = function() {
    elem.addClass('menu-open');
  };

  this.close = function() {
    elem.removeClass('menu-open');
  };

}
```

Разбор соглашений в коде

Разберём более подробно каждый пункт, обращая внимание на возможные ошибки.

1. **Инициализация — в ней делается только то, что мы обязаны сделать на этапе создания виджета.**

   Если действие можно отложить — оно откладывается. Например, если календарик показывается при клике — то и его DOM генерируется при клике, а не в момент запуска `new Calendar`.

   Это позволяет экономить ресурсы, ведь, возможно, посетитель вообще никогда не кликнет.

   Фаза инициализации очень чувствительна к производительности, так как при загрузке страницы со сложным интерфейсом на ней создаётся много всего. А мы хотим, чтобы она начала работать как можно быстрее.

   Если изначально подходить к оптимизации на этой фазе «спустя рукава», то потом поправить может быть сложно. Всё-таки, инициализация — это фундамент, начало работы виджета.

2. **В инициализации передаётся объект аргументов `options`, а не список аргументов.**

   Это удобно, так как виджеты часто пополняются новыми (необязательными) аргументами.

3. **Обработчики лучше выносить в отдельные приватные методы.**

   Сравните два фрагмента кода:

   Первый: обработчик — анонимная функция.

```
elem.on('click', '.menu-title', function() {
  if (elem.hasClass('menu-open')) {
    self.close();
  } else {
    self.open();
  }
});
```

   Второй: обработчик вынесен отдельно.

```
elem.on('click', '.menu-title', onTitleClick);
...

function onTitleClick() {
  if (elem.hasClass('menu-open')) {
    self.close();
  } else {
    self.open();
  }
}
```

   Второй подход лучше. И вот почему.

   - Код инициализации, если описывать в нём все обработчики, будет длинным и страшным. Нам же наоборот нужно, чтобы всё было ясно и понятно. Если вынести обработчики — получаем чёткое разделение: что где.
   - Обработчик, который вынесен в отдельный метод, легче найти в коде. Его название можно начинать с `on`, далее обозначаем элемент (`title`) и событие (`click`). Получается `"onTitleClick"`.

4. **Обработчики назначаются через делегирование.**

   Иначе говоря, вместо:

```
elem.find('.menu-title').click(...)
elem.find('.menu-items a').click(...)
```

Пишем так:

```
elem.on('click', '.menu-title', onTitleClick);
elem.on('click', '.menu-items a', onItemClick);
```

В этом сплошные плюсы, так как:

1. Ускоряется инициализация, не надо искать элементы
2. Можно легко менять DOM-структуру, перемещать элементы, генерировать новые, загружать с сервера и т.п.

Поэтому делегирование используется везде, где можно. В некоторых виджет-фреймворках даже делают специальный объект для событий:

```
// Конфигурация событий: селектор => объект с событиями
var events = {
  '.menu-title': {
    'click': onTitleClick
  },
  '.menu-items a': {
    'click': onItemClick
  }
};
```

Этот объект конфигурации передаётся специальной функции, одной для всех виджетов:

```
// делегировать события
function delegateEvents(events) {
  for(var selector in events) {
    elem.on( events[selector], selector );
  }
}

delegateEvents(events);
```

5. **Задачей обработчика события является вызов нужного действия. Как только он определил, что надо сделать — он вызывает соответствующий метод.**

   Например, `onTitleClick` вызывает `open/close`, а `onItemClick` вызывает `select` — функцию выбора элемента меню.

   Сам обработчик занимается только «разбором произошедшего», до той степени, когда понятно что делать с виджетом, но не далее.

   Конечно, код может быть короче, если в обработчике делать всё — и разбор события и операции над виджетом. Но в таком разделении есть важный плюс. При чтении кода, когда смотришь в сложный обработчик, бывает непонятно «что же он делает?» На то, чтобы разобраться, тратится время.

   Имя функции является отличным комментарием. Сразу видно, какое действие происходит.

   Кроме того, при большом обработчике такое выделение функции делает код чище. Мы получаем два коротких и понятных метода вместо одного длинного.

6. **Объект события остаётся в обработчике.**

   В метод, производящий операцию над виджетом, передаются лишь нужные ему аргументы.

   Это нужно как для правильного разделения действий, так и для того, чтобы метод можно было использовать не из обработчика. В случае с меню, метод `selectItem(item)` может быть в будущем вызван в любом месте кода.

7. **Везде используются jQuery-элементы.**

   Это, конечно же, если мы вообще используем jQuery.

   Чтобы избежать путаницы — где у нас DOM-элемент, а где jQuery-коллекция, все элементы, как только это возможно, оборачиваются в jQuery. Как правило, получается, что единственное место в коде, где мы оперируем с обычным DOM — это обработчик события:

   ```
   function onItemClick(e) {
     // тут же обернуть e.target в jQuery
     selectItem( $(e.target) );
     return false;
   }
   ```

   Такая политика хороша тем, что не возникает вопросов — что у нас: «обычный» элемент или jQuery-коллекция. И не нужно «на всякий случай» оборачивать элемент на входе функции, вот так:

   ```
   function selectItem(item) {
     item = $(item); // эта строка не нужна
     ...
   }
   ```

   Важное исключение в её применении — соображения производительности, так как такое оборачивание — это дополнительные JavaScript-вызовы. Кроме того, обычный JavaScript позволяет многое сделать быстрее и более оптимально, чем jQuery.

Итого

Выше перечислены полезные в разработке подходы. Если вы не запомнили их все — ничего страшного. Вы сможете вернуться к ним на практике, при решении задач.

Свои события, подписка-уведомление

Обычные JavaScript-события можно расширить на произвольные объекты. Если обычный `<select>` взаимодействует с кодом через события `change`, `focus` и другие, то почему бы также не делать и нашему компоненту (меню)?

Паттерн «подписка-уведомление» означает, что любые объекты могут генерировать события и подписываться на них.

Свои события на элементах

Перед тем, как обсуждать произвольные объекты, рассмотрим свои DOM-события на элементах, которые предлагает jQuery, а впрочем и многие другие фреймворки тоже.

Метод trigger [10] позволяет генерировать на элементе любое событие `eventName`, передав данные из массива `dataArr`.

У него два синтаксиса.

1. Первый — `trigger(eventName, dataArr)`, где `dataArr` — массив с аргументами для обработчика:

```
$('body').on('test', function(e, pageX, pageY) {
  alert( pageX + pageY );
})

$('body').trigger('test', [100, 200]);
```

Обратите внимание — обработчик получает первым аргументом событие. Этот объект генерируется автоматически и содержит `type`, `timestamp` и прочие параметры, которые обычно не нужны. Сами данные идут за ним.

2. Второй — `trigger(event)`, где `event` — любой объект события:

```
$('body').on('test2', function(e) {
  alert( e.pageX + e.pageY );
})

$('body').trigger({
  type: 'test2', // любое название
  pageX: 100,    // любые свойства
  pageY: 200
});
```

Сложно решить, какой синтаксис лучше. Второй — гибче, первый с именованными параметрами — нагляднее.

Здесь и далее мы будем использовать второй синтаксис как более гибкий и близкий к обычным событиям.

Например, выбор элемента меню будет сопровождаться событием `select`:

```
...
function selectItem(item) {
  var value = item.attr('href');

  elem.trigger({
    type: 'select',
    value: value
  });
}
...
```

Внешний код, который хочет реагировать на выбор посетителя, подписывается на это событие:

```
// для подписки на событие используется фреймворк jQuery
// внешний код:
var menu = new Menu({
  elem: $('#food-menu')
});

$('#food-menu').on('select', function(e) {
  alert('Вы выбрали ' + e.value);
});
```

Меню с таким обработчиком:

Продуктовое меню

- Сыр
- Колбаса
- Торт

Метод `trigger` эмулирует обработку, близкую к «родным» браузерным событиям. В частности, любое событие всплывает.

При `trigger` не сработают обработчики, назначенные через `addEventListener`

Это результат технической реализации событий в jQuery.

- Когда jQuery назначает обработчик (метод `on`) — ссылка на него сохраняется в специальной структуре данных, привязанной к элементу.
- Вызов `elem.trigger('click'...)` идёт вверх по DOM от `elem` и просматривает элементы на предмет наличия этой структуры, а в ней — обработчика `click`. Если находит — выполняет.

При этом наличие обработчика, присвоенного в `elem.onclick` он также видит, а вот назначенного через `addEventListener` — не может обнаружить.

Не генерируйте браузерные события

Никогда не следует генерировать браузерные события, например `click` на ссылках.

Несмотря на то, что это может быть соблазнительно, в 95% случаев такой код неверен и его придётся переделывать. Просто наблюдение. Сравните его со своим опытом.

Больше информации о генерации своих событий на элементе вы можете получить в документации к trigger [11].

Свои события на любых объектах

Можно дать любому объекту возможность генерировать события и подписываться на них.

Во-первых, такую возможность даёт jQuery. Например:

```
var obj = {};

// поставить обработчик
$(obj).on('select', function(e) {
  alert(e.value);
  alert(e.target == obj && this == obj); // сам объект тоже есть в событии!
});

// запустить обработчик
$(obj).triggerHandler({
  type: 'select',
  value: "Тест"
}); // Тест
```

Технически, методы `on/off/triggerHandler` в jQuery реализованы через хранение обработчиков в самом объекте `obj` (jQuery добавляет в него свои свойства). Поэтому совершенно неважно, что мы сделали две независимые обёртки в jQuery.

Изменения в коде меню, необходимые для генерации события на объекте вместо элемента — минимальные: просто поменять `elem` на `$(self)`.

Было событие на элементе:

```
function selectItem(item) {
  elem.trigger({
    type: 'select',
    value: item.attr('href')
  });
}
```

Станет событие на объекте:

```
function selectItem(item) {
  $(self).triggerHandler({
    type: 'select',
    value: item.attr('href')
  });
}
```

Для jQuery: не `trigger`, но `triggerHandler`!

Вместо `triggerHandler` можно использовать и `trigger`. Но у этого метода, так как он предназначен для элементов, есть одна неприятная особенность: он вызывает метод объекта, если его название совпадёт с типом события.

Например, если в объекте есть метод `select`, то `$(obj).trigger({ type: 'select'})` вызовет этот метод.

Попробуйте сами:

```
var obj = {
  select: function() { alert('метод!'); }
}

$(obj).trigger({ type: 'select' }); // выведет "метод!"
```

Это сделано для корректного срабатывания обработчиков, назначенных не через jQuery, но при работе с произвольными объектами, однако, это скорее неприятность, нежели удобство.

А `triggerHandler` имеет тот же синтаксис, что и `trigger`, но не умеет работать с коллекциями элементов (событие будет только на первом), и, что нам важно, не вызовет ненароком метод объекта.

Без jQuery

Можно сделать события и без jQuery. Достаточно добавить в объект три метода: `on`, `off` и `trigger`.

Вот — пример их реализации:

```
var EventMixin = {

  /**
   * Подписка на событие
   * Использование:
   *  menu.on('select', function(item) { ... }
  */
  on: function(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = [];
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * Прекращение подписки
   *  menu.off('select',  handler)
   */
  off: function(eventName, handler) {
    var handlers = this._eventHandlers[eventName];
    if (!handlers) return;
    for(var i=0; i<handlers.length; i++) {
      if (handlers[i] == handler) {
        handlers.splice(i--, 1);
      }
    }
  },

  /**
   * Генерация события с передачей данных
   *  this.trigger('select', item);
   */
  trigger: function(eventName) {

    if (!this._eventHandlers[eventName]) {
      return; // обработчиков для события нет
    }

    // вызвать обработчики
    var handlers = this._eventHandlers[eventName];
    for (var i = 0; i < handlers.length; i++) {
      handlers[i].apply(this, [].slice.call(arguments, 1));
    }

  }
}
```

Чтобы использовать методы `EventMixin` — нужно скопировать эти методы из него в произвольный объект.

Пример использования:

```
// предполагается, что в объект menu скопированы методы из EventMixin

// подписка
menu.on('select', function(item) {
  alert( item.html() );
}

// генерация события
menu.trigger('select', item);
```

Почти каждый JavaScript-фреймворк предоставляет средства для генерации событий. Остальное — детали синтаксиса.

Далее, для простоты, будем использовать для событий jQuery.

Практика, практика, практика!

Шаблонизация в JavaScript

В этой главе мы рассмотрим *шаблонизацию* — удобный способ генерации HTML-структуры для виджета, различные варианты и её место в архитектуре.

Все виджеты можно условно разделить на три группы по генерации DOM-структуры:

1. **Получают готовый DOM и «оживляют» его.**

   До этого момента, большинство виджетов учебника были именно такими. Мы много их уже создали, так что пойдём дальше.

2. **Создают DOM для компоненты самостоятельно.**
   В ряде случаев нужно генерировать DOM динамически. Например, виджет «подсказка» при наведении на элемент создаёт красивую подсказку и показывает её. Элемент генерируется в JavaScript-коде.

   Или красивое диалоговое окно для взаимодействия с посетителем — его тоже надо нарисовать перед показом.

3. **Должны уметь как «оживить» уже готовый DOM, так и создать свой.**
   Бывает и так, что виджет должен уметь и то и другое.

   Например, так работает Twitter, который при загрузке сразу показывает текущие сообщения (HTML генерируется на сервере), но может динамически добавлять новые.

   Как правило, так делают для оптимизации. Браузер быстро скачивает HTML и отображает его, посетитель видит текущие сообщения и рад этому. А затем уже подгружается объёмный JavaScript, который умеет загружать сообщения с сервера, создавать их и т.д.

С первым типом виджетов — вопросов нет. Добавляем обработчики, и дело сделано.

Интересно начинается со второго типа, и совсем интересно — с третьим типом.

Меню: создаём DOM в JS

До этого мы неоднократно работали с меню, которое использует существующую DOM-структуру. Теперь сделаем так, что оно будет

генерировать свой DOM динамически.

Вначале сделаем это при помощи jQuery-методов. Функцию, которая будет этим заниматься, назовём `render`. Она может выглядеть так:

```
function Menu(options) {

  var items = options.items;
  var elem;

  function render() {
    elem = $('<ul class="menu"/>');
    $(items).each(function() {
      var li = $('<li/>').appendTo(elem);
      li.append( $('<a/>', {
        href: '#'+this.link,
        html: this.html
      }) );
    });
  }

  ...
}
```

Посмотрите, пожалуйста, на код выше. Понятен ли он? Очевидно ли, что метод `render` генерирует такую структуру HTML (для соответствующих данных)?

```
<ul class="menu">
  <li><a href="#javascript">JavaScript</a></li>
  <li><a href="#html">HTML</a></li>
  <li><a href="#css">CSS</a></li>
</ul>
```

…Скорее всего, не очень понятен, нужно внимательно смотреть на код, чтобы разобраться. Причём, это — с jQuery, если на обычном JavaScript написать, код ещё длиннее и сложнее будет.

…А что, если нужно изменить создаваемый HTML? …А что, если эта задача досталась не программисту, который написал этот код, а верстальщику, который с HTML/CSS проекта знаком отлично, но этот JS-код видит впервые? Вероятность ошибок при этом зашкаливает за все разумные пределы.

К счастью, генерацию HTML можно упростить. Для этого воспользуемся шаблонами.

Меню: на шаблонах

Шаблон — это заготовка, которая путём подстановки значений (текст сообщения, цена и т.п.) превращается в DOM/HTML.

Например, шаблон для меню:

```
<ul class="menu">
<% $(items).each(function() { %>
  <li><a href="#<%=this.link%>"><%=this.html%></a></li>
<% }); %>
</ul>
```

Для работы с таким шаблоном используется специальная функция `_.template`, которая предоставляется фреймворком (посмотрим далее).

Взглянем, каким может быть метод `render` с использованием шаблона, просто чтобы оценить простоту:

```
// шаблон должен быть передан как options.template
function render() {
  var html = _.template(options.template, { items: items });
  elem = $(html);
}
```

Этот метод `render` выглядит проще, не правда ли? Шаблоны тоже довольно просты, сейчас мы поближе познакомимся с ними и посмотрим, как всё работает.

Строковые шаблоны

Шаблонных систем для JavaScript много. Для начала мы рассмотрим одну из них, предложенную Джоном Ресигом [12] и модифицированную в библиотеке LoDash [13]. Есть и другие модификации, но все они похожи.

Сначала рассмотрим синтаксис шаблона, а затем — функцию для его обработки.

Синтаксис шаблона

В шаблоне, как вы наверняка уже заметили выше, используется HTML и специальные разделители:

`<% code %>`
: Код между разделителями `<% ... %>` будет выполнен «как есть»

`<%= expr %>`
: Переменная или выражение внутри `<%= ... %>` будет вставлено «как есть». Например: `<%= (x+1)*f(5) %>`.

`<%- expr %>`
: Переменная или выражение внутри `<%- ... %>` будет вставлено с экранированием. Например, если `str = "<script>"`, то при <%- str %> в результат попадёт `<script>`.

Хранение шаблона в документе

Шаблон можно хранить прямо на странице, в её HTML. Чтобы текст шаблона не показывался, заключим его в блок `SCRIPT` с нестандартным `type`, например `"text/template"`:

```
<script type="text/template" id="list-template">
<ul>
  <% for (var i=1; i<=count; i++) { %>
    <li><%=i%></li>
  <% } %>
</ul>
</script>
```

Такие скрипты не выполняются, их содержимое игнорируется браузером, но доступно при помощи `innerHTML`.

Почему использован именно нестандартный `SCRIPT`, а не `<div style="display:none">`? Как вы считаете, что лучше?

Основная причина — шаблон может быть любым, даже некорректным HTML. В `DIV` доставить незакрытым тег — и могут быть проблемы. А в скрипте может быть почти что угодно, его содержимое полностью игнорируется.

Кроме того, содержимое `DIV` браузер обрабатывает, добавляет его в DOM документа, но там оно совсем не нужно, это же шаблон.

Функция `_.template`

Для работы с шаблоном в библиотеке LoDash [14] есть функция `_.template(tmpl, data, options)`.

Аргументы:

`tmpl`
: Шаблон.

`data`
: Объект с данными.

`options`
: Дополнительные настройки, см. документацию.

Эта функция «запускает» сборку шаблона с объектом `data`. Затем возвращает результат в виде строки.

Например:

```
<!-- файл с кодом шаблонки -->
<script src="http://cdnjs.cloudflare.com/ajax/libs/lodash.js/0.9.2/lodash.js"></script>

<!-- шаблон для списка от 1 до count -->
<script type="text/template" id="list-template">
<ul>
  <% for (var i=1; i<=count; i++) { %>
    <li><%=i%></li>
  <% } %>
</ul>
</script>

<script>
  var tmpl = document.getElementById('list-template').innerHTML;
  var data = { count: 5 };

  // ..а вот и результат
  var result = _.template(tmpl, data);
  document.write( result );
</script>
```

Как работает функция `_.template`?

Вызов `_.template(tmpl, data)` сначала разбирает строку `tmpl` по разделителям и превращает её в JavaScript-функцию, при помощи вызова `var compiled = new Function('obj', 'ТЕКСТ')`, где `ТЕКСТ` генерируется динамически и выглядит примерно так:

```
var __p = [];
with(obj) { // 'obj' - первый аргумент функции
  __p.push("<ul>");
  for (var i=1; i<=count; i++) {
    __p.push("<li>", i, "</li>");
  }
  __p.push("</ul>");
}
return __p.join("");
```

…То есть, сначала создаётся массив `__p`, который будет содержать результат. В него добавляются (`push`) фрагменты HTML, а также переменные из выражений `<%=...%>`. Код из `<%...%>` копируется в функцию «как есть».

Доступ к переменным шаблона из функции обеспечивает `with(obj) { .. }`.

Вторым этапом этой функции передаётся объект данных, т.е. происходит вызов `compiled(data)`, результат которого и будет готовой строкой.

Эти два процесса можно разделить, вот так:

```
var compiled = _.template(tmpl); // вызываем _.template без второго аргумента

var result = compiled(data); // запускаем откомилированный шаблон compiled

...
// в дальнейшем можно также вызывать уже скомпилированный шаблон compiled
```

Настройки шаблонизатора

- Любые разделители можно поменять.
- Название `obj` можно поменять на любое другое, например `items`, а вставку `with` — отключить.

Как раз для этого используется 3-й аргумент функции `_.template`. Читайте документацию, оно того стоит 

Отладка шаблонов

Что, если в шаблоне ошибка? Например, синтаксическая.

Конечно, ошибки будут возникать, куда же без них. Шаблон компилируется в функцию, и отладчик, скорее всего, покажет ошибку в ней. Но так как мы знаем устройство функции, то обычно можно разобраться, где именно проблема в исходном шаблоне.

А чтобы было легче понять, какому именно шаблону соответствует функция, шаблонизатор обеспечивает нам информационную поддержку (речь о LoDash): можно передать свою строку и функция будет ей помечена.

Выглядит это так:

```
_.template(tmpl, data, { sourceURL: "/my/template/url" });
```

Теперь при отладке в Chrome можно будет видеть, что ошибка происходит не в неизвестной «анонимной» функции, а в функции с данного `sourceURL`.

Кроме того, при компиляции текст шаблона копируется в свойство `source` создаваемой функции, так что можно получить его в консоли, например как `arguments.callee.source`.

Продвинутая шаблонизация

«Включение» шаблонов друг в друга

Зачастую бывает так, что один шаблон описывает фрагмент, который может появляться во многих контекстах.

Например, пользователя мы выводим при помощи такого шаблона:

```
<script type="text/template" id="user-template">
  <div class="user">
    <h2><%=firstName%> <%=lastName%></h2>
    <div class="age"><%=age%></div>
    <div class="email"><a href="mailto:<%=email%>"><%=email%></a></div>
  </div>
</script>
```

Информация о пользователях может выводиться в списке посетителей сайта, в списке людей онлайн, да и просто — как подсказка при наведении на ссылку с именем.

В каждом из этих виджетов есть свои шаблоны, которые могут использовать `user-template` для вывода посетителя.

Самый прямой способ — явно получать и вызывать `user-template` внутри другого шаблона:

```
<script type="text/template" id="user-template">
  <ul class="user-list">
    <% $(items).each(function() { %>
    <li><%=_.template($('#user-template').html(), this)%></li>
    <% }); %>
  </ul>
</script>
```

Это будет работать, но шаблон каждый раз компилируется заново. Поэтому если уж использовать такой подход, то придумать какой-то глобальный объект, который кеширует шаблон и возвращает его — нечто вроде `$.Template('user-template')`.

Альтернатива этого подхода — передать нужный шаблон через локальную переменную `itemTemplate`:

```
...
  <% $(items).each(function() { %>
  <!-- itemTemplate передаётся текущему шаблону из виджета -->
  <li><%=itemTemplate(this)%></li>
  <% }); %>
...
```

Используйте такую передачу для включения шаблона при решении задачи ниже.

Автосоздание переменных с DOM-узлами

Бывают ситуации, когда ссылки на определённые подэлементы нужны в виджете. Например, в случае с меню из решения задачи Меню с шаблоном списка и элемента [15], ссылка на элемент `UL` со списком элементов может понадобиться, чтобы добавлять новые элементы меню.

Есть много подобных примеров, когда какие-то элементы нужны после создания виджета. Иногда такие подэлементы называют *«точками прикрепления»*, т.к. именно в этих местах JS-код связан с DOM.

Можно каждый раз, когда нужно, искать их: `elem.find('.menu-items')`. Но обычно более эффективно — находить их один раз при генерации DOM и прикреплять.

Для удобства можно автоматизировать этот процесс. А именно, помечать все элементы, требующие прикрепления, атрибутом вида

`data-attach="items"`. Значением атрибута будет имя свойства, в котором нужно сохранить ссылку на узел.

В метод `render` добавим поиск элементов с таким атрибутом и запись их в переменную `attach`. Так что элемент будет доступен как `attach.items`.

```
var attach = {};
var html = строка-шаблон;

// сохранит элементы с data-attach в объекте attach
function render() {
  elem = $(html);

  elem.find('*[data-attach]').each(function() {
    var prop = $(this).data('attach');
    attach[prop] = $(this);
  });

}

// доступ к элементу с data-attach="items"
this.addItem = function() {
   attach.items.append( _.template(...) );
};
```

Размеченный шаблон с `data-attach`:

```
<script type="text/template" id="menu-template">
<div class="menu">
  ...
  <ul class="menu-items" data-attach="items">
    <% $(items).each(function() { %>
      <%=itemTemplate(this)%>
    <% }); %>
  </ul>
</div>
</script>
```

Смысл этих действий:

- Универсальный код прикрепления может быть вынесен во фреймворк, так что код компоненты будет короче.
- При исправлении шаблона *видно*, в каких местах что привязывается. В результате изменений точка прикрепления будет перенесена, но не потеряна. То есть, изменение шаблонов станет лучше защищено от ошибок.

Шаблонка «на узлах»

Альтернативный подход к шаблонизации — это клонировать готовый DOM-узел.

Например:

```
<script src="http://code.jquery.com/jquery.js"></script>

<!-- шаблон -->
<div id="user-template" style="display:none">
  <div>Имя: <span class="user-name"></span></div>
  <div>Возраст: <span class="user-age"></span></div>
</div>

<script>
// /клонировать user-template и записать данные
function makeUserNode(data) {
  return $('#user-template')
    .clone()
    .show()
    .prop('id', 'user-' + data.id)
    .find('.user-name').html(data.name).end()
    .find('.user-age').html(data.age).end();
}

// создать два узла из шаблона
makeUserNode({ id: 1, name: 'Паша', age: 25 }).appendTo('body');
makeUserNode({ id: 2, name: 'Василий', age: 29 }).appendTo('body');
</script>
```

Как видно, код для заполнения получился гораздо больше. Но его можно автоматизировать, поставив в DOM специальные атрибуты, которые будут описывать что куда вставлять.

Шаблонка читает такой DOM-узел, анализирует атрибуты, а потом клонирует. Такой подход реализован в библиотеках Knockout.JS, Angular.JS

Связывание JS-DOM

Полноценное связывание JS-DOM — логичный шаг в направлении «автоприкрепления».

Ведь много кода уходит на то, чтобы сказать «при изменении этой переменной запиши ее в свойство того DOM-элемента». Или «этот массив нужно отображать в виде такого-то списка UL/LI».

Так давайте указывать соответствие JS-DOM при помощи атрибутов, а следить за ними будет специальная библиотека, которая для этого и написана.

Такие библиотеки есть, например это делают Knockout.JS [16], Angular.JS [17].

Пример разметки шаблона, который поддерживает соответствие списка `UL` и массива `people`:

```
<!-- Пример шаблона для библиотеки Knockout.JS -->
<h4>Люди</h4>
<ul data-bind="foreach: people">
    <li>
        Имя номер <span data-bind="text: $index"></span>:
        <span data-bind="text: name"></span>
        <a href="#" data-bind="click: $parent.removePerson">Удалить</a>
    </li>
</ul>
<button data-bind="click: addPerson">Добавить</button>
```

В этом шаблоне, как видно, указаны и обработчики. Они тоже будут привязаны автоматически.

Как правило, для этого используется DOM-шаблонизация, а не строковая. Например, `foreach` в цикле клонирует шаблон и вставляет элементы.

Генерация на сервере и на клиенте

Самый интересный случай — когда нужно использовать шаблон и на клиенте и на сервере.

Например, сервер генерирует страницу со списком сообщений. А на клиенте JavaScript получает новые сообщения и добавляет их.

Как сделать единый шаблон «сообщение», который бы работал и на клиенте и на сервере?

Здесь есть два подхода.

- **Организовать шаблонизацию при помощи JavaScript на сервере.** Это проще всего, если серверная часть написана на JavaScript. Но даже и для других языков, JavaScript-машина V8 [18] легко интегрируется с ними. Есть успешные реализации JavaScript-шаблонизации при серверной части на Java, Perl, PHP.
- **Использовать шаблоны без логики.**
  Есть шаблонные системы, например mustache [19], которые содержат вместо команд специальные «команды-помощники» («helper»), описываемые в плагинах к шаблонизатору.

  Например,

  ```
  <ul class="people_list">
    {{#each people}}
    <li>{{this}}</li>
    {{/each}}
  </ul>
  ```

  Функционал `each` описан в шаблонизаторе. На PHP это один код, на Java — другой, на JavaScript — как вы, наверное, уже догадались, это будет цикл `$(people).each(...)`.

  В результате сам текст шаблона полностью кросс-платформенный. Но удобно ли это? Решать вам, на практике есть удачные применения и этого подхода тоже.

Сжатие и сборка

Современные системы организации проекта, например requireJS [20], позволяют вынести шаблоны из тела страницы в отдельные файлы. Они подключаются непосредственно из JavaScript.

В процессе сборки такой шаблон будет скомпилирован в функцию и сохранён в `.js`-файл. Для этого в систему сборки нужно добавить плагин, соответствующий шаблонной системе, например https://github.com/ZeeAgency/requirejs-tpl.

Итого

Шаблоны полезны для того, чтобы отделить HTML от кода. Это упрощает разработку и поддержку. Код становится понятнее.

- Мы разобрали систему шаблонизации из библиотеки LoDash [21]. Есть и другие.
- Шаблонки бывают «на строках» и «на узлах». Первые работают преобразованием строки в функцию, вторые — клонируют готовый узел. Строки — более универсальны.
- Для того, чтобы использовать один и тот же шаблон на клиенте и на сервере, нужно либо поставить JavaScript на сервере (подключить V8 [22] не так сложно), либо использовать шаблоны без JavaScript, например mustache [23].

Используйте шаблоны в дальнейшем, при разработке виджетов.

Еще практика!

Решения задач

Решение задачи: Семантическое меню

Несмотря на то, что меню более-менее прилично отображается, эта вёрстка совершенно не семантична.

Ошибки:

1. Во-первых, меню представляет собой *список элементов*, а для списка существует тег `LI`.

   Семантический подход — это когда теги используются по назначению. Для элементов списка `<li>`, для адреса `<address>`, для заголовка таблицы `<th>` и т.п.

2. Во-вторых, класс `rounded-horizontal-blocks` показывает, что содержимое должно быть *оформлено* как скругленные горизонтальные блоки. Любой класс, отражающий оформление, несемантичен.

   Правильно — чтобы класс был *смысловым*. Например, класс `primary-menu` говорит о том, что смысл элемента — «главное меню».

3. В-третьих, элементами меню в данном случае являются ссылки. Их целесообразно сделать `<a href="...">`. Если есть возможность, то желательно указать и URL для клиентов, в которых не работает JavaScript (поисковые роботы).

Вариант ниже — семантичен:

```
<ul class="menu">
  <li><a href="#">Главная</a></li>
  <li><a href="#">Товары</a></li>
  <li><a href="#">Фотографии</a></li>
  <li><a href="#">Контакты</a></li>
</ul>
```

Полное меню со стилями: http://learn.javascript.ru/play/tutorial/widgets/menu-semantic.html

Решение задачи: Ошибки в вёрстке

- Самая главная ошибка: классы без префиксов.
- Класс `selected` находится на ссылке:

```
<li><a href="#tabs-3" class="selected">Открытая вкладка</a></li>
```

  ..Но состояние «выбранности» относится к заголовку `LI` целиком. Для его нормального отображения класс должен быть на `LI`.

- В использовании `.tabs > div` особо криминала нет, но нужно учесть, что добавить `DIV` с другим функционалом на этот уровень будет затруднительно. Иначе говоря, такая вёрстка усложняет расширение виджета. Лучше использовать класс `.tabs-tab`.

Правильный вариант:

```
<div id="tabs" class="tabs">
  <ul class="tabs-headers">
    <li><a href="#tabs-1">Вкладка 1</a></li>
    <li><a href="#tabs-2">Вкладка 2</a></li>
    <li class="tabs-selected"><a href="#tabs-3">Открытая..</a></li>
  </ul>
  <div class="tabs-tab" id="tabs-1">Содержимое...</div>
  <div class="tabs-tab" id="tabs-2">Содержимое...</div>
  <div class="tabs-tab" id="tabs-3" class="tabs-selected">
    Посетитель видит содержимое третьей вкладки.
  </div>
</div>
```

Решение задачи: Написать объект с геттерами и сеттерами

```
function User() {

  var firstName, surName;

  this.setFirstName = function(newFirstName) {
    firstName = newFirstName;
  };

  this.setSurname = function(newSurname) {
    surname = newSurname;
  };

  this.getFullName = function() {
    return firstName + ' ' + surname;
  }
}

var user = new User();
user.setFirstName("Петя");
user.setSurname("Иванов");

alert( user.getFullName() ); // Петя Иванов
```

Решение задачи: Часики

Решение: http://learn.javascript.ru/play/tutorial/widgets/clock/index.html.

Решение задачи: Выпадающее меню

Допишем конструктор `Menu`:

```
function Menu(options) {
  // ...

  elem.on('click', '.menu-items a', onItemClick);

  function onItemClick(e) {
    selectItem( $(e.target) );
    return false;
  }

  function selectItem(a) {
    alert( a.html() );
  }
}
```

Обратите внимание: обработчик (`onItemClick`) отделён от метода, который выполняет действие (`selectItem`).

Цель обработчика получить событие, свести всё к элементам (обёрнутым в jQuery) и данным, и передать вызов дальше.

Так код становится более читаемым и расширяемым, каждый метод делает своё.

Полный код `Menu`: http://learn.javascript.ru/play/tutorial/widgets/menu-click/index.html.

Решение задачи: Компонент: список с выделением

Решение: http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-component.html

Решение задачи: Слайдер-компонент

Пример переписанного слайдера:

http://learn.javascript.ru/play/tutorial/widgets/slider-component.html

Решение задачи: Голосовалка

Решение: http://learn.javascript.ru/play/tutorial/widgets/voter/index.html.

Решение задачи: Голосовалка "на событиях"

http://learn.javascript.ru/play/tutorial/widgets/voter-event/index.html.

Решение задачи: Список с выделением и событием

Решение: http://learn.javascript.ru/play/tutorial/widgets/selectable-list-events.html

Обратите внимание:

- `onLiClick` не генерирует событие `select`. Это обработчик, его роль — разобраться, что происходит, и передать работу нужным методам.
- В событие передаётся массив значений `value`. Код виджета производит всю работу по подготовке этого значения. Было бы совершенно недопустимо, хотя это проще, передавать массив выбранных `LI`. Вообще, элементы — это внутреннее дело компонента, они могут измениться в любой момент, доступ к ним снаружи крайне нежелателен.
- Код получения значений вынесен в отдельную функцию `getValues` — для чистоты (каждая функция делает свою работу), и потому что скорее всего она ещё где-то понадобится.

Решение задачи: Свой селект

Решение: http://learn.javascript.ru/play/tutorial/widgets/customselect/index.html

Решение задачи: Подсказка над элементом

Подсказки

Функция показа подсказки должна при первом показе сгенерировать элемент с подсказкой, а затем — показать в нужном месте страницы.

Обсудим, как это сделать.

Для генерации подсказки добавим вспомогательную функцию `getTooltipElem()`. Она будет возвращать существующий элемент, если он есть, а если нет — генерировать новый.

```
function getTooltipElem() {
  if (!tooltipElem) {
    tooltipElem = $('<div/>', {
      "class" : 'tooltip',
      html: html
    });
  }
  return tooltipElem;
}
```

Основная настройка вида подсказки — в CSS-классе `tooltip`.

Например:

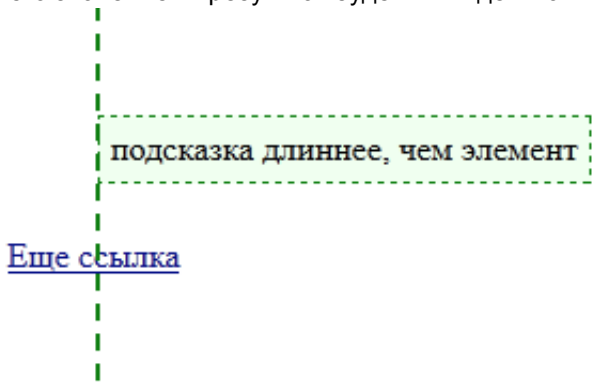
```
.tooltip {
  position:absolute;
  z-index:100; /* подсказка должна перекрывать другие элементы */
  padding: 10px 20px;

  /* красивости... */
  border: 1px solid #b3c9ce;
  border-radius: 4px;
  text-align: center;
  font: italic 14px/1.3 arial, sans-serif;
  color: #333;
  background: #fff;
  box-shadow: 3px 3px 3px rgba(0,0,0,.3);
}
```

Как правильно отпозиционировать подсказку? Для начала, по горизонтали.

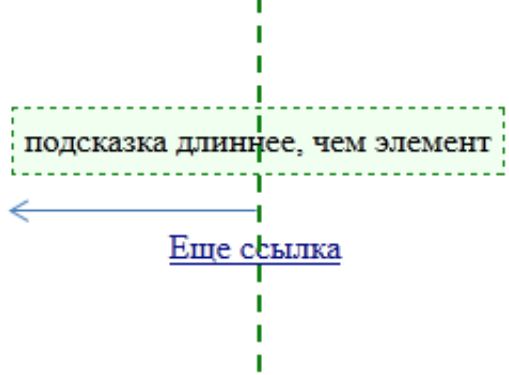
Центр подсказки должен быть ровно над центром элемента, который имеет координату `elem.offset().left + elem.outerWidth()/2`.

Если поставить `tooltipElem.left` в это значение — результат будет выглядеть так:




```
tooltipElem.left = elem.offset().left + elem.outerWidth()/2
```

Дополнительно нужно сдвинуть подсказку на половину собственной ширины влево:




```
left -= tooltipElem.outerWidth()/2;
```

Теперь отпозиционируем по вертикали.

Это гораздо проще: нужно взять координату Y элемента и вычесть из неё высоту подсказки и дополнительный отступ.

```
elemCoords.top - tooltipElem.outerHeight() - 10
```

Проверка решения

Проверьте, пожалуйста, ваше решение на предмет возможных ошибок:

- Подсказка корректно работает при прокрутке?
- Не используете ли вы события `mouseout/mouseover`? Лучше — `mouseenter/mouseleave` (или, более кратко в jQuery: `hover`).
- Вторая подсказка с `offset:0` расположена сразу над элементом, а не как первая.
- Элемент с подсказкой генерируется динамически, при наведении, а не когда подсказка только создаётся?
- Элемент с подсказкой *позиционируется* при показе, а не при создании? Ведь элемент, на котором стоит подсказка, может менять своё положение.

Код решения

Код решения: http://learn.javascript.ru/play/tutorial/widgets/tooltip-fixed/index.html.

Решение задачи: Подсказка, следующая за курсором

События

Три события имеют отношение к подсказке:

1. При `mouseover` — показать.
2. При `mousemove` — показать на новом месте.
3. При `mouseout` — спрятать.

…Но при заходе на элемент происходят сразу оба события: `mouseover` и `mousemove` на нём же. Зачем вызывать код для показа два раза? Можно оставить его только в `mousemove`.

Стиль элемента подсказки:

```
.tooltip {
  position:absolute;
  z-index:100; /* подсказка должна перекрывать другие элементы */
  ...
}
```

Решение

Решение: http://learn.javascript.ru/play/tutorial/widgets/tooltip-moving/index.html

В нём есть две тонкости.

1. Для того, чтобы можно было получить высоту и ширину, подсказку надо сначала показать.
2. При показе — важно, чтобы подсказка не оказалась *под* курсором. Если такое произойдет, то дальнейшие события `mousemove` станут происходить *уже на подсказке*, а не на элементе, и их обработка сломается.

Решение задачи: Менять размер картинки при помощи мыши

Решение, шаг 1

Для захвата проще всего создать дополнительные `DIV`'ы и отпозиционировать их сбоку и справа-снизу `IMG`. При нажатии на них начинать смену размера.

CSS-структура:

```
<div class="resize-wrapper" style="width: 503px; height: 285px;">
  <img src="heroes.jpg" id="heroes" style="width:500px;height:282px">
  <div class="resize-handle-s"></div>
  <div class="resize-handle-e"></div>
  <div class="resize-handle-se"></div>
</div>
```

Внешний `DIV` подстраивается под размер картинки и имеет `position:relative`. Внутри него расположены абсолютно позиционированные «ручки» для захвата.

Стиль:

```css
.resize-wrapper {
  position: relative;
}
.resize-wrapper img {
  /* img при таком DOCTYPE в некоторых браузерах имеет display:inline, в некоторых display:block
     если оставить inline, то под img браузер оставит пустое место для "хвостов" букв
  */
  display: block;
}

.resize-handle-se { /* правый-нижний угол */
  position: absolute;
  bottom: 0;
  right: 0;
  width: 16px;
  height: 16px;
  background: url(handle-se.png) no-repeat;
  cursor: se-resize;
}

.resize-handle-s { /* нижняя "рамка", за которую можно потащить */
  position: absolute;
  bottom: 0;
  height: 3px;
  width:100%;
  background: gray;
  cursor: s-resize;
}

.resize-handle-e { /* правая "рамка", за которую можно потащить */
  position: absolute;
  right: 0;
  top: 0;
  width: 3px;
  height:100%;
  background: gray;
  cursor: e-resize;
}
```

Для обозначения низа используется буква «s» (south, «юг» по-английски), обозначения правой стороны — буква «e» (east, «восток» по-английски).

Использование сторон света для обозначения направления можно часто встретить в скриптах.

Решение, шаг 2

Алгоритм:

1. При инициализации IMG оборачивается во внешнюю обертку и обкладывается `DIV`'ами, на которых ловим `mousedown`. Обертка нужна, чтобы абсолютно позиционировать `DIV`'ы внутри неё под/сбоку изображения.
2. При наступлении `mousedown`, начинаем ловить `document.mousemove` и подгонять картинку под размер, а обертку — под картинку.

   Желательно заодно отменить браузерное выделение и Drag'n'Drop, возвратив `false` из собработчиков событий `mousedown` и `dragstart`.

3. При наступлении `mouseup` — конец.

Решение, шаг 3

Решение: http://learn.javascript.ru/play/tutorial/widgets/resizeable/index.html

Решение задачи: Выбор фрагмента картинки мышью

HTML/CSS

Область выделения можно оформить как `DIV`, серого цвета, полупрозрачный, с рамкой:

```
.crop-area {
  position: absolute;
  border: 1px dashed black;
  background: #F5DEB3;
  opacity: 0.3;
  filter:alpha(opacity=30); /* IE opacity */
}
```

Решение

Решение: http://learn.javascript.ru/play/tutorial/widgets/croppable/index.html

Обратите внимание: обработчики `mousemove/mouseup` ставятся на `document`, не на элемент.

Это для того, чтобы посетитель мог начать выделение на элементе, а продолжить и завершить его, двигая зажатой мышкой снаружи, вне его границ.

Решение задачи: Меню с анимированным раскрытием

Решение:

Сладости (наведи курсор)!

.

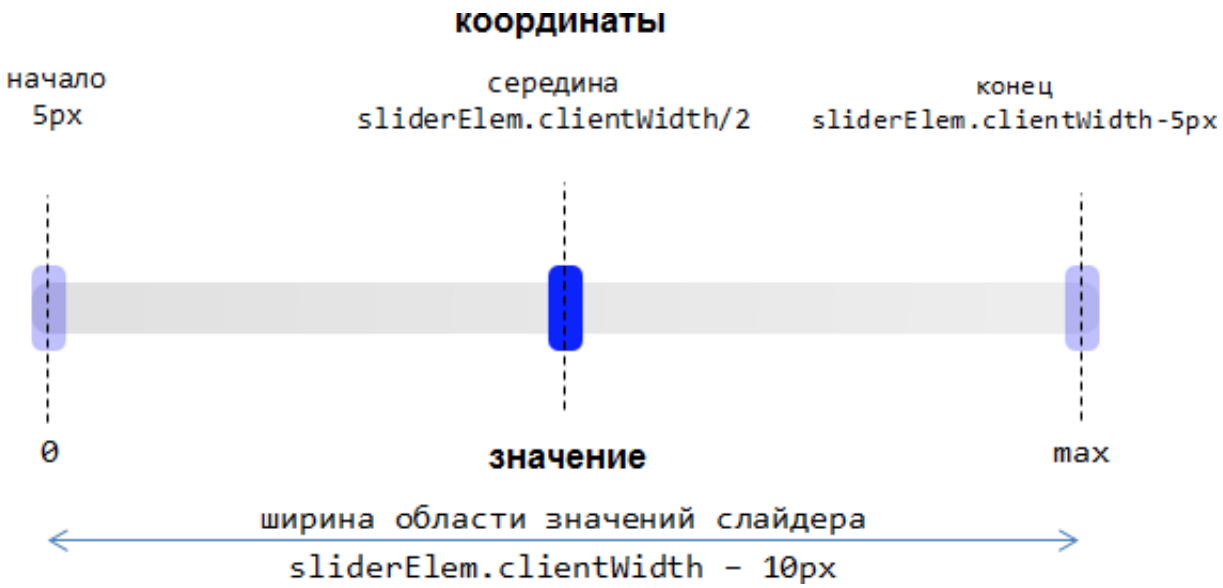
Решение задачи: Слайдер с событиями

Сопоставление значения и слайдера

Как сопоставить позицию слайдера и значение?

Для этого посмотрим крайние значения слайдера. Допустим, размер бегунка `10px`.

Раз центр соответствует значению, то крайнее левое значение будет соответствовать центру на `5px`, а крайнее правой — центру на `5px` от правой границы:



Соответственно, ширина области изменения будет `sliderElem.clientWidth - thumbElem.clientWidth`. Далее её можно уже поделить на части, количество пикселей на значение будет:

```
pixelsPerValue = (sliderElem.clientWidth-thumbElem.clientWidth) / max;
```

Может получиться так, что это значение будет дробным, меньше единицы. Например, если `max = 1000`, а ширина слайдера `110` (пробег 100), то будет `0.1` пикселя на значение.

Используя `pixelsPerValue` мы сможем переводить позицию бегунка в значение и обратно.

Крайнее левое значение `thumbElem.style.left` равно нулю, крайнее правой — как раз ширине доступной области `sliderElem.clientWidth - thumbElem.clientWidth`. Поэтому можно получив значение, поделив его на `pixelsPerValue`:

```
value = Math.round( newLeft / pixelsPerValue);
```

Полное решение

Решение: http://learn.javascript.ru/play/tutorial/widgets/slider-events/index.html.

Решение задачи: Календарь

http://learn.javascript.ru/play/tutorial/widgets/calendar/index.html.

Решение задачи: Автокомплит

http://learn.javascript.ru/play/tutorial/widgets/autocomplete/index.html

Решение задачи: Вложенное меню с выпаданием по клику

Решение: http://learn.javascript.ru/play/tutorial/widgets/menu-nested/index.html

Решение задачи: Шаблон для таблицы с пользователями

http://learn.javascript.ru/play/tutorial/widgets/template-grid/index.html

Решение задачи: Меню с шаблоном

http://learn.javascript.ru/play/tutorial/widgets/template-menu/index.html

Решение задачи: Меню с шаблоном списка и элемента

http://learn.javascript.ru/play/tutorial/widgets/template-menu-additem/index.html

Решение задачи: Селектор даты

http://learn.javascript.ru/play/tutorial/widgets/dateselector/index.html.

Решение задачи: Переносимые окна

Подсказки

- Так как высота и ширина окна известны, вёрстка внутри может содержать точные пиксельные размеры.
- При обработке события `document.onmousemove`, мы вычисляем новые координаты `left/top` и смотрим, вылезает ли окно за границы. Если да — меняем `left/top` на максимально возможные, чтобы не вылезало.
- На форме вешаем обработчик `onsubmit`, т.к. иначе Enter в поле отправить её на сервер.

Решение

http://learn.javascript.ru/play/tutorial/widgets/window/index.html

Решение задачи: Менеджер окон

http://learn.javascript.ru/play/tutorial/widgets/window-manager/index.html
Другая реализация похожего функционала с лучшей архитектурой: http://learn.javascript.ru/play/tutorial/widgets/window-manager-2/index.html

Решение задачи: Двойной календарь со стрелками

Решение: http://learn.javascript.ru/play/tutorial/widgets/datepicker/index.html.

В нём присутствуют файлы из задачи про календарь [24], т.к. он используется в качестве компонента.

Решение задачи: Дерево с чекбоксами

Получение данных

Данные регионов можно хранить в разном виде.

Наиболее естественное представление дерева — в виде вложенного объекта: свойство `children` содержит поддеревья.

Выглядит это так:

```
var regions = [
  {
    title: 'Россия',
    id: 1,
    children: [
      {
        title: 'Центр',
        id: 2,
        children: [ ... поддеревья ... ]
      },
      ...
    }
  }
  ...
]
```

У такого вложенного объекта есть важный недостаток: сложно перейти напрямую к узлу по ID. Нужно «прыгать» по дереву.

Поэтому может быть более удобен другой вариант:

```
var regions = [
  {
    title: 'Россия',
    id: 1,
    children: [ 2 ]
  },
  {
    title: 'Центр',
    id: 2,
    children: [ ... ]
  },
  ...
]
```

..То есть, массив содержит все узлы дерева, и каждый узел хранит в `children` список `id` детей.

Но и это не совсем удобно. Ведь хочется по ID получить данные. Значит, нужно хранить не массив, а объект вида `id => { title: .., id: .., children: [... ] }`.

- Свойство `children` может отсутствовать, если детей нет.
- `id` есть и в ключе объекта и в данных узла. Так удобнее.
- Список внешних узлов дерева содержится в корневом узле без имени, с `id = 0`.

Выберите наиболее симпатичную структуру и получите её из исходного дерева http://learn.javascript.ru/play/tutorial/widgets/checkbox-tree-src/index.html.

Данные

Скрипт для получения данных в последнем формате, описанном выше: fetch.js [25].

Результат в файле (после `JSON.stringify`): regions.js [26].

Исправления

Желательно сделать следующие исправления:

- Переделать вёрстку. В частности, заменить текстовые + - на оформление при помощи CSS.
- Оформить всё в виде виджета с шаблоном.
- Использовать ленивый рендеринг! Не рисовать всё дерево сразу (зачем рисовать то, чего не видно?), а дорисовывать при открытии.

Решение

http://learn.javascript.ru/play/tutorial/widgets/checkbox-tree/index.html

Ссылки

1. JQuery. Подробное руководство по продвинутому JavaScript http://www.ozon.ru/context/detail/id/6277333/
2. JQuery.on http://api.jquery.com/on/
3. Процедурный подход http://ru.wikipedia.org/wiki/Процедурное_программирование
4. Объектно-ориентированное программирование http://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование
5. Math https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Math
6. Инкапсуляция http://ru.wikipedia.org/wiki/Инкапсуляция_(программирование)
7. Дескриптор свойства, геттеры и сеттеры http://learn.javascript.ru/descriptors-getters-setters
8. Специальный синтаксис http://learn.javascript.ru/descriptors-getters-setters
9. Выпадающее меню http://learn.javascript.ru/task/vypadaushee-menu
10. Trigger http://api.jquery.com/trigger/
11. Trigger http://api.jquery.com/trigger/
12. Джоном Ресигом http://ejohn.org/blog/javascript-micro-templating/
13. LoDash https://github.com/bestiejs/lodash
14. LoDash https://github.com/bestiejs/lodash
15. Меню с шаблоном списка и элемента http://learn.javascript.ru/task/menu-s-shablonom-spiska-i-elementa
16. Knockout.JS http://knockoutjs.com
17. Angular.JS http://angularjs.org
18. V8 http://code.google.com/p/v8/
19. Mustache http://mustache.github.com/
20. RequireJS http://requirejs.com
21. LoDash https://github.com/bestiejs/lodash
22. V8 http://code.google.com/p/v8/
23. Mustache http://mustache.github.com/
24. Задачи про календарь http://learn.javascript.ru/task/kalendar
25. Fetch.js http://learn.javascript.ru/files/tutorial/widgets/checkbox-tree/fetch.js
26. Regions.js http://learn.javascript.ru/files/tutorial/widgets/checkbox-tree/regions.js