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: Разные темы

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

Содержание

- Таймеры
  - setTimeout и setInterval
    - setTimeout
      - Параметры для функции и контекст
      - Отмена исполнения
    - setInterval
    - Очередь и наложение вызовов в setInterval
    - Повторение вложенным setTimeout
    - Минимальная задержка таймера
    - Реальная частота срабатывания
    - Разбивка долгих скриптов
    - Трюк setTimeout(func, 0)
    - Итого
  - Клонировать setTimeout и setInterval
    - setTimeout
      - Отмена исполнения clearTimeout
    - setInterval
    - Рекурсивный setTimeout при помощи NFE
    - Тонкости тайминга с setInterval
    - Повторение вложенным setTimeout
    - Минимальная задержка таймера
    - Реальная частота срабатывания
    - Разбивка долгих скриптов
    - Трюк setTimeout(func, 0)
    - Итого
  - setImmediate
    - Метод setImmediate(func)
    - Тест производительности

- Привязка контекста
  - Привязка функции к объекту и карринг: "bind/bindLate"
    - Привязка через замыкание
    - Современный метод bind
      - bind с аргументами
    - Кросс-браузерная эмуляция bind
      - Вариант bind с каррингом
      - Вариант bind для методов
    - Итого
  - Позднее связывание "bindLate"
    - Раннее связывание
    - Позднее связывание
    - Привязка метода, которого нет
    - Итого
- Статические и фабричные методы объектов
  - Статические свойства
  - Статические методы
  - Пример: сравнение объектов
  - Фабричные методы
  - Итого
- Массив: Перебирающие методы
  - forEach
  - filter
  - map
  - every/some
  - reduce/reduceRight
- Запуск кода из строки: eval
  - Пример eval
  - eval и локальные переменные
  - Запуск кода в глобальном контексте
  - Взаимодействие с внешним кодом, new Function
  - Итого
- Перехват ошибок, "try..catch"
  - Типы ошибок
    - Конструкция try..catch
    - Объект ошибки
    - Секция finally
  - Генерация своих ошибок
    - Пример: проверка значений
    - Генерация ошибки: throw
    - Вложенные вызовы и try..catch
  - Итого
- Формат JSON
  - Формат JSON
  - JSON.stringify и JSON.parse
  - Детали JSON.stringify
    - Сериализация объектов, toJSON
    - Исключение свойств
    - Красивое форматирование

Умный разбор: JSON.parse(str, reviver)
IE<8: разбор JSON с eval
Безопасность, проверка JSON
Тест: повторение тонких мест
Решения задач

Таймеры

setTimeout и setInterval

Почти все реализации JavaScript имеют внутренний таймер-планировщик, который позволяет задавать вызов функции через заданный период времени.

В частности, эта возможность поддерживается в браузерах и в сервере Node.JS.

setTimeout

Синтаксис:

```
var timerId = setTimeout(func/code, delay[, arg1, arg2...])
```

Параметры:

`func/code`
: Функция или строка кода для исполнения.
Строка поддерживается для совместимости, использовать её не рекомендуется.

`delay`
: Задержка в милисекундах, 1000 милисекунд равны 1 секунде.

`arg1`, `arg2`...
: Аргументы, которые нужно передать функции. Не поддерживаются в IE9-.

Исполнение функции произойдёт спустя время, указанное в параметре `delay`.

Например, следующий код вызовет `alert('Привет')` через одну секунду:

```
function func() {
  alert('Привет');
}
setTimeout(func, 1000);
```

Если первый аргумент является строкой, то интерпретатор создаёт анонимную функцию из этой строки.

То есть такая запись работает точно так же:

```
setTimeout("alert('Привет')", 1000);
```

Использование строк не рекомендуется, так как они могут вызвать проблемы при минимизации кода, и, вообще, сама возможность использовать строку сохраняется лишь для совместимости.

Вместо них используйте анонимные функции:

```
setTimeout(function() { alert('Привет') }, 1000);
```

Параметры для функции и контекст

Во всех современных браузерах, с учетом IE10, `setTimeout` позволяет указать параметры функции.

Пример ниже выведет `"Привет, я Вася"` везде, кроме IE9-:

```
function sayHi(who) {
  alert("Привет, я " + who);
}

setTimeout(sayHi, 1000, "Вася");
```

…Однако, в большинстве случаев нам нужна поддержка старого IE, а он не позволяет указывать аргументы. Поэтому, для того, чтобы их передать, оборачивают вызов в анонимную функцию:

```
function sayHi(who) {
  alert("Привет, я " + who);
}

setTimeout(function() { sayHi('Вася') }, 1000);
```

Вызов через `setTimeout` не передаёт контекст `this`.

В частности, вызов метода объекта через `setTimeout` сработает в глобальном контексте. Это может привести к некорректным результатам.

Например, вызовем `user.sayHi()` через одну секунду:

```
function User(id) {
  this.id = id;

  this.sayHi = function() {
    alert(this.id);
  };
}

var user = new User(12345);

setTimeout(user.sayHi, 1000); // ожидается 12345, но выведет "undefined"
```

Так как `setTimeout` запустит функцию `user.sayHi` в глобальном контексте, она не будет иметь доступ к объекту через `this`.

Иначе говоря, эти два вызова `setTimeout` делают одно и то же:

```
// (1) одна строка
setTimeout(user.sayHi, 1000);

// (2) то же самое в две строки
var func = user.sayHi;
setTimeout(func, 1000);
```

К счастью, эта проблема также легко решается созданием промежуточной функции:

```
function User(id) {
  this.id = id;

  this.sayHi = function() {
    alert(this.id);
  };
}

var user = new User(12345);

setTimeout(function() {
  user.sayHi();
}, 1000);
```

Функция-обёртка используется, чтобы кросс-браузерно передать аргументы и сохранить контекст выполнения.

В следующих главах мы разберём дополнительные способы привязки функции к объекту.

Отмена исполнения

Функция `setTimeout` возвращает идентификатор `timerId`, который можно использовать для отмены действия.

Синтаксис: `clearTimeout(timerId)`.

В следующем примере мы ставим таймаут, а затем удаляем (передумали). В результате ничего не происходит.

```
var timerId = setTimeout(function() { alert(1) }, 1000);

clearTimeout(timerId);
```

setInterval

Метод `setInterval` имеет синтаксис, аналогичный `setTimeout`.

```
var timerId = setInterval(func/code, delay[, arg1, arg2...])
```

Смысл аргументов — тот же самый. Но, в отличие от `setTimeout`, он запускает выполнение функции не один раз, а регулярно повторяет её через указанный интервал времени. Остановить исполнение можно вызовом `clearInterval(timerId)`.

Следующий пример при запуске станет выводить сообщение каждые две секунды, пока вы не нажмете на кнопку «Стоп»:

```
<input type="button" onclick="clearInterval(timer)" value="Стоп">

<script>
  var i = 1;
  var timer = setInterval(function() { alert(i++) }, 2000);
</script>
```

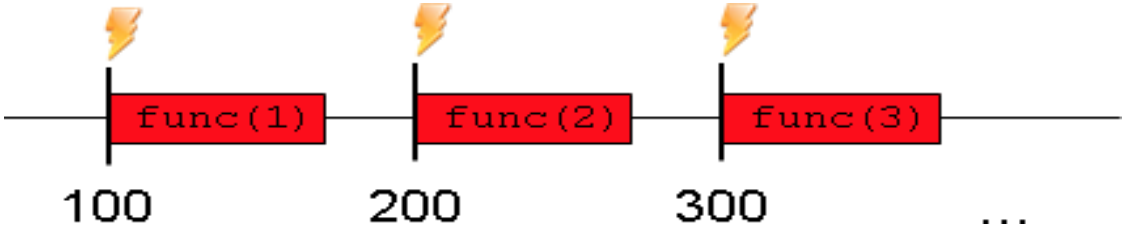
Очередь и наложение вызовов в setInterval

Вызов `setInterval(функция, задержка)` ставит функцию на исполнение через указанный интервал времени. Но здесь есть тонкость.

На самом деле пауза между вызовами меньше, чем указанный интервал.

Для примера, возьмем `setInterval(function() { func(i++) }, 100)`. Она выполняет `func` каждые 100 мс, каждый раз увеличивая значение счетчика.

На картинке ниже, красный блок - это время исполнения `func`. Время между блоком — это время между запусками функции, и оно меньше, чем установленная задержка!



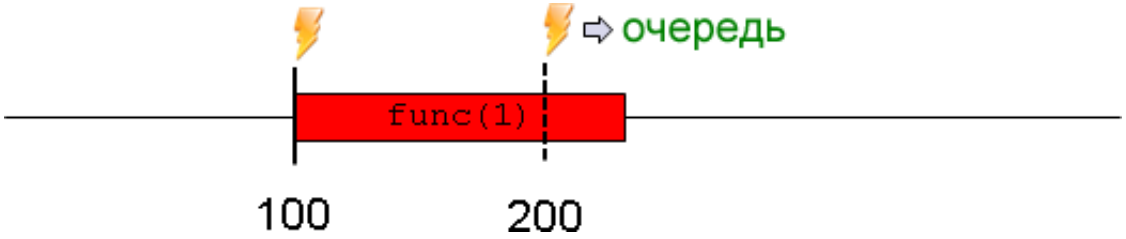
То есть, браузер инициирует запуск функции аккуратно каждые `100мс`, без учета времени выполнения самой функции.

Бывает, что исполнение функции занимает больше времени, чем задержка. Например, функция сложная, а задержка маленькая. Или функция содержит операторы `alert/confirm/prompt`, которые блокируют поток выполнения. В этом случае начинаются интересные вещи

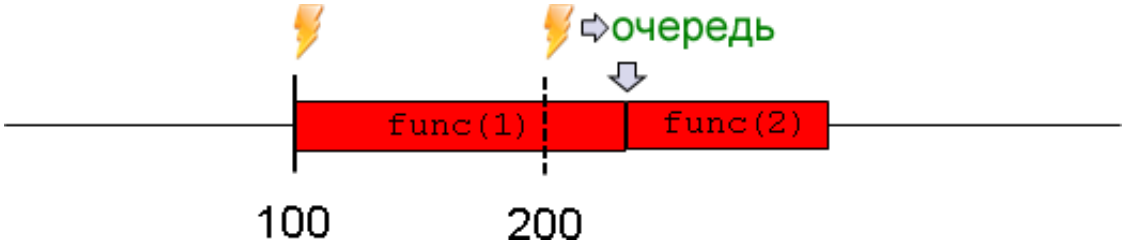
Если запуск функции невозможен, потому что браузер занят — она становится в очередь и выполнится, как только браузер освободится.

Изображение ниже иллюстрирует происходящее для функции, которая долго исполняется.

Вызов функции, инициированный `setInterval`, добавляется в очередь и незамедлительно происходит, когда это становится возможным:



Второй запуск функции происходит сразу же после окончания первого:

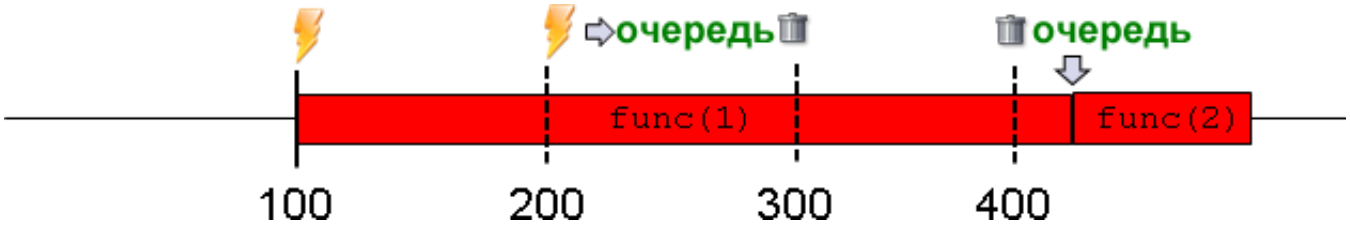


Больше одного раза в очередь выполнение не ставится.

Если выполнение функции занимает больше времени, чем несколько запланированных исполнений, то в очереди она всё равно будет стоять один раз. Так что «накопления» запусков не происходит.

На изображении ниже `setInterval` пытается выполнить функцию в 200 мс и ставит вызов в очередь. В 300 мс и 400 мс таймер

пробуждается снова, но ничего не просходит.



Давайте посмотрим на примере, как это работает.

Модальные окна в Safari/Chrome блокируют таймер

Внутренний таймер в браузерах Safari/Chrome во время показа `alert/confirm/prompt` не «тикает». Если до исполнения оставалось 3 секунды, то даже при показе `alert` на протяжении минуты — задержка остаётся 3 секунды.

Поэтому пример ниже не воспроизводится в этих браузерах. В других браузерах всё в порядке.

1. Запустите пример ниже в любом браузере, кроме Chrome/Safari и дождитесь всплывающего окна. Обратите внимание, что это `alert`. Пока модальное окошко отображается, исполнение JavaScript блокируется. Подождите немного и нажмите OK.
2. Вы должны увидеть, что второй запуск будет тут же, а третий - через небольшое время от второго, меньшее чем 2000 мс.
3. Чтобы остановить повторение, нажмите кнопку `Стоп`.

```
<input type="button" onclick="clearInterval(timer)" value="Стоп">

<script>
  var i = 1;
  var timer = setInterval(function() { alert(i++) }, 2000);
</script>
```

Происходит следующее.

1. Браузер выполняет функцию каждые 2 секунды
2. **Когда всплывает окно `alert`** — исполнение блокируется и остается заблокированным всё время, пока `alert` отображается.
3. Если вы ждете достаточно долго, то внутренние часики-то идут. Браузер ставит следующее исполнение в очередь, один раз (в Chrome/Safari внутренний таймер не идёт! это ошибка в браузере).
4. **Когда вы нажмёте OK** - моментально вызывается исполнение, которое было в очереди.
5. Следующее исполнение вызовется с меньшей задержкой, чем указано. Так происходит потому, что планировщик просыпается каждые 2000мс. И если alert был закрыт в момент времени, соответствующий 3500мс от начала, то следующее исполнение назначено на 4000 мс, т.е. произойдёт через 500мс.

Вызов `setInterval(функция, задержка)` не гарантирует реальной задержки *между* исполнениями.

Бывают случаи, когда реальная задержка больше или меньше заданной. Вообще, не факт, что будет хоть какая-то задержка.

Повторение вложенным `setTimeout`

В случаях, когда нужно не просто регулярное повторение, а обязательна задержка между запусками, используется повторная установка `setTimeout` при каждом выполнении функции.

Ниже — пример, который выдает `alert` с интервалами 2 секунды *между ними*.

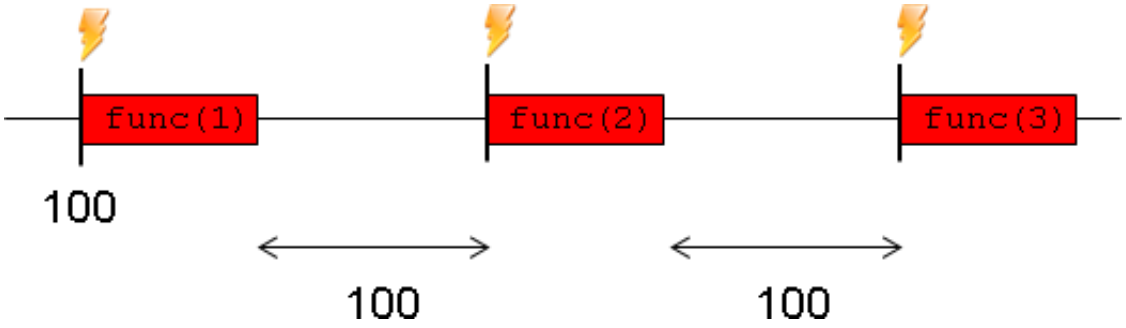
```
<input type="button" onclick="clearTimeout(timer)" value="Стоп">

<script>
  var i = 1;

  var timer = setTimeout(function run() {
    alert(i++);
    timer = setTimeout(run, 2000);
  }, 2000);

</script>
```

На временной линии выполнения будут фиксированные задержки между запусками. Иллюстрация для задержки 100мс:



Минимальная задержка таймера

У браузерного таймера есть минимальная возможная задержка. Она меняется от примерно нуля до 4мс в современных браузерах. В более старых она может быть больше и достигать 15мс.

По стандарту [1] , минимальная задержка составляет 4мс. Так что нет разницы между `setTimeout(..,1)` и `setTimeout(..,4)`.

Посмотреть минимальное разрешение «вживую» можно на следующем примере.

В примере ниже находятся `DIV'ы`, каждый удлиняется вызовом `setInterval` с указанной в нём задержкой — от 0мс (сверху) до 20мс (внизу).

Запустите его в различных браузерах, в частности, в Chrome и Firefox. Вы наверняка заметите, что несколько первых `DIV'ов` анимируются с одинаковой скоростью. Это как раз потому, что слишком маленькие задержки таймер не различает.

Старт Стоп
0
2
4
6
8
10
12
14
16
18
20

В поведении `setTimeout` и `setInterval` с нулевой задержкой есть браузерные особенности.

- В Opera, `setTimeout(.., 0)` — то же самое, что `setTimeout(.., 4)`. Оно выполняется *реже*, чем `setTimeout(.. ,2)`. Это особенность данного браузера.
- В Internet Explorer, нулевая задержка `setInterval(.., 0)` не сработает. Это касается именно `setInterval`, т.е. `setTimeout(.., 0)` работает нормально.

Пример ниже реализует такую же анимацию, но через `setTimeout`. Если посмотреть его в различных браузерах, то можно заметить отличия от `setInterval`.

Старт Стоп
0
2
4
6
8
10
12
14
16
18
20

Реальная частота срабатывания

Срабатывание может быть и гораздо реже

В ряде случаев задержка может быть не 4мс, а 30мс или даже 1000мс.

- Большинство браузеров (десктопных в первую очередь) продолжают выполнять `setTimeout/setInterval`, даже если вкладка неактивна.

  При этом ряд из них (Chrome, FF, IE10) снижают минимальную частоту таймера, до 1 раза в секунду. Получается, что в «фоновой» вкладке будет срабатывать таймер, но редко.

- При работе от батареи, в ноутбуке — браузеры тоже могут снижать частоту, чтобы реже выполнять код и экономить заряд батареи. Особенно этим известен IE. Снижение может достигать нескольких раз, в зависимости от настроек.
- При слишком большой загрузке процессора JavaScript может не успевать обрабатывать таймеры вовремя. При этом некоторые запуски `setInterval` будут пропущены.

Вывод: на частоту 4мс стоит ориентироваться, но не стоит рассчитывать.

Посмотрим снижении частоты в действии на небольшом примере.

При клике на кнопку ниже запускается `setInterval(..., 90)`, который выводит список интервалов времени между 25 последними срабатываниями таймера. Запустите его. Перейдите на другую вкладку и вернитесь.

Запустить повтор с интервалом в 90 мс

Остановить повтор

Если ваш браузер увеличивает таймаут при фоновом выполнении вкладки, то вы увидите увеличенные интервалы, помеченные красным.

Кроме того, вы точно увидите, что таймер не является идеально точным 

Вывод интервалов в консоль

Код, который используется в примере выше и считает интервалы времени между вызовами, выглядит примерно так:

```
var timeMark = new Date;
setTimeout(function go() {
  var diff = new Date - timeMark;

  // вывести очередную задержку в консоль вместо страницы
  console.log(diff);

  // запомним время в самом конце,
  // чтобы измерить задержку именно между вызовами
  timeMark = new Date;

  setTimeout(go, 100);
}, 100);
```

Разбивка долгих скриптов

Нулевой или небольшой таймаут также используют, чтобы разорвать поток выполнения «тяжелых» скриптов.

Например, скрипт для подсветки синтаксиса должен проанализировать код, создать много цветных элементов для подсветки и добавить их в документ — на большом файле это займёт много времени.

Браузер сначала будет есть 100% процессора, а затем может выдать сообщение о том, что скрипт выполняется слишком долго.

Для того, чтобы этого избежать, сложная задача разбивается на части, выполнение каждой части запускается через мини-интервал после предыдущей, чтобы дать браузеру время. Например, планируется подсветка 20 строк каждые 10мс.

Трюк `setTimeout(func, 0)`

Этот трюк достоин войти в анналы JavaScript-хаков.

Функцию оборачивают в `setTimeout(func, 0)`, если хотят запустить ее после окончания текущего скрипта.

Дело в том, что `setTimeout` никогда не выполняет функцию сразу. Он лишь планирует ее выполнение. Но интерпретатор JavaScript начнёт выполнять запланированные функции лишь после выполнения текущего скрипта.

По стандарту, `setTimeout` в любом случае не может выполнить функцию с задержкой `0`. Как мы говорили раньше, обычно задержка составит 4мс. Но главное здесь именно то, что выполнение в любом случае будет после выполнения текущего кода.

Например:

```
01 var result;
02
03 function showResult() {
04   alert(result);
05 }
06
07 setTimeout(showResult, 0);
08
09 result = 2*2;
10
11 // выведет 4
```

Позже, в главе Управление порядком обработки, setTimeout(…0) [2], мы рассмотрим различные применения этого трюка при работе с событиями.

Итого

Методы `setInterval(func, delay)` и `setTimeout(func, delay)` позволяют запускать `func` регулярно/один раз через `delay` миллисекунд.

Оба метода возвращают идентификатор таймера. Его используют для остановки выполнения вызовом `clearInterval/clearTimeout`.

Особенности

	`setInterval`	`setTimeout`
Тайминг	Идет вызов строго по таймеру. Если интерпретатор занят — один вызов становится в очередь. Время выполнения функции не учитывается, поэтому	Рекурсивный вызов `setTimeout` используется вместо `setInterval` там, где нужна фиксированная пауза

	промежуток времени от окончания одного запуска до начала другого может быть различным.	между выполнениями.
Задержка	Минимальная задержка: 4мс.	Минимальная задержка: 4мс.
	Минимальная задержка для этих методов в современных браузерах различна и колеблется от примерно нуля до 4мс. В старых браузерах она может доходить до 15мс.	
Браузерные особенности	В IE не работает задержка `0`.	В Opera нулевая задержка эквивалентна 4мс, остальные задержки обрабатываются точно, в том числе нестандартные 1мс, 2мс и 3мс.

Клонировать setTimeout и setInterval

Почти все реализации JavaScript имеют внутренний таймер-планировщик, который позволяет задавать вызов функции через заданный период времени.

В частности, эта возможность поддерживается в браузерах и в сервере Node.JS.

setTimeout

Синтаксис:

```
var timerId = setTimeout(func/code, delay[, arg1, arg2...])
```

Параметры:

`func/code`
: Функция или строка кода для исполнения.
Строка поддерживается для совместимости, использовать её не рекомендуется.

`delay`
: Задержка в милисекундах, 1000 милисекунд равны 1 секунде.

`arg1, arg2...`
: Аргументы, которые нужно передать функции. Не поддерживаются в IE9-.

Исполнение функции произойдёт спустя время, указанное в параметре `delay`.

Например, следующий код вызовет `alert('Привет')` через одну секунду:

```
function func() {
  alert('Привет');
}
setTimeout(func, 1000);
```

Если первый аргумент является строкой, то интерпретатор создаёт анонимную функцию из этой строки.

То есть такая запись работает точно так же:

```
setTimeout("alert('Привет')", 1000);
```

Использование строк не рекомендуется, так как они могут вызвать проблемы при минимизации кода, и, вообще, сама возможность использовать строку сохраняется лишь для совместимости.

Вместо них используйте анонимные функции:

```
setTimeout(function() { alert('Привет') }, 1000);
```

Отмена исполнения `clearTimeout`

Функция `setTimeout` возвращает идентификатор `timerId`, который можно использовать для отмены действия.

Синтаксис: `clearTimeout(timerId)`.

В следующем примере мы ставим таймаут, а затем удаляем (передумали). В результате ничего не происходит.

```
var timerId = setTimeout(function() { alert(1) }, 1000);

clearTimeout(timerId);
```

`setInterval`

Метод `setInterval` имеет синтаксис, аналогичный `setTimeout`.

```
var timerId = setInterval(func/code, delay[, arg1, arg2...])
```

Смысл аргументов — тот же самый. Но, в отличие от `setTimeout`, он запускает выполнение функции не один раз, а регулярно повторяет её через указанный интервал времени. Остановить исполнение можно вызовом `clearInterval(timerId)`.

Следующий пример при запуске станет выводить сообщение каждые две секунды, пока не пройдёт 5 секунд:

```
var timerId = setInterval(function() {
  alert("тик");
}, 2000);

setTimeout(function() {
  clearInterval(timerId);
  alert('стоп');
}, 5000);
```

Модальные окна в Safari/Chrome/Opera блокируют таймер

Что будет, если долго не жать `OK` на появившемся `alert`?

Это зависит от браузера.

В браузерах Chrome, Opera и Safari внутренний таймер «замирает» во время показа `alert/confirm/prompt` и продолжит отсчёт с момента закрытия. Поэтому после закрытия `alert` в любом случае пройдёт две секунды до следующего тика.

А вот в IE и Firefox внутренний таймер продолжит идти, и, если долго не закрывать `alert`, то может подойти время следующего вызова. Но, так как браузер не может показать новый `alert`, пока открыт этот, он будет ждать нажатия `OK`, и потом покажет новый `alert` *сразу же*.

Рекурсивный `setTimeout` при помощи NFE

Бывает так, что функцию необходимо выполнять с определённым интервалом, но следующий интервал определяется исходя из её последнего результата.

Реальный пример — мы пытаемся подсоединиться к серверу получить данные, но это не получается, например потому что сервер перегружен (мало ли что, может атака на него).

Чтобы не грузить сервер ещё больше, мы сделаем следующий запрос данных через чуть больший интервал, потом, если проблема ещё есть — ещё через больший и так до максимального интервала, когда мы прекращаем автозапросы и предлагаем пользователю решить, что делать. Примерно так поступает, к примеру, Gmail [3].

Здесь, чтобы не отвлекаться на другие темы,

Например, будем выводить `alert` со случайным интервалом:

Например, мы пишем приложение, которое пытается раз в секунду присоединиться к серверу и получить с него данные. Это будет делать функция `connect()`

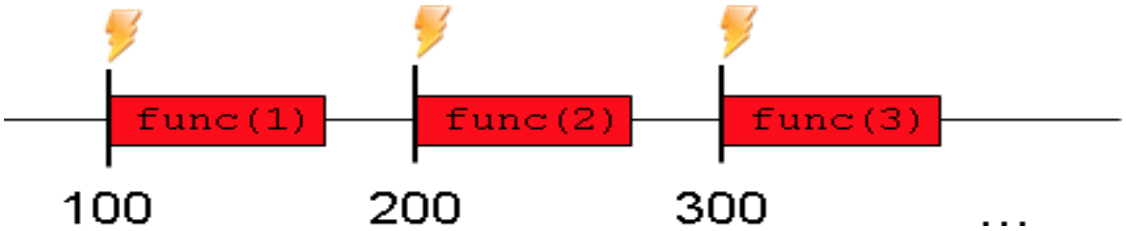
Тонкости тайминга с `setInterval`

Вызов `setInterval(функция, задержка)` ставит `функцию` на исполнение через указанный интервал времени. Но здесь есть тонкость.

На самом деле пауза между вызовами меньше, чем указанный интервал.

Для примера, возьмем `setInterval(function() { func(i++) }, 100)`. Она выполняет `func` каждые 100 мс, каждый раз увеличивая значение счетчика.

На картинке ниже, красный блок - это время исполнения `func`. Время между блоком — это время между запусками функции, и оно меньше, чем установленная задержка!



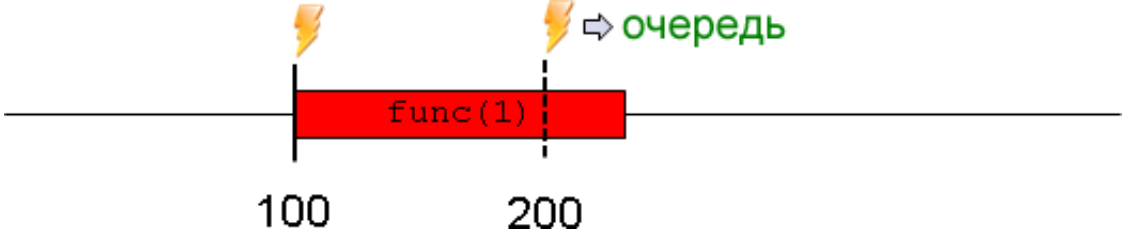
То есть, браузер инициирует запуск функции аккуратно каждые `100мс`, без учета времени выполнения самой функции.

Бывает, что исполнение функции занимает больше времени, чем задержка. Например, функция сложная, а задержка маленькая. Или функция содержит операторы `alert/confirm/prompt`, которые блокируют поток выполнения. В этом случае начинаются интересные вещи

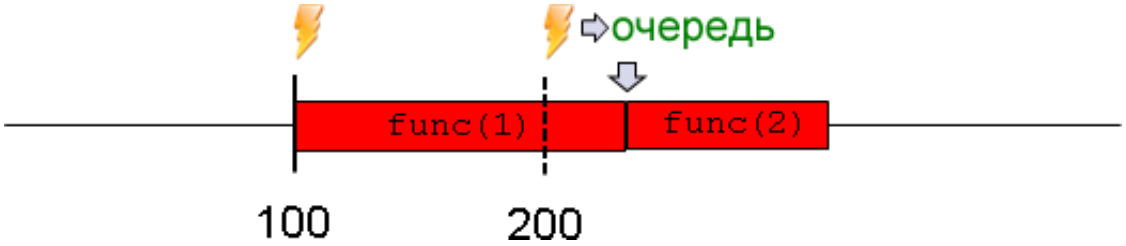
Если запуск функции невозможен, потому что браузер занят — она становится в очередь и выполнится, как только браузер освободится.

Изображение ниже иллюстрирует происходящее для функции, которая долго исполняется.

Вызов функции, инициированный `setInterval`, добавляется в очередь и незамедлительно происходит, когда это становится возможным:



Второй запуск функции происходит сразу же после окончания первого:



Больше одного раза в очередь выполнение не ставится.

Если выполнение функции занимает больше времени, чем несколько запланированных исполнений, то в очереди она всё равно будет стоять один раз. Так что «накопления» запусков не происходит.

На изображении ниже `setInterval` пытается выполнить функцию в 200 мс и ставит вызов в очередь. В 300 мс и 400 мс таймер пробуждается снова, но ничего не просходит.

Давайте посмотрим на примере, как это работает.

Модальные окна в Safari/Chrome/Opera блокируют таймер

Внутренний таймер в браузерах Safari/Chrome во время показа `alert/confirm/prompt` не «тикает». Если до исполнения оставалось 3 секунды, то даже при показе `alert` на протяжении минуты — задержка остаётся 3 секунды.

Поэтому пример ниже не воспроизводится в этих браузерах. В других браузерах всё в порядке.

1. Запустите пример ниже в любом браузере, кроме Chrome/Safari и дождитесь всплывающего окна. Обратите внимание, что это `alert`. Пока модальное окошко отображается, исполнение JavaScript блокируется. Подождите немного и нажмите OK.
2. Вы должны увидеть, что второй запуск будет тут же, а третий - через небольшое время от второго, меньшее чем 2000 мс.
3. Чтобы остановить повторение, нажмите кнопку `Стоп`.

```
<input type="button" onclick="clearInterval(timer)" value="Стоп">

<script>
  var i = 1;
  var timer = setInterval(function() { alert(i++) }, 2000);
</script>
```

Происходит следующее.

1. Браузер выполняет функцию каждые 2 секунды
2. **Когда всплывает окно `alert` —** исполнение блокируется и остается заблокированным всё время, пока `alert` отображается.
3. Если вы ждете достаточно долго, то внутренние часики-то идут. Браузер ставит следующее исполнение в очередь, один раз (в Chrome/Safari внутренний таймер не идёт! это ошибка в браузере).
4. **Когда вы нажмёте OK** - моментально вызывается исполнение, которое было в очереди.
5. Следующее исполнение вызовется с меньшей задержкой, чем указано. Так происходит потому, что планировщик просыпается каждые 2000мс. И если alert был закрыт в момент времени, соответствующий 3500мс от начала, то следующее исполнение назначено на 4000 мс, т.е. произойдёт через 500мс.

Вызов `setInterval(функция, задержка)` не гарантирует реальной задержки *между* исполнениями.

Бывают случаи, когда реальная задержка больше или меньше заданной. Вообще, не факт, что будет хоть какая-то задержка.

Повторение вложенным `setTimeout`

В случаях, когда нужно не просто регулярное повторение, а обязательна задержка между запусками, используется повторная установка `setTimeout` при каждом выполнении функции.

Ниже — пример, который выдает `alert` с интервалами 2 секунды *между ними*.

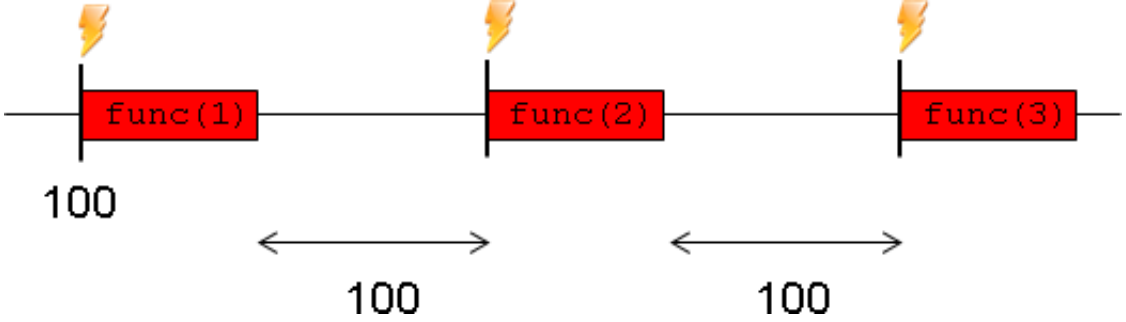
```
<input type="button" onclick="clearTimeout(timer)" value="Стоп">

<script>
  var i = 1;

  var timer = setTimeout(function run() {
    alert(i++);
    timer = setTimeout(run, 2000);
  }, 2000);

</script>
```

На временной линии выполнения будут фиксированные задержки между запусками. Иллюстрация для задержки 100мс:



Минимальная задержка таймера

У браузерного таймера есть минимальная возможная задержка. Она меняется от примерно нуля до 4мс в современных браузерах. В более старых она может быть больше и достигать 15мс.

По стандарту [4] , минимальная задержка составляет 4мс. Так что нет разницы между `setTimeout(..,1)` и `setTimeout(..,4)`.

Посмотреть минимальное разрешение «вживую» можно на следующем примере.

В примере ниже находятся `DIV`'ы, каждый удлиняется вызовом `setInterval` с указанной в нём задержкой — от 0мс (сверху) до 20мс (внизу).

Запустите его в различных браузерах, в частности, в Chrome и Firefox. Вы наверняка заметите, что несколько первых `DIV`'ов анимируются с одинаковой скоростью. Это как раз потому, что слишком маленькие задержки таймер не различает.

Старт Стоп
0
2
4
6
8
10
12
14
16
18
20

В поведении `setTimeout` и `setInterval` с нулевой задержкой есть браузерные особенности.

- В Opera, `setTimeout(.., 0)` — то же самое, что `setTimeout(.., 4)`. Оно выполняется *реже*, чем `setTimeout(.. ,2)`. Это особенность данного браузера.
- В Internet Explorer, нулевая задержка `setInterval(.., 0)` не сработает. Это касается именно `setInterval`, т.е. `setTimeout(.., 0)` работает нормально.

Пример ниже реализует такую же анимацию, но через `setTimeout`. Если посмотреть его в различных браузерах, то можно заметить отличия от `setInterval`.

Старт Стоп
0
2
4
6
8
10
12
14
16
18
20

Реальная частота срабатывания

Срабатывание может быть и гораздо реже

В ряде случаев задержка может быть не 4мс, а 30мс или даже 1000мс.

- Большинство браузеров (десктопных в первую очередь) продолжают выполнять `setTimeout/setInterval`, даже если вкладка неактивна.

  При этом ряд из них (Chrome, FF, IE10) снижают минимальную частоту таймера, до 1 раза в секунду. Получается, что в «фоновой» вкладке будет срабатывать таймер, но редко.
- При работе от батареи, в ноутбуке — браузеры тоже могут снижать частоту, чтобы реже выполнять код и экономить заряд батареи. Особенно этим известен IE. Снижение может достигать нескольких раз, в зависимости от настроек.
- При слишком большой загрузке процессора JavaScript может не успевать обрабатывать таймеры вовремя. При этом некоторые запуски `setInterval` будут пропущены.

Вывод: на частоту 4мс стоит ориентироваться, но не стоит рассчитывать.

Посмотрим снижении частоты в действии на небольшом примере.

При клике на кнопку ниже запускается `setInterval(..., 90)`, который выводит список интервалов времени между 25 последними срабатываниями таймера. Запустите его. Перейдите на другую вкладку и вернитесь.

Запустить повтор с интервалом в 90 мс

Остановить повтор

Если ваш браузер увеличивает таймаут при фоновом выполнении вкладки, то вы увидите увеличенные интервалы, помеченные красным.

Кроме того, вы точно увидите, что таймер не является идеально точным

Вывод интервалов в консоль

Код, который используется в примере выше и считает интервалы времени между вызовами, выглядит примерно так:

```
var timeMark = new Date;
setTimeout(function go() {
  var diff = new Date - timeMark;

  // вывести очередную задержку в консоль вместо страницы
  console.log(diff);

  // запомним время в самом конце,
  // чтобы измерить задержку именно между вызовами
  timeMark = new Date;

  setTimeout(go, 100);
}, 100);
```

Разбивка долгих скриптов

Нулевой или небольшой таймаут также используют, чтобы разорвать поток выполнения «тяжелых» скриптов.

Например, скрипт для подсветки синтаксиса должен проанализировать код, создать много цветных элементов для подсветки и добавить их в документ — на большом файле это займёт много времени.

Браузер сначала будет есть 100% процессора, а затем может выдать сообщение о том, что скрипт выполняется слишком долго.

Для того, чтобы этого избежать, сложная задача разбивается на части, выполнение каждой части запускается через мини-интервал после предыдущей, чтобы дать браузеру время. Например, планируется подсветка 20 строк каждые 10мс.

Трюк `setTimeout(func, 0)`

Этот трюк достоин войти в анналы JavaScript-хаков.

Функцию оборачивают в `setTimeout(func, 0)`, если хотят запустить ее после окончания текущего скрипта.

Дело в том, что `setTimeout` никогда не выполняет функцию сразу. Он лишь планирует ее выполнение. Но интерпретатор JavaScript начнёт выполнять запланированные функции лишь после выполнения текущего скрипта.

По стандарту, `setTimeout` в любом случае не может выполнить функцию с задержкой `0`. Как мы говорили раньше, обычно задержка составит 4мс. Но главное здесь именно то, что выполнение в любом случае будет после выполнения текущего кода.

Например:

```
var result;

function showResult() {
  alert(result);
}

setTimeout(showResult, 0);

result = 2*2;

// выведет 4
```

Позже, в главе Управление порядком обработки, setTimeout(…0) [5], мы рассмотрим различные применения этого трюка при работе с событиями.

Итого

Методы `setInterval(func, delay)` и `setTimeout(func, delay)` позволяют запускать `func` регулярно/один раз через `delay` миллисекунд.

Оба метода возвращают идентификатор таймера. Его используют для остановки выполнения вызовом `clearInterval/clearTimeout`.

Особенности

	`setInterval`	`setTimeout`
Тайминг	Идет вызов строго по таймеру. Если интерпретатор занят — один вызов становится в очередь. Время выполнения функции не учитывается, поэтому промежуток времени от окончания одного запуска до начала другого может быть различным.	Рекурсивный вызов `setTimeout` используется вместо `setInterval` там, где нужна фиксированная пауза между выполнениями.
Задержка	Минимальная задержка: 4мс.	Минимальная задержка: 4мс.
	Минимальная задержка для этих методов в современных браузерах различна и колеблется от примерно нуля до 4мс. В старых браузерах она может доходить до 15мс.	
Браузерные особенности	В IE не работает задержка `0`.	В Opera нулевая задержка эквивалентна 4мс, остальные задержки обрабатываются точно, в том числе нестандартные 1мс, 2мс и 3мс.

setImmediate

Функция, отложенная через `setTimeout(..0)` выполнится не ранее следующего «тика» таймера, минимальная частота которого может составлять от 4 до 1000мс. И, конечно же, это произойдёт после того, как все текущие изменения будут перерисованы.

Но нужна ли нам эта дополнительная задержка? Как правило, используя `setTimeout(func, 0)`, мы хотим перенести выполнение `func` на «ближайшее время после текущего кода», и какая-то дополнительная задержка нам не нужна. Если бы была нужна — мы бы её указали вторым аргументом вместо `0`.

Метод `setImmediate(func)`

Для того, чтобы поставить функцию в очередь на выполнение без задержки, в Microsoft предложили метод setImmediate(func) [6] . Он реализован в IE10.

У `setImmediate` единственный аргумент — это функция, выполнение которой нужно запланировать.

В других браузерах `setImmediate` нет, но его можно эмулировать, используя, к примеру, метод postMessage [7] , предназначенный для пересылки сообщений от одного окна другому. Детали работы с `postMessage` вы найдёте в статье Общение окон с разных доменов: postMessage [8]. Желательно читать её после освоения темы «События».

Эмуляция `setImmediate` с его помощью для всех браузеров, кроме IE<8 (в которых нет `postMessage`, так что будет использован setTimeout):

```
if (!window.setImmediate) window.setImmediate = (function() {
  var head = { }, tail = head; // очередь вызовов, 1-связный список

  var ID = Math.random(); // уникальный идентификатор

  function onmessage(e) {
    if(e.data != ID) return; // не наше сообщение
    head = head.next;
    var func = head.func;
    delete head.func;
    func();
  }

  if(window.addEventListener) { // IE9+, другие браузеры
    window.addEventListener('message', onmessage, false);
  } else { // IE8
    window.attachEvent( 'onmessage', onmessage );
  }

  return window.postMessage ? function(func) {
    tail = tail.next = { func: func };
    window.postMessage(ID, "*");
  } :
  function(func) { // IE<8
    setTimeout(func, 0);
  };
}());
```

Есть и более сложные эмуляции, включая MessageChannel [9] для работы с Web Workers [10] и хитрый метод для поддержки IE6-8: https://github.com/NobleJS/setImmediate. Все они по существу являются «хаками», направленными на то, чтобы обеспечить поддержку `setImmediate` в тех браузерах, где его нет.

Тест производительности

Чтобы сравнить реальную частоту срабатывания — измерим время на подсчет от 1 до 100 при `setTimeout/setImmediate`:

testTimeout testImmediate

Запустите пример выше — и вы увидите реальную разницу во времени между `setTimeout(.., 0)` и `setImmediate`. Да, она может быть более в 50, 100 и более раз.

Привязка контекста

Привязка функции к объекту и карринг: "bind/bindLate"

Функцию можно привязать к объекту, чтобы у неё всегда был один и тот же `this`, вне зависимости от контекста вызова.

Это удобно для передачи функции как параметра, чтобы не передавать вместе с ней ссылку на объект.

Для примера, рассмотрим потерю контекста, с которой мы сталкивались в главе про таймеры: вызов `setTimeout(user.sayHi, 1000)` запустит функцию `user.sayHi` в глобальном контексте, не сохраняет `this`:

```
01 function User() {
02   this.id = 1;
03
04   this.sayHi = function() {
05     alert(this.id);
06   };
07 }
08
09 var user = new User();
10
11 setTimeout(user.sayHi, 1000); // выведет "undefined" (ожидается 1)
```

Проблема, конечно, не только в конкретном свойстве `id`, а в том, что, так как не передаётся `this`, из метода нельзя обратиться к другим свойствам объекта.

Самый простой способ это обойти — сделать вызов через обёртку:

```
1 // анонимная фунция-обёртка
2 setTimeout(function() {
3   user.sayHi();
4 }, 1000);
```

Но неужели это единственное решение? Что же теперь — при каждом подобном вызове оборачивать `user.sayHi`?

..Конечно, нет. Можно *привязать* контекст к функции, так что он будет всегда фиксирован. А как — мы сейчас увидим 

Привязка через замыкание

Самый простой способ привязать функцию к правильному `this` — это… Не использовать `this`!

Например, обращаться из функции к объекту через замыкание:

```
function User() {
  this.id = 1;

  var self = this;  // сохранить this в замыкании

  this.sayHi = function() {
    alert(self.id);
  };
}

var user = new User();

setTimeout(user.sayHi, 1000); // выведет "1" (ура, работает)
```

Так как функция была «отучена» от `this`, то её можно смело передавать куда угодно. Контекст будет правильным.

Современный метод `bind`

В современном JavaScript для привязки функций есть метод bind [11] . Он поддерживается большинством современных браузеров, за исключением IE<9, но легко эмулируется.

Этот метод позволяет привязать функцию к нужному контексту и даже к аргументам.

Синтаксис `bind`:

```
var wrapper = func.bind(context[, arg1, arg2...])
```

`func`
: Произвольная функция

`wrapper`
: Функция-обёртка, которую возвращает вызов `bind`. Она вызывает `func`, фиксируя контекст и, если указаны, первые аргументы.

`context`
: Обертка `wrapper` будет вызывать функцию с контекстом `this = context`.

`arg1, arg2, ...`
: Если указаны аргументы `arg1, arg2...` — они будут прибавлены к каждому вызову новой функции, причем встанут *перед* теми, которые указаны при вызове.

Простейший пример, фиксируем только `this`:

```
function f() {
  alert(this.name);
}

var user = { name: "Вася" };

var f2 = f.bind(user);

f2(); // выполнит f с this = user
```

Использование в конструкторе, для привязки метода `sayHi` к создаваемому объекту:

```
function User() {
  this.id = 1;

  this.sayHi = function() {
    alert(this.id);
  }.bind(this);
}

var user = new User();

setTimeout(user.sayHi, 1000); // выведет "1"
```

bind с аргументами

Метод `bind` может создавать обёртку, которая фиксирует не только контекст, но и ряд аргументов.

Например, есть функция перемножения `mul(a, b)`:

```
function mul(a, b) {
  return a * b;
};
```

На ее основе мы можем создать функцию `double`, которая будет удваивать значения:

```
function mul(a, b) {
  return a * b;
};

// double умножает только на два
var double = mul.bind(null, 2); // первым аргументом всегда идёт контекст

alert( double(3) );  // 3*2 = 6
alert( double(4) );  // 4*2 = 8
```

Вызов `mul.bind(null, 2)` возвратил обёртку, которая фиксирует контекст `this = null` и первый аргумент `2`. Контекст в функциях не используется, поэтому не важно, чему он равен.

Получилась функция `double = mul(2, *)`.

Так же можно создать `triple`, утраивающую значение:

```
var triple = mul.bind(null, 3);
```

Карринг

Создание новой функции путём фиксирования аргументов существующей «научно» называется карринг [12] .

Для полноты картины рассмотрим сочетание обеих привязок: контекста и аргументов.

Пусть у объектов `User` есть метод `send(to, message)`, который умеет посылать пользователю `to` сообщение `message`. Создадим функцию для отсылки сообщений Пете от Васи. Для этого нужно зафиксировать контекст и первый аргумент в `send`:

```
function User(name) {
  this.toString = function() {
    return name;
  };

  this.send = function(to, message) {
    alert( this + ': ' + to + ', ' + message );
  };
}

var visitor = new User("Вася");
var admin = new User("Админ");

// создать функцию для пересылки сообщений от admin к visitor
var sendFromAdminToVisitor = admin.send.bind(admin, visitor);

sendFromAdminToVisitor('привет!');// Админ: Вася, привет!
sendFromAdminToVisitor('пока!');  // Админ: Вася, пока!
```

Зачем такая функция может быть нужна? Ну, например, для того чтобы передать её в `setTimeout` или любое другое место программы, где может быть и про пользователей ничего не знают, а нужна какая-нибудь функция одного аргумента для сообщений. И такая вполне подойдёт.

Кросс-браузерная эмуляция `bind`

Для IE<9 и старых версий других браузеров, которые не поддерживают `bind`, его можно реализовать самостоятельно.

Без поддержки карринга это очень просто.

Вот наша собственная функция привязки `bind`:

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}
```

Её использование:

```
function User() {
  this.id = 1;

  this.sayHi = bind(function() {
    alert(this.id);
  }, this);
}

var user = new User();

setTimeout(user.sayHi, 1000); // выведет "1"
```

Вариант `bind` с каррингом

Чтобы функция `bind` передавала аргументы, её нужно «слегка» усложнить:

```
function bind(func, context /*, args*/) {
  var bindArgs = [].slice.call(arguments, 2); // (1)
  function wrapper() {                        // (2)
    var args = [].slice.call(arguments);
    var unshiftArgs = bindArgs.concat(args);  // (3)
    return func.apply(context, unshiftArgs);  // (4)
  }
  return wrapper;
}
```

Страшновато выглядит, да?

Если интересно, работает так (по строкам):

1. Вызов `bind` сохраняет дополнительные аргументы `args` (они идут со 2го номера) в массив `bindArgs`.
2. ... и возвращает обертку `wrapper`.
3. Эта обёртка делает из `arguments` массив `args` и затем, используя метод concat [13], прибавляет их к аргументам `bindArgs (3)`.
4. Затем передаёт вызов `func (4)`.

Использование — в точности, как в примере выше, только вместо `send.bind(admin, visitor)` вызываем `bind(send, admin, visitor)`.

Вариант `bind` для методов

Предыдущие версии `bind` привязывают *любую* функцию к *любому* объекту.

Но если нужно привязать к объекту не произвольную функцию, а ту, которая уже есть в объекте, т.е. его метод, то синтаксис можно упростить.

Обычный вызов `bind`:

```
var userMethod = bind(user.method, user);
```

Альтернативный синтаксис:

```
var userMethod = bind(user, 'method');
```

Поддержка этого синтаксиса легко встраивается в обычный `bind`. Для этого достаточно проверять типы первых аргументов:

```
function bind(func, context /*, args*/) {
  var args = [].slice.call(arguments, 2);

  if (typeof context == "string") {  // если второй аргумент - строка
    // значит аргументы на самом деле имеют вид: (object, "methodName")
    var object = arguments[0];
    var methodName = arguments[1];
    args.unshift(object[methodName], object);
    return bind.apply(this, args); // аргументы переделаны для обычного bind
  }

  function wrapper() {
    var unshiftArgs = args.concat( [].slice.call(arguments) );
    return func.apply(context, unshiftArgs);
  }
  return wrapper;
}
```

Во фреймворках, как правило, есть свои методы привязок. Например, в jQuery это $.proxy [14], который работает как описано ранее:

```
var userMethod = $.proxy(user.method, user);
var userMethod = $.proxy(user, 'method');
```

…С другой стороны, редакторы, которые поддерживают автодополнение, не очень любят такие «оптимизации». Скажем, при попытке автоматизированного переименования `method`, они смогут найти его в вызове `bind(user.method, user)`, но не смогут в `bind(user, 'method')`.

Итого

Итоговый, укороченный, код `bind` для привязки функции или метода объекта:

```
function bind(func, context /*, args*/) {
  var args = [].slice.call(arguments, 2);

  if (typeof context == "string") {   // bind(obj, 'method', ...)
    args.unshift( func[context], func );
    return bind.apply(this, args);
  }

  return function() {
    var unshiftArgs = args.concat( [].slice.call(arguments) );
    return func.apply(context, unshiftArgs);
  };

}
```

Синтаксис: `bind(func, context, аргументы)` или `bind(obj, 'method', аргументы)`.

Также можно использовать func.bind [15] из современного JavaScript, при необходимости добавив кросс-браузерную эмуляцию библиотекой es5-shim [16]:

Позднее связывание "bindLate"

Обычный метод `bind` называется «ранним связыванием», поскольку фиксирует привязку сразу же.

Как только значения привязаны — они уже не могут быть изменены. В том числе, если метод объекта, который привязали, кто-то переопределит — «привязанная» функция этого не заметит.

Позднее связывание — более гибкое, оно позволяет переопределить привязанный метод когда угодно.

Раннее связывание

Например, попытаемся переопределить метод при раннем связывании:

```
01 function bind(func, context) {
02   return function() {
03     return func.apply(context, arguments);
04   };
05 }
06
07 var user = {
08   sayHi: function() { alert('Привет!'); }
09 }
10
11 // привязали метод к объекту
12 var userSayHi = bind(user.sayHi, user);
13
14 // понадобилось переопределить метод
15 user.sayHi = function() { alert('Новый метод!'); }
16
17 // будет вызван старый метод, а хотелось бы - новый!
18 userSayHi(); // выведет "Привет!"
```

…Привязка всё ещё работает со старым методом, несмотря на то что он был переопределён.

Позднее связывание

При позднем связывании `bind` вызовет не ту функцию, которая была в `sayHi` на момент привязки, а ту, которая есть на момент вызова.**

Встроенного метода для этого нет, поэтому нужно реализовать.

Синтаксис будет таков:

```
var func = bindLate(obj, "method");
```

obj
Объект
method
Название метода (строка)

Код:

```
1 function bindLate(context, funcName) {
2   return function() {
3     return context[funcName].apply(context, arguments);
4   };
5 }
```

Этот вызов похож на обычный `bind`, один из вариантов которого как раз и выглядит как `bind(obj, "method")`, но работает по-другому.

Поиск метода в объекте: `context[funcName]`, осуществляется при вызове, самой обёрткой.

Поэтому, если метод переопределили — будет использован всегда последний вариант.

В частности, пример, рассмотренный выше, станет работать правильно:

```
01 function bindLate(context, funcName) {
02   return function() {
03     return context[funcName].apply(context, arguments);
04   };
05 }
06
07 var user = {
08   sayHi: function() { alert('Привет!'); }
09 }
10
11 var userSayHi = bindLate(user, 'sayHi');
12
13 user.sayHi = function() { alert('Здравствуйте!'); }
14
15 userSayHi(); // Здравствуйте!
```

Привязка метода, которого нет

Позднее связывание позволяет привязать к объекту даже метод, которого ещё нет!

Конечно, предполагается, что к моменту вызова он уже будет определён  .

Например:

```
01 function bindLate(context, funcName) {
02   return function() {
03     return context[funcName].apply(context, arguments);
04   };
05 }
06
07 // метода нет
08 var user = {  };
09
10 // ..а привязка возможна!
11 var userSayHi = bindLate(user, 'sayHi');
12
13 // по ходу выполнения добавили метод..
14 user.sayHi = function() { alert('Привет!'); }
15
16 userSayHi(); // Метод работает: Привет!
```

В некотором смысле, позднее связывание всегда лучше, чем раннее. Оно удобнее и надежнее, так как всегда вызывает нужный метод, который в объекте сейчас.

Но оно влечет и небольшие накладные расходы — поиск метода при каждом вызове.

Итого

***Позднее связывание* ищет функцию в объекте в момент вызова.**

Оно используется для привязки в тех случаях, когда метод *может быть переопределён* после привязки или *на момент привязки не существует*.

Обёртка для позднего связывания (без карринга):

```
function bindLate(context, funcName) {
  return function() {
    return context[funcName].apply(context, arguments);
  };
}
```

Статические и фабричные методы объектов

Ранее мы рассматривали статические переменные функции.

Здесь мы поговорим о реализации статических переменных и методов объектов. Они используются для реализации функционала, не привязанного к конкретному экземпляру.

Статические свойства

Статические свойства объекта записываются в его конструктор, например:

```
function Article() {
  Article.count++;
}

Article.count = 0; // статическое свойство-переменная
Article.DEFAULT_FORMAT = "html"; // статическое свойство-константа
```

Они хранят данные, специфичные не для одного объекта, а для всех статей целиком. В примере выше это общее количество статей `Article.count` и формат «по умолчанию» `Article.DEFAULT_FORMAT`.

Статические методы

Статические методы также привязывают к функции-конструктору. Примерами являются встроенные методы String.fromCharCode [17], Date.parse [18].

`Article` в примере ниже считает количество созданных объектов, а метод `Article.showCount()` выводит его:

```
function Article() {
  Article.count++;

  //...
}
Article.count = 0;

Article.showCount = function() {
  alert(this.count);  // (1)
}

// использование
new Article();
new Article();
Article.showCount(); // (2)
```

Здесь `Article.count` — статическое свойство, а `Article.showCount` — статический метод.

Обратите внимание на контекст `this`. Несмотря на то, что переменная и метод — статические, он всё ещё полезен. В строке `(1)` он равен `Article`!

Пример: сравнение объектов

Ещё один хороший способ применения — сравнение объектов.

Например, у нас есть объект `Journal` для журналов. Журналы можно сравнивать — по толщине, по весу, по другим параметрам.

Объявим «стандартную» функцию сравнения, которая будет сравнивать по дате издания. Эта функция сравнения, естественно, не привязана к конкретному журналу, но относится к журналам вообще.

Поэтому зададим её как статический метод `Journal.compare`:

```
function Journal(date) {
  this.date = date;
  // ...
}

// возвращает значение, большее 0, если A больше B, иначе меньшее 0
Journal.compare = function(journalA, journalB) {
  return journalA.date - journalB.date;
};
```

В примере ниже эта функция используется для поиска самого раннего журнала из массива:

```
function Journal(date) {
  this.date = date;

  this.formatDate = function(date) {
    return date.getDate() + '.' + (date.getMonth()+1) + '.' + date.getFullYear();
  };

  this.getTitle = function() {
    return "Выпуск от " + this.formatDate(this.date);
  };

}

Journal.compare = function(journalA, journalB) {
  return journalA.date - journalB.date;
};

// использование:
var journals = [
  new Journal(new Date(2012,1,1)),
  new Journal(new Date(2012,0,1)),
  new Journal(new Date(2011,11,1))
];

function findMin(journals) {
  var min = 0;
  for(var i=0; i<journals.length; i++) {
    // используем статический метод
    if ( Journal.compare(journals[min], journals[i]) > 0 ) min = i;
  }
  return journals[min];
}

alert( findMin(journals).getTitle() );
```

Статический метод также можно создать для реализации функционала, который вообще не требует существования объекта.

Например, метод `formatDate(date)` можно сделать статическим. Он будет форматировать дату «как это принято в журналах», при этом его можно использовать в любом месте кода, не обязательно создавать журнал.

Например:

```
function Journal() { /*...*/ }

Journal.formatDate = function(date) {
  return date.getDate() + '.' + (date.getMonth()+1) + '.' + date.getFullYear();
}

// ни одного объекта Journal нет, просто форматируем дату
alert( Journal.formatDate(new Date) );
```

Фабричные методы

Рассмотрим ситуацию, когда объект нужно создавать различными способами, по разным данным.

Например, это реализовано во встроенном объекте Date [19]. Он по-разному обрабатывает аргументы разных типов:

- `new Date()` — создаёт объект с текущей датой,
- `new Date(milliseconds)` — создаёт дату по количеству миллисекунд `milliseconds`,
- `new Date(year, month, day ...)` — создаёт дату по компонентам год, месяц, день…
- `new Date(datestring)` — читает дату из строки `datestring`

Фабричный статический метод — удобная альтернатива такому конструктору. Так называется статический метод, который служит для создания новых объектов (поэтому и называется «фабричным»).

Пример встроенного фабричного метода — String.fromCharCode(code) [20]. Этот метод создает строку из кода символа:

```
var str = String.fromCharCode(65);
alert(str); // 'A'
```

Но строки — слишком простой пример, посмотрим что-нибудь посложнее.

Допустим, нам нужно создавать объекты `User`: анонимные `new User()` и с данными `new User({name: 'Вася', age: 25})`.

Можно, конечно, создать полиморфную функцию-конструктор `User`:

```
function User(userData) {
  if (userData) { // если указаны данные -- одна ветка if
    this.name = userData.name;
    this.age = userData.age;
  } else {   // если не указаны -- другая
    this.name = 'Аноним';
  }

  this.sayHi = function() { alert(this.name) };
  // ...
}

// Использование

var guest = new User();
guest.sayHi(); // Аноним

var knownUser = new User({name: 'Вася', age: 25});
knownUser.sayHi(); // Вася
```

Подход с использованием фабричных методов был бы другим. Вместо разбора параметров в конструкторе — делаем два метода: `User.createAnonymous` и `User.createFromData`.

Код:

```
function User() {
  this.sayHi = function() { alert(this.name) };
}

User.createAnonymous = function() {
  var user = new User;
  user.name = 'Аноним';
  return user;
}

User.createFromData = function(userData) {
  var user = new User;
  user.name = userData.name;
  user.age = userData.age;
  return user;
}

// Использование

var guest = User.createAnonymous();
guest.sayHi(); // Аноним

var knownUser = User.createFromData({name: 'Вася', age: 25});
knownUser.sayHi(); // Вася
```

Преимущества использования фабричных методов:

- Лучшая читаемость кода. Как конструктора — вместо одной большой функции несколько маленьких, так и вызывающего кода — явно видно, что именно создаётся.
- Лучший контроль ошибок, т.к. если в `createFromData` ничего не передали, то будет ошибка, а полиморфный конструктор создал бы анонимного посетителя.
- Удобная расширяемость.
  Например, нужно добавить создание администратора, без аргументов. Фабричный метод сделать легко: `User.createAdmin = function() { ... }`. А для полиморфного конструктора вызов без аргумента создаст анонима, так что нужно добавить параметр — «тип посетителя» и усложнить этим код.

Поэтому полиморфные конструкторы лучше использовать там, где нужна именно полиморфность, т.е. когда непонятно, какого типа аргумент передадут, и хочется в одном конструкторе охватить все варианты.

А в остальных случаях отличная альтернатива — фабричные методы.

Итого

Статические свойства и методы объекта удобно применять в следующих случаях:

- Общие действия и подсчёты, имеющие отношения ко всем объектам данного типа. В примерах выше это подсчёт количества.
- Методы, не привязанные к конкретному объекту, например сравнение.
- Вспомогательные методы, которые полезны вне объекта, например для форматирования даты.

- Фабричные методы.

Массив: Перебирающие методы

Современный стандарт JavaScript предоставляет много методов для «умного» перебора массивов.

Для их поддержки в IE<9 подключите библиотеку ES5-shim [21].

forEach

Метод arr.forEach(callback[, thisArg]) [22] вызывает функцию `callback` для каждого элемента массива.

Необязательный аргумент `thisArg` будет передан как `this`. Функция вызывается с параметрами: `callback(item, i, arr)`:

- `item` — очередной элемент массива.
- `i` — его номер.
- `arr` — массив, который перебирается.

```
var arr = ["Яблоко", "Апельсин"];

function outputItem(item, i, arr) {
  alert(i + ": " + item + " (массив:" + arr + ")");
}

arr.forEach(outputItem);
```

filter

Метод arr.filter(callback[, thisArg]) [23] создаёт новый массив, в который войдут только те элементы `arr`, для которых вызов `callback(item, i, arr)` возвратит `true`.

Например:

```
var arr = [1, -1, 2, -2, 3];

function isPositive(number) {
  return number > 0;
}

var positiveArr = arr.filter(isPositive);

alert(positiveArr); // 1,2,3
```

map

Метод arr.map(callback[, thisArg]) [24] создаёт новый массив, который будет состоять из результатов вызова `callback(item, i, arr)` для каждого элемента `arr`.

Например:

```
var arr = [1, 2, 3, 4];

function square(number) {
  return number * number;
}

var squaredArr = arr.map(square);

alert(squaredArr); // 1, 4, 9, 16
```

every/some

Метод arr.every(callback[, thisArg]) [25] возвращает `true`, если вызов `callback` вернёт `true` для *каждого* элемента `arr`.

Метод arr.some(callback[, thisArg]) [26] возвращает `true`, если вызов `callback` вернёт `true` для *какого-нибудь* элемента `arr`.

```
var arr = [1, -1, 2, -2, 3];

function isPositive(number) {
  return number > 0;
}

alert( arr.every(isPositive) ); // false, не все положительные
alert( arr.some(isPositive) ); // true, есть хоть одно положительное
```

reduce/reduceRight

Метод arr.reduce(reduceCallback[, initialValue]) [27] применяет функцию `reduceCallback` по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.

Аргументы функции `reduceCallback(previousValue, currentItem, index, arr)`:

- `previousValue` — последний результат вызова функции, он же «промежуточный результат».
- `currentItem` — текущий элемент массива, элементы перебираются по очереди слева-направо.
- `index` — номер текущего элемента.
- `arr` — обрабатываемый массив.

Значение `previousValue` при первом вызове равно `initialValue`. Если `initialValue` нет, то оно равно первому элементу массива, а перебор начинается со второго.

Проще всего это понять на примере соединения строк:

Пусть мы хотим соединить все элементы массива, т.е. сделать `arr.join("")`. Это можно сделать, по очереди прибавив каждый элемент, начиная с первого, к постоянно увеличивающейся результирующей строке.

```
var arr = ["a", "b", "c", "d"];

function join(previousStr, currentItem, i) {
  // прибавляем к строке очередной элемент
  var str = previousStr + currentItem;
  alert(str); // a, ab, abc, abcd
  return str;
}

var result = arr.reduce(join, ""); // abcd

alert("result = " + result);
```

1. Функция `join` начнёт работать с начала массива. Её первые аргументы:
   - ➡ `previousStr = ""` (начальное значение)
   - ➡ `currentItem = "a"` (первый элемент)
   - ➡ `i = 0` (его индекс)

   К пустой строке прибавится `"a"`, что даст вывод `"a"`. *Это значение нужно возвратить!* Оно будет передано как `previousStr` при следующем запуске функции.

2. Второй запуск будет с аргументами:
   - ➡ `previousStr = "a"` (предыдущее возвращённое значение)
   - ➡ `currentItem = "b"` (второй элемент)
   - ➡ `i = 1` (его индекс)

   Его результат: `"ab"`. Как видно, в `previousStr` аккумулируется промежуточный результат.

3. ..И так далее до последнего запуска с аргументами:
   - ➡ `previousStr = "abc"` (предыдущее возвращённое значение)
   - ➡ `currentItem = "d"` (последний элемент)
   - ➡ `i = 3` (его индекс)

Значение, возвращённое из последнего запуска, вернётся как результат: `"abcd"`.

Пример без начального значения:

Посмотрим, что будет, если не указать `initialValue` в вызове `arr.reduce`:

```
var arr = ["a", "b", "c", "d"];

function join(previousStr, currentItem, i) {
  return previousStr + currentItem;
}

var result = arr.reduce(join); // только один аргумент

alert("result = " + result); // abcd
```

Результат — точно такой же! Это потому, что при отсутствии `initialValue` в качестве первого значения берётся первый элемент массива, а перебор стартует со второго.

То есть, первый вызов функции `join` был с аргументами:

- `previousStr = "a"` (первый элемент)
- `currentItem = "b"` (следующий)
- `i = 1` (его индекс)

…А далее — всё так же, как в предыдущем примере.

Пример с суммой:

```
function sum(previousSum, currentItem) {
  return previousSum + currentItem;
}

alert( [1,2,3,4,5].reduce(sum) ); // 1+2+3+4+5 = 15
```

Метод arr.reduceRight [28] работает аналогично, но идёт по массиву справа-налево:

```
function join(previousStr, currentItem) {
  return previousStr + currentItem;
}

alert( ["a","b","c","d"].reduceRight(join) ); // dcba
```

Запуск кода из строки: eval

Функция `eval(code)` позволяет выполнить код, переданный ей в виде строки.

Этот код будет выполнен в *текущей области видимости*.

Пример `eval`

В простейшем случае `eval` всего лишь выполняет код, например:

```
var a = 1;

(function() {

  var a = 2;

  eval(' alert(a) '); // 2

})()
```

Но он может не только выполнить код, но и вернуть результат.

Вызов `eval` возвращает последнее вычисленное выражение:

Например:

```
alert( eval('1+1') ); // 2
```

`eval` и локальные переменные

Код выполняется в текущей области видимости.

Это означает, что текущие переменные могут быть изменены или дополнены:

```
eval("var a = 5");
alert(a); // 5, определена новая переменная
```

```
var a = 0;
eval("a = 5");
alert(a); // 5, переписана существующая переменная
```

Новые переменные не появятся, если код работает в строгом режиме:

```
"use strict";

eval("var a = 5");
alert(a); // ошибка, переменная не определена
```

…Но существующие переменные всё равно будут переопределены:

```
"use strict";

var a = 0;
eval("a = 5");
alert(a); // 5, переписана существующая переменная
```

Это естественно и полностью соответствует логике работы `eval`, которая заключается в том, чтобы динамически включить новый код в существующий.

Запуск кода в глобальном контексте

Хотя, технически, код из `eval` имеет доступ к текущим локальным переменным, но использование этой возможности считается очень плохой практикой. Если уж мы выполняем динамический, возможно, даже загруженный с сервера, код, то пусть он будет сам по себе, со своими переменными, и ни в коем случае не приведет к конфликту с текущими именами.

Поэтому были придуманы трюки, которые позволяют запускать `eval` в глобальной области видимости, без доступа к локальным переменным.

- Например, можно вызвать `eval` не напрямую, а через `window.eval`.

  Это работает везде, кроме IE<9, например:

```
var a = 1;

(function() {

  var a = 2;

  window.eval(' alert(a) '); // 1, везде кроме IE<9

})();
```

➡ В старых IE<9 можно применить нестандартную фунцию execScript [29] . Она, как и `eval`, выполняет код, но всегда в глобальной области видимости и не возвращает значение.

Оба способа можно объединить в единой функции, выполняющей код без доступа к локальным переменным:

```
01 function globalEval(code) { // объединим два способа в одну функцию
02   window.execScript ? execScript(code) : window.eval(code);
03 }
04
05 var a = 1;
06
07 (function() {
08
09   var a = 2;
10
11   globalEval(' alert(a) '); // 1, во всех браузерах
12
13 })();
```

Взаимодействие с внешним кодом, `new Function`

Итак, через `eval` не стоит *менять* локальные переменные. Это плохо, так как нарушает целостность кода, делает его фрагментарным и сложно поддерживаемым.

Но, кроме того, `eval` не стоит даже и *читать* локальные переменные. А то вдруг мы решим переименовать переменную и забудем, что она использовалась в `eval`?

К счастью, существует отличная альтернатива `eval`, которая позволяет корректно взаимодействовать с внешним кодом: `new Function`.

Вызов `new Function('a,b', '..тело..')` создает функцию с указанными аргументами `a,b` и телом. Доступа к текущему замыканию у такой функции не будет, но можно передать локальные переменные и получить результат.

Например:

```
1 var a = 2;
2
3 // вместо обращения к локальной переменной через eval
4 // будем принимать ее как аргумент функции
5 var f = new Function('x', 'alert(x)');
6
7 f(a); // 2
```

Так как областью видимости функции, созданной через `new Function`, является глобальный объект [30], она даже технически не может обратиться к *локальной переменной `a`:

```
1 (function() {
2
3   var a = 2;
4
5   var f = new Function('', 'alert(a)');
6
7   f(); // ошибка
8
9 })();
```

Итого

- Функция `eval(str)` выполняет код и возвращает последнее вычисленное выражение. В современном JavaScript она используется редко.
- Она выполняется в текущей области видимости, поэтому может получать и изменять локальные переменные. Этого следует избегать. Если выполняемый код всё же должен взаимодействовать с локальными переменными — используйте `new Function`. Создавайте функцию из строки и передавайте переменные ей, получайте результат явным образом.
- Есть трюки для выполнения `eval` в глобальной области видимости. Но лучше, все же, использовать `new Function`.

Ещё примеры использования `eval` вы найдёте далее, в главе Формат JSON [31].

Перехват ошибок, "try..catch"

Конструкция `try..catch` — мощное встроенное средство по обработке ошибок и управлению потоком выполнения.

Здесь мы рассмотрим её использование, и общие подходы к обработке ошибок в коде.

Типы ошибок

Ошибки делятся на два глобальных типа.

1. **Синтаксические ошибки** — когда нарушена структура кода, например:

   ```
   for (a = 5) { // неправильная конструкция for, пропущены точки с запятой ;
   ]; // неправильная закрывающая скобка
   ```

   Обычно это ошибки программиста.

   Их также называют «ошибки времени компиляции», поскольку они происходят на этапе разбора кода.

   Как-либо «восстановить» выполнение скрипта здесь нельзя. Браузер не может такой код разобрать и выполнить.

2. **Семантические ошибки**, также называемые «ошибки времени выполнения» — возникают, когда браузер смог прочитать скрипт и уже выполняет код, но вдруг натыкается на проблему.

   Например, «не определена переменная»:

   ```
   alert(nonexistant); // ошибка, переменная не определена!
   ```

   Эти ошибки можно перехватить и обработать самостоятельно, не дать скрипту «упасть».

Что особенно важно, семантические ошибки, в отличие от программных, могут быть частью нормальной работы скрипта!

Конечно, это не касается забытых определений переменных  . Но ошибки могут возникать и в других ситуациях.

Например, посмотрим на следующий интерфейс, который переводит введенное пользователем число в заданную систему счисления, используя вызов toString(основание системы счисления) [32] :

```
var number = +prompt("Введите число", '9');
var base = +prompt("Введите основание системы счисления", '2');

var convertedNumber = number.toString(base);
alert( convertedNumber );  // для 9 в двоичной системе счисления: 1001
```

Пользователь при его использовании может попытаться ввести что-то нестандартное. Например, не число. А вызов toString [33] с `base`, не являющимся числом, приведет к ошибке JavaScript. Вы можете попробовать сами, запустив этот пример.

Мы можем перед вызовом `toString` дотошно проверять аргументы, с риском забыть какую-то ситуацию. В данном конкретном случае можно ввести и числа, но все равно получить ошибку JavaScript. А бывают случаи гораздо менее очевидные, в которых возможных ошибок может быть куча, и все их проверять — дело чрезвычайно муторное…

…К счастью, JavaScript предоставляет нам альтернативу, которая называется `try..catch`!

Она заключается в том, что мы выполняем код «как есть», а если в нем возникнет ошибка, то он не рухнет, а передаст ее в специально выделенный блок.

Конструкция `try..catch`

Конструкция `try..catch` состоит из двух основных блоков: `try`, и затем `catch`. Например:

```
try {
  var number = 9;
  alert( number.toString(2) );

  // ...

  alert('Выполнено без ошибок!');

} catch(e) {

  alert('Ошибка!');

}
```

Работает это так:

1. Выполняется код внутри блока `try`.
2. Если в нём возникнет ошибка, то выполнение `try` прерывается, и управление прыгает в начало блока `catch`.

   Например:

```
try {

  alert('Начало блока try');  // (1) <--

  lalala; // ошибка, переменная не определена!

  alert('Конец блока try');

} catch(err) {

  alert('Управление получил блок catch!'); // (2) <--

}
```

Будут два `alert`'а: `(1)` и `(2)`.

Внутри `catch` получает объект с информацией, какая именно ошибка произошла. В примере выше он назван `err`, но можно и любое другое имя. Мы можем его использовать, для того, чтобы правильно проинформировать посетителя и корректно возобновить работу. Более подробно мы поговорим о нем немного позже, после того, как разберем общую логику работы `try..catch`.

3. Если ошибки нет — блок `catch` игнорируется.

   Например:

   ```
   try {

     alert('Начало блока try');  // (1) <--

     // .. код без ошибок

     alert('Конец блока try');   // (2) <--

   } catch(e) {

     alert('Блок catch не получит управление');

   }
   ```

   Будут два `alert`'а: `(1)` и `(2)`. Блок `catch` не будет использован.

Логика работы `try..catch` позволяет отловить любые семантические ошибки.

В том числе, обработать ошибку при неверном основании системы счисления:

```
var number = +prompt("Введите число", '9');
var base = +prompt("Введите основание системы счисления", '2');

try {
  var convertedNumber = number.toString(base);

  alert( convertedNumber );
} catch(err) {
  alert("Произошла ошибка " + err);
}
```

Объект ошибки

У блока `catch` есть аргумент, в примере выше он обозначен `err`. Это — объект ошибки или, как ещё говорят, *объект исключения* (exception object).

Он содержит информацию о том, что произошло, и может быть разным, в зависимости от ошибки.

Как правило, `err` — объект встроенного типа `Error` и производных от него.

Есть несколько свойств, которые присутствуют в объекте ошибки:

`name`
: Тип ошибки. Например, при обращении к несуществующей переменной равен `"ReferenceError"`.

`message`
: Текстовое сообщение о деталях ошибки.

`stack`
: Везде, кроме IE<9, есть также свойство `stack`, указывающее, где в точности произошла ошибка.

В зависимости от браузера, у него могут быть и дополнительные свойства, см. например Error в MDN [34] и Error в MSDN [35] .

В примере ниже идёт попытка вызова числовой переменной как функции. Это ошибка типа `name = "TypeError"`, с сообщением `message`, которое зависит от браузера:

```
try {
  var a = 5;

  var res = a(1); // ошибка!

} catch(err) {
  alert("name: " + err.name + "\nmessage: " + err.message + "\nstack: " + err.stack);
}
```

..А попытка преобразовать число к неверной системе счисления даст ошибку типа `name = "RangeError"`:

```
try {

  var number = 9;
  var base = 100;

  var convertedNumber = number.toString(base);

} catch(err) {

  alert("name: " + err.name + "\nmessage: " + err.message + "\nstack: " + err.stack);

}
```

Секция `finally`

Конструкция `try..catch` может содержать ещё один блок: `finally`. Выглядит этот расширенный синтаксис так:

```
try {
   .. пробуем выполнить код ..
} catch(e) {
   .. перехватываем исключение ..
} finally {
   .. выполняем всегда ..
}
```

Секция `finally` не обязательна, но если она есть, то она выполняется всегда:

- после блока `try`, если ошибок не было,
- после `catch`, если они были.

Её используют, чтобы завершить начатые операции и очистить ресурсы, которые должны быть очищены в любом случае — как при ошибке, так и при нормальном потоке выполнения.

Например, функция в процессе работы создает новый объект, который нужно в конце уничтожить. Реализация будет выглядеть так:

```
function doSomethingCool() {
  var tmpObject = document.createElement("div"); // создать что-то, что нужно будет очистить

  try {

    .. поработать с tmpObject ..

  } catch(e) {

    .. обработать ошибку ..

  } finally {

    .. удалить объект tmpObject ..

  }

}
```

Блок `finally` позволяет избежать дублирования кода в `try/catch`.

`finally` и `return`

Блок `finally` срабатывает при *любом* выходе из `try..catch`, в том числе и `return`.

В примере ниже, из `try` происходит `return`, но `finally` получает управление до того, как контроль возвращается во внешний код.

```
function func() {

  try {
    // сразу вернуть значение
    return 1;

  } catch(e) {
    alert('Сюда управление не попадёт, ошибок нет');
  } finally {
    alert('Вызов завершён');
  }
}

alert( func() );
```

Это гарантирует освобождение ресурсов в любом случае.

Генерация своих ошибок

Конструкцию `try..catch` можно использовать не только со встроенными ошибками JavaScript. Можно создавать свои ошибки.

Разберем это на примере «из жизни», который позволит понять, зачем это нужно.

Пример: проверка значений

Пример заключается в проверке значений формы. Обычно это реализуется функцией с именем `check...` или `validate...`, которая получает то, что ввёл посетитель и, в простейшем случае, возвращает `true/false`:

```
function checkValidAge(age) {
  if (age >= 18) {
    return true; // OK
  } else {
    return false; // Ошибка
  }
}
```

В реальной жизни простого `true/false` может оказаться недостаточно! Если проверка достаточно сложная, то хорошо бы вернуть ошибку, с текстовым описанием, что именно не так.

Решение без исключений (плохое!) выглядит так:

```
// функция возвращает:
//   false, если всё хорошо
//   или строку с ошибкой
function checkValidAge(age) {
  if (age >= 18) {
    return false;
  } else {
    return "Вы слишком молоды. Приходите через " + (18-age) + " лет";
  }
}
```

..То есть, при таком подходе возврат `false` используется в смысле «всё в порядке», а строка означает ошибку.

Использовать эту функцию нужно так:

```
var error = checkValidAge(age);
if (error) {
  // показать ошибку
}
```

Почему это решение плохое?

В данном случае оно работает, но что, если строка является нормальным результатом функции? Как тогда обозначить ошибку? Вернуть специальный объект?..

Зачем придумывать? Ответ уже есть — и это исключения! Генерация своей ошибки предоставляет альтернативный подход к проблеме, который мы сейчас рассмотрим.

Генерация ошибки: `throw`

Синтаксис: `throw <объект ошибки>`.

Инициирует ошибку, которую можно поймать в `catch`. Объектом ошибки может быть что угодно, например число:

```
try {
  throw 123;
} catch(e) {
  alert(e); // 123
}
```

Можно использовать для этих целей и встроенный объект `Error`:

```
try {
  //...
  throw new Error("Упс, ошибка");
  //...
} catch(e) {
  alert(e.message); // Упс, ошибка
  alert(e.stack); // Где ошибка
}
```

..А можно и свой объект, в том числе с дополнительной информацией. Перепишем функции, которые осуществляют проверки, с использованием `throw` и нашего объекта `BadValueError`:

```
function BadValueError(message) {
  this.name = "BadValue";
  this.message = message;
}

function checkValidAge(age) {
  if (age < 18) {
    throw new BadValueError("Возраст не подходит");
  }
}
```

```
function checkRequired(value) {
  if (value == '') {
    throw new BadValueError("Отсутствует значение");
  }
}
```

Код для проверки:

```
var value = ageInput.value;

try {
  checkRequired(value); // проверить, что значение есть
  checkInteger(value);   // проверить, что значение целое
  checkValidAge(value);// проверить, что диапазон соответствует возрасту
  // ... еще проверки

  /* ввод успешен */
} catch(e) {
  /* обработать ошибку */
}
```

Гораздо короче, не правда ли? И при этом надёжнее, т.к. конструкция `try..catch` поймает *любую* ошибку.

Вложенные вызовы и `try..catch`

Брошенное исключение выпадает из всех циклов, функций и т.п.

Это — особое, уникальное свойство исключений.

Оно означает, что в случае вложенных вызовов — не важно, где было исключение, оно будет поймано ближайшим внешним `try..catch`.

Например:

```
function checkAll(value) {
  checkRequired(value);
  checkInteger(value);
  checkValidAge(value);
}


try {
  checkAll(value);

  alert('Да, вы нам подходите!');

} catch(e) {
  if (e.name == "BadValue") {
    // ок, я знаю, что делать с этой ошибкой
    alert('Ошибка '+e.message);
  } else {
    // ошибка неизвестна, пробросить ее дальше
    throw e;
  }
}
```

В примере выше дополнительно использована техника «проверки и проброса» исключения. Взглянем внимательнее на фрагмент кода:

```
if (e.name == "BadValue") {
  // ок, я знаю что делать с этой ошибкой
  alert('Ошибка '+e.message);
} else {
  // ошибка неизвестна, пробросить ее дальше
  throw e;
}
```

Его идея — в том, что исключения могут быть самыми разными. Возможно, это ошибка в значении, а может быть — нет прав на действие, а может быть — доступ к необъявленной переменной… Возможно что угодно.

Данная конкретная функция `checkAll` умеет обрабатывать только ошибку проверки `BadValue`, что и делает. В том случае, если исключение какое-то другое, то она «пробрасывает» (`throw e`) его дальше, а там оно либо попадёт во внешний блок `try..catch`, который знает, что с ним делать, либо выпадает из скрипта как ошибка — и тогда разработчик увидит его в консоли.

try..finally

Синтаксис `try..finally` (без `catch`) применяется тогда, когда мы хотим произвести очистку в `finally`, но совсем не умеем обрабатывать ошибки.

```
try {
  ..
} finally {
  // очистить ресурсы, а ошибки пусть летят во внешний try..catch
  // (который мы предусмотрели, не правда ли? иначе скрипт упадёт)
}
```

Итого

Исключения в JavaScript нужны редко. Но там, где они нужны — они нужны.

- Основная область применения — попытка сделать что-то, что может вернуть ошибку, но проверить заранее нельзя. Например, получить информацию из `IFRAME'a`.
- Другая область применения — завернуть блок кода, в котором может быть ошибка, в конструкцию `try..catch`, и вместо многочисленных проверок обработать ошибку один раз, внизу.

  Мы видели это в примере с `checkRequired/checkInteger/checkAge`.

Ошибка, генерируемая `throw`, «выпадает» из функции, и передаёт управление на ближайший `catch`:

```
function a() {  b(); }

function b() {  c(); }

function c() { throw new Error("Ошибка!"); }

try {
  a(); // a() -> вызывает b() -> вызывает c() -> Ошибка!
} catch(e) {
  alert(e); // <-- ошибка выпадет в ближайший try..catch
}
```

Это делает `try..catch + throw` мощным средством контроля выполнения.

При использовании своих ошибок рекомендуется генерировать объекты встроенного типа Error [36] или, когда мы познакомимся с наследованием в JavaScript — наследующие от него.

Ваши объекты также могут содержать дополнительную информацию об обстоятельствах ошибки, полезную для её обработки:

```
var err = new Error("Ошибка");
err.extra = new Date();

alert(err.message + " в " + err.extra); // "Ошибка в (дата)"
```

Формат JSON

В этой главе мы рассмотрим работу с форматом JSON [37] . Это один из наиболее удобных форматов для JavaScript.

В современных браузерах (для IE7- реализуются библиотекой) есть замечательные методы, знание тонкостей которых делает операции с JSON простыми и комфортными.

Формат JSON

Данные в формате JSON (RFC 4627 [38]) представляют собой значения или JavaScript-объекты `{ ... }` или массивы `[ ... ]`, содержащие значения одного из типов:

- строки в двойных кавычках,
- число,

- логическое значение `true/false`,
- `null`.

Объекты JSON отличаются от обычных JavaScript-объектов более строгими требованиями к строкам — они должны быть именно в двойных кавычках.

В частности, первые два свойства объекта ниже — некорректны:

```
{
  name: "Вася",          // ошибка: ключ name без кавычек!
  "surname": 'Петров',   // ошибка: одинарные кавычки у значения!
  "age": 35              // .. а тут всё в порядке.
  "isAdmin": false       // и тут тоже всё ок
}
```

`JSON.stringify` и `JSON.parse`

- Метод `JSON.stringify(value, replacer, space)` преобразует (*«сериализует»*) значение в JSON-строку.

  Он поддерживается во всех браузерах, включая IE8+. Для более старых IE рекомендуется библиотека JSON-js [39], которая добавляет аналогичную функциональность.

- Метод `JSON.parse(str, reviver)` читает JavaScript-значение из строки.

Пример использования:

```
var event = {
  title: "Конференция",
  date: "сегодня"
};

var str = JSON.stringify(event);
alert( str ); // {"title":"Конференция","date":"сегодня"}

// Обратное преобразование.
event = JSON.parse(str);
```

Детали `JSON.stringify`

Метод `JSON.stringify` обладает рядом расширенных возможностей, которые бывают очень полезны в реальных задачах.

Сериализация объектов, `toJSON`

При сериализации объекта вызывается его метод `toJSON`. Если такого метода нет — перечисляются его свойства, кроме функций.

Посмотрим это в примере посложнее:

```
function Room() {
   this.number = 23;

   this.occupy = function() {};
}

event = {
  title: "Конференция",
  date: new Date(2012, 0, 1),
  room: new Room()
};

alert( JSON.stringify(event) );
/*
  {
    "title":"Конференция",
    "date":"2012-01-30T20:00:00.000Z",  // (1), при зоне GMT+4
    "room": {"number":23}               // (2)
  }
*/
```

Обратим внимание на два момента:

1. Дата превратилась в строку. Это не случайно: у всех дат есть встроенный метод `toJSON`. Его результат в данном случае — строка в зоне UTC.

   Если объект `new Date` создавался на компьютере с локальной временной зоной GMT+4, то сериализованный UTC-вариант — на 4 часа меньше.

2. У объекта `new Room` нет метода `toJSON`. Поэтому он сериализуется перечислением свойств.

   Мы, конечно, могли бы добавить такой метод, тогда в итог попал бы её результат:

   ```
   function Room() {
     this.number = 23;

     this.toJSON = function() {
       return this.number;
     };
   }

   alert( JSON.stringify( new Room() ) ); // 23
   ```

Исключение свойств

Попытаемся преобразовать в JSON объект, содержащий ссылку на DOM.

Например:

```
var user = {
  name: "Вася",
  age: 25,
  elem: document.body
}

alert( JSON.stringify(user) ); // ошибка!
// TypeError: Converting circular structure to JSON (текст из Chrome)
```

Произошла ошибка! В чём же дело, неужели DOM-объекты запрещены? Как видно из текста ошибки — дело совсем в другом. DOM-объект — сложная структура с круговыми ссылками, поэтому его преобразовать невозможно. Да и нужно ли?

Во втором параметре `JSON.stringify(value, replacer)` можно указать массив свойств, которые подлежат сериализации.

Например:

```
var user = {
  name: "Вася",
  age: 25,
  elem: document.body
};

alert( JSON.stringify(user, ["name", "age"]) );
// {"name":"Вася","age":25}
```

Для более сложных ситуаций вторым параметром можно передать функцию `function(key, value)`, которая возвращает сериализованное `value` либо `undefined`, если его не нужно включать в результат:

```
var user = {
  name: "Вася",
  age: 25,
  elem: document.body
};

var str = JSON.stringify(user, function(key, value) {
  if (key == 'elem') return undefined;
  return value;
} );

alert(str); // {"name":"Вася","age":25}
```

В примере выше функция пропустит свойство с названием `elem`. Для остальных она просто возвращает значение, передавая его стандартному алгоритму. А могла бы и как-то обработать.

Функция `replacer` работает рекурсивно.

То есть, если объект содержит вложенные объекты, массивы и т.п., то все они пройдут через `replacer`.

Красивое форматирование

В методе `JSON.stringify(value, replacer, space)` есть ещё третий параметр `space`.

Если он является числом — то уровни вложенности в JSON оформляются указанным количеством пробелов, если строкой — вставляется эта строка.

Например:

```
var user = {
  name: "Вася",
  age: 25,
  roles: {isAdmin: false, isEditor: true}
};

var str = JSON.stringify(user, "", 4);

alert(str);
/* Результат -- красиво сериализованный объект:
{
    "name": "Вася",
    "age": 25,
    "roles": {
        "isAdmin": false,
        "isEditor": true
    }
}
*/
```

Умный разбор: `JSON.parse(str, reviver)`

Предположим, мы получили с сервера корректно сериализованный объект `event`. И теперь нужно восстановить его.

Вызовем для этого `JSON.parse`:

```
var str = '{"title":"Конференция","date":"2012-11-30T00:00:00.000Z"}';

var event = JSON.parse(str);

alert( event.date.getDate() ); // ошибка!
// TypeError: Object 2012-11-30T00:00:00.000Z has no method 'getDate'
```

…Увы, ошибка!

Дело в том, что значением `event.date` является строка, а отнюдь не объект `Date`. Откуда методу `JSON.parse` знать, что нужно превратить строку именно в дату?

Для интеллектуального восстановления из строки у `JSON.parse(str, reviver)` есть второй параметр, который является функцией `function (key, value)`.

Она принимает по очереди все создаваемые пары ключ-значение и может возвратить преобразованное значение `value`, либо `undefined`, если его нужно игнорировать.

В данном случае мы можем создать правило, что ключ `date` всегда означает дату:

```
// дата в строке - в формате UTC
var str = '{"title":"Конференция","date":"2012-11-30T20:00:00.000Z"}';

var event = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( event.date.getDate() ); // 1 или 30, в зависимости от локальной зоны
```

IE<8: разбор JSON с `eval`

Функция `eval(code)` выполняет код и, если это выражение, то возвращает его значение.

Поэтому её можно использовать для чтения JSON, например:

```
var str= '{ \
    "name": "Вася", \
    "age": 25 \
}';

var user = eval('(' + str + ')');

alert(user.name); // Вася
```

Зачем здесь нужны скобки, почему не просто `eval(str)`? Как вы считаете?

Подумали?… Нет, правда, зачем?

…Всё дело в том, что в JavaScript с фигурной скобки { начинаются и объекты и блоки кода. Если передать `eval` объект напрямую, то он подумает, что первая { начинает блок кода:

```
{
    "name": "Вася",
    "age": 25
}
```

Выполнение такого кода, конечно, приведёт к ошибке.

А если `eval` получает выражение в скобках `( ... )`, то интерпретатор точно знает, что блока кода внутри быть не может, значит это объект.

Безопасность, проверка JSON

Если JSON разбирается при помощи `eval`, то мы должны быть уверены, что это именно JSON, а не злонамеренный скрипт.

При получении JSON с нашего сервера такая уверенность есть, а если он из другого источника?

Метод `JSON.parse` гарантирует, что некорректный JSON просто выдаст ошибку, а в `eval` можно добавить дополнительную проверку регулярным выражением, описанным в RFC 4627, секции 6.

Код преобразования с проверкой:

```
var obj = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test(
  str.replace(/"(\\.|[^"\\])*"/g, ''))) &&
  eval('(' + str + ')');
```

Пример опасного сценария с `eval`:

```
// злой объект
var str = '{}(function() {alert(1)}())';

eval('(' + str + ')'); // 1, выполнилась функция
```

Опасный сценарий с проверкой:

```
// злая функция
var str = '{}(function() {alert(1)}())';

var user = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test(
  str.replace(/"(\\.|[^"\\])*"/g, ''))) &&
  eval('(' + str + ')');

alert(user); // false, объект не преобразован
```

Тест: повторение тонких мест

Вопросы из этого теста проверяют знание множества важных нюансов JavaScript.

Ряд вопросов взят из тестов http://dmitry.baranovskiy.com/post/91403200, http://perfectionkills.com/javascript-quiz/ и http://dmitrysoshnikov.com/ecmascript/the-quiz/. Спасибо их составителям, рекомендую пройти и их тоже.

- Все задачи выполняются в достаточно современных браузерах, IE8+.
- Код задачи является единственным кодом, который запускается в окне. В частности, никаких дополнительных переменных, кроме указанных, нет.

Итак, поехали…

Тест сложный! Если у вас не получилось ответить на многие вопросы — не расстраивайтесь. Его цель — не только проверить знания, но и помочь заполнить пробелы в них.

Решения задач

Решение задачи: Вывод чисел каждые 100мс

```
function printNumbersInterval20_100() {
  var i = 1;
  var timerId = setInterval(function() {
    console.log(i);
    if (i == 20) clearInterval(timerId);
    i++;
  }, 100);
}

// вызов
printNumbersInterval20_100();
```

Решение задачи: Вывод чисел каждые 100мс, через setTimeout

```
function printNumbersTimeout20_100() {
  var i = 1;
  var timerId = setTimeout(function go() {
    console.log(i);
    if (i < 20) setTimeout(go, 100);
    i++;
  }, 100);
}

// вызов
printNumbersTimeout20_100();
```

Решение задачи: Для подсветки setInterval или setTimeout?

Нужно выбрать вариант 2, который гарантирует браузеру свободное время между выполнениями `highlight`.

Первый вариант может загрузить процессор на 100%, если `highlight` занимает время, близкое к 10мс или, тем более, большее чем 10мс, т.к. таймер не учитывает время выполнения функции.

Что интересно, в обоих случаях браузер не будет выводить предупреждение о том, что скрипт занимает много времени. Но от 100% загрузки процессора возможны притормаживания других операций. В общем, это совсем не то, что мы хотим, поэтому вариант 2.

Решение задачи: Кто быстрее?

Обычно первый бегун будет быстрее.

Но возможен и такой вариант, что время совпадает.

Создадим реальные объекты `Runner` и запустим их для проверки:

```
function Runner() {
  this.steps = 0;

  this.step = function() {
    doSomethingHeavy();
    this.steps++;
  }

  function doSomethingHeavy() {
    for(var i=0; i<10000; i++) {
      this[i] = this.step + i;
    }
  }

}

var runner1 = new Runner();
var runner2 = new Runner();

// запускаем бегунов
setInterval(function() {
  runner1.step();
}, 15);

setTimeout(function go() {
  runner2.step();
  setTimeout(go, 15);
}, 15);

// кто сделает больше шагов?
setTimeout(function() {
  alert(runner1.steps);
  alert(runner2.steps);
}, 5000);
```

- Если бы в шаге `step()` не было вызова `doSomethingHeavy()`, то количество шагов было бы равным, так как времени на такой шаг нужно очень мало.

  Интерпретатор JavaScript старается максимально оптимизировать такие часто повторяющиеся «узкие места». В данном случае вызов `step` свёлся бы к одной операции микропроцессора. Это пренебрежимо мало по сравнению с остальным кодом.

- Так как у нас шаг, всё же, что-то делает, и функция `doSomethingHeavy()` специально написана таким образом, что к одной операции свести её нельзя, то первый бегун успеет сделать больше шагов.

  Ведь в `setTimeout` пауза `15` мс будет *между* шагами, а `setInterval` шагает равномерно, каждые `15` мс. Получается чаще.

- Наконец, есть браузеры (IE9), в которых при выполнении JavaScript таймер «не тикает». Для них оба варианта будут вести себя как `setTimeout`, так что количество шагов будет одинаковым.

Решение задачи: Что выведет setTimeout?

Вызов `alert(i)` в `setTimeout` выведет `100000000`, так как срабатывание будет гарантировано после окончания работы текущего кода.

Очередь до запланированных вызовов доходит всегда лишь после окончания текущего скрипта.

Можете проверить это запуском:

```
01 setTimeout(function() {
02   alert(i);
03 }, 10);
04
05 var i;
06
07 function f() {
08   // точное время выполнения не играет роли
09   // здесь оно заведомо больше задержки setTimeout
10   for(i=0; i<1e8; i++) f[i%10] = i;
11 }
12
13 f();
```

Ответ на второй вопрос: 3 (сразу после).

Вызов планируется на `10мс` от времени вызова `setTimeout`, но функция выполняется больше, чем `10мс`, поэтому к моменту ее окончания время уже подошло и отложенный вызов выполняется тут же.

Решение задачи: Что выведет после setInterval?

Вызов `alert(i)` в `setTimeout` введет `100000001`. Почему — будет понятно из ответа на второй вопрос.

Можете проверить это запуском:

```
01 var timer = setInterval(function() {
02   i++;
03 }, 10);
04
05 setTimeout(function() {
06   clearInterval(timer);
07   alert(i);
08 }, 50);
09
10 var i;
11
12 function f() {
13   // точное время выполнения не играет роли
14   // здесь оно заведомо больше 50мс
15   for(i=0; i<1e8; i++) f[i%10] = i;
16 }
17
18 f();
```

Ответ на второй вопрос: 4 (сразу же по окончании `f` один раз).

Планирование `setInterval` будет вызывать функцию каждые `10мс` после текущего времени. Но так как интерпретатор занят долгой функцией, то до конца ее работы никакого вызова не происходит.

За время выполнения `f` может пройти время, на которое запланированы несколько вызовов `setInterval`, но в этом случае остается только один, т.е. накопления вызовов не происходит. Такова логика работы `setInterval`.

После окончания текущего скрипта интерпретатор обращается к очереди запланированных вызовов, видит в ней `setInterval` и выполняет. А затем тут же выполняется `setTimeout`, очередь которого тут же подошла.

Итого, как раз и видим, что `setInterval` выполнился ровно 1 раз по окончании работы функции. Такое поведение кросс-браузерно.

Решение задачи: Функция-задержка

```
01 function delay(f, ms) {
02
03   return function() {
04     var savedThis = this;
05     var savedArgs = arguments;
06
07     setTimeout(function() {
08       f.apply(savedThis, savedArgs);
09     }, ms);
10   };
11
12 }
13
14 function f(x) {
15   alert(x);
16 }
17
18 var f1000 = delay(f, 1000);
19 var f1500 = delay(f, 1500);
20
21 f1000("тест"); // выведет "тест" через 1000 миллисекунд
22 f1500("тест2"); // выведет "тест2" через 1500 миллисекунд
```

Обратим внимание на то, как работает обёртка:

```
1 return function() {
2   var savedThis = this;
3   var savedArgs = arguments;
4
5   setTimeout(function() {
6     f.apply(savedThis , savedArgs);
7   }, ms);
8 };
```

Именно обёртка возвращается декоратором `delay` и будет вызвана. Чтобы передать аргумент и контекст функции, вызываемой через `ms` миллисекунд, они копируются в локальные переменные `savedThis` и `savedArgs`.

Это один из самых простых, и в то же время удобных способов передать что-либо в функцию, вызываемую через `setTimeout`.

Решение задачи: Вызов не чаще чем в N миллисекунд

Решение: http://learn.javascript.ru/play/tutorial/timers/debounce.html

Вызов `debounce` возвращает функцию-обёртку. Все необходимые данные для неё хранятся в замыкании.

При вызове ставится таймер и состояние `state` меняется на константу `COOLDOWN` («в процессе охлаждения»).

Последующие вызовы игнорируются, пока таймер не обнулит состояние.

Решение задачи: Тормозилка

Решение: http://learn.javascript.ru/play/tutorial/timers/throttle.html

Вызов `throttle` возвращает функцию-обёртку. Все необходимые данные для неё хранятся в замыкании.

В первую очередь — это состояние `state`, которое вначале не назначено (`null`), при первом вызове получает значение `COOLDOWN` («в процессе охлаждения»), а при следующем вызове `CALL_SCHEDULED`.

При таймауте состояние проверяется, и если оно равно `CALL_SCHEDULED` — происходит новый вызов.

Решение задачи: Повторный bind

Ответ: `"Вася"`.

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "Вася"} ).bind( {name: "Петя"} );

f(); // Вася
```

Первый вызов `f.bind(..Вася..)` возвращает «обёртку», которая устанавливает контекст для `f` и передаёт вызов `f`.

Следующий вызов `bind` будет устанавливать контекст уже для этой обёртки, это ни на что не влияет.

Чтобы это проще понять, используем наш собственный вариант `bind` вместо встроенного:

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}
```

Код станет таким:

```
function f() {
  alert(this.name);
}

f = bind(f, {name: "Вася"} ); // (1)
f = bind(f, {name: "Петя"} ); // (2)

f(); // Вася
```

Здесь видно, что первый вызов `bind`, в строке `(1)`, возвращает обёртку вокруг `f`, которая выглядит так (выделена):

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}
```

В этой обёртке нигде не используется `this`, только `func` и `context`. Посмотрите на код, там нигде нет `this`.

Поэтому следующий `bind` в строке `(2)`, который выполняется уже над обёрткой и фиксирует в ней `this`, ни на что не влияет. Какая разница, что будет в качестве `this` в функции, которая этот `this` не использует?

Решение задачи: Счетчик объектов

Решение (как вариант):

```
01 function Article() {
02   this.created = new Date;
03
04   Article.count++; // увеличиваем счетчик при каждом вызове
05   Article.last = this.created; // и запоминаем дату
06 }
07 Article.count = 0; // начальное значение
08 // (нельзя оставить undefined, т.к. Article.count++ будет NaN)
09
10 Article.showStats = function() {
11   alert('Всего: ' + this.count + ', Последняя: ' + this.last);
12 };
13
14 new Article();
15 new Article();
16
17 Article.showStats(); // Всего: 2, Последняя: (дата)
18
19 new Article();
20
21 Article.showStats(); // Всего: 3, Последняя: (дата)
```

Решение задачи: Массив частичных сумм

Метод `arr.reduce` подходит здесь идеально. Достаточно пройтись по массиву слева-направа, накапливая текущую сумму в переменной и, кроме того, добавляя её в результирующий массив.

Первоначальный вариант может выглядеть так:

```
01 function getSums(arr) {
02   var result = [];
03
04   arr.reduce(function(sum, item) {
05     result.push(sum);
06     return sum + item;
07   });
08
09   return result;
10 }
11
12 alert(getSums([1,2,3,4,5])); // результат: 1,3,6,10
```

Если вы его запустите, то обнаружите, что результат не совсем тот. В получившемся массиве всего четыре элемента, отсутствует последняя сумма.

Дело в том, что последняя сумма — это и есть результат метода `reduce`, который на ней заканчивает проход и далее функцию не вызывает. Поэтому она оказывается не добавленной в `result`.

Исправим это:

```
01 function getSums(arr) {
02   var result = [];
03
04   var totalSum = arr.reduce(function(sum, item) {
05     result.push(sum);
06     return sum + item;
07   });
08   result.push(totalSum);
09
10   return result;
11 }
12
13 alert(getSums([1,2,3,4,5])); // 1,3,6,10,15
14 alert(getSums([-2,-1,0,1])); // -2,-3,-3,-2
```

Решение задачи: Eval-калькулятор

Вычислить любое выражение нам поможет `eval`:

```
var expr = prompt("Введите выражение?", '2*3+2');

alert(eval(expr));
```

При этом посетитель потенциально может делать все, что угодно.

Чтобы ограничить выражения только математикой, вводимую строку нужно проверять при помощи регулярных выражений [40] на наличие любых символов, кроме букв, пробелов и знаков пунктуации.

Решение задачи: Finally или просто код?

Поведение будет разным, если управление каким-то образом выпрыгнет из `try..catch`, например:

```
function f() {
  try {
    ...
    return result;
  } catch(e) {
    ...
  } finally {
    очистить ресурсы
  }
}
```

Или ошибка будет проброшена наружу:

```
function f() {
  try {
    ...

  } catch(e) {
    ...
    if (не умею обрабатывать эту ошибку) {
      throw e;
    }

  } finally {
    очистить ресурсы
  }
}
```

В этих случаях `finally` гарантирует выполнение кода и корректную очистку ресурсов функции.

Решение задачи: Eval-калькулятор с ошибками

Схема решения

Вычислить любое выражение нам поможет `eval`:

```
alert( eval("2+2") ); // 4
```

Считываем выражение в цикле `while(true)`. Если при вычислении возникает ошибка — ловим её в `try..catch`.

Ошибкой считается, в том числе, получение `NaN` из `eval`, хотя при этом исключение не возникает. Можно бросить своё исключение в этом случае.

Код решения

```
while(true) {
  expr = prompt("Введите выражение?", '2-');

  try {
    var res = eval(expr); // при ошибке будет catch

    if (isNaN(res)) { // наша ошибка
      throw new Error("Результат неопределён");
    }

    break; // все ок, выход из цикла

  } catch(e) {

    alert("Ошибка: "+e.message+", повторите ввод");

  }
}

alert(res);
```

Полное решение: http://learn.javascript.ru/play/tutorial/intro/eval-calc-try.html.

Решение задачи: Превратите объект в JSON

Обычный вызов `JSON.stringify(team)` выдаст ошибку, так как объекты `leader` и `soldier` внутри структуры ссылаются друг на друга.

Формат JSON не предусматривает средств для хранения ссылок.

Чтобы превращать такие структуры в JSON, обычно используются два подхода:

1. Добавить в `team` свой код `toJSON`:

```
team.toJSON = function() {
  /* свой код, который может создавать копию объекта без круговых ссылок и передавать управление JSON.stringify */
}
```

При этом, конечно, понадобится и своя функция чтения из JSON, которая будет восстанавливать объект, а затем дополнять его круговыми ссылками.

2. Можно учесть возможную проблему в самой структуре, используя вместо ссылок `id`. Как правило, это несложно, ведь на сервере у данных тоже есть идентификаторы.

   Изменённая структура может выглядеть так:

```
var leader = {
  id: 12,
  name: "Василий Иванович"
};

var soldier = {
  id: 51,
  name: "Петька"
};

// поменяли прямую ссылку на ID
leader.soldierId = 51;
soldier.leaderId = 12;

var team = {
 12: leader,
 51: soldier
};
```

   ..Но действительно ли это решение будет оптимальным? Использовать структуру стало сложнее, и вряд ли это изменение стоит делать лишь из-за JSON. Вот если есть другие преимущества, тогда можно подумать.

3. Универсальный вариант подхода, описанного выше — это особая реализация JSON, расширяющая формат для поддержки ссылок.

   Она, к примеру, сделана во фреймворке Dojo.

   При вызове `dojox.json.ref.toJson(team)` будет создано следующее строковое представление:

```
[{"name":"Василий Иванович","soldier":{"name":"Петька","leader":{"$ref":"#0"}}},
{"$ref":"#0.soldier"}]
```

   Метод разбора такой строки — также свой: `dojox.json.ref.fromJson`.

Решение задачи: JS-вопросник

Ответ: **2** (код функции).

```
f.call(f);

function f() {
  alert( this );
}
```

Функция `f` определяется при входе в область видимости, так что на первой строке она уже есть.

Далее, вызов `f.call(f)` вызывает функцию, передавая ей `f` в качестве `this`, так что выводится строковое представление `f`.

Решение задачи: JS-вопросник

Ответ: **3**

Если при вызове функции через `call/apply` первым аргументом является `null/undefined`, то при работе в режиме старого стандарта JavaScript браузер вызывает её в контексте `window`.

```
function f() {
  alert(this); // выведет текстовое представление window
}
f.call(null);
```

С другой стороны, если стоит режим соответствия стандартам, то будет выведен именно `null`:

```
"use strict";

function g() {
  alert(this); // выведет null, если браузер поддерживает strict mode
}
g.call(null);
```

То есть, результат может быть разным, в зависимости от того, включён ли строгий режим. В этой задаче предполагается, что режим обычный, поэтому ответ `3`.

Решение задачи: JS-вопросник

Ответ: **5** (зависит от браузера)

В современных браузерах будет ошибка, так как это Named Function Expression. Его имя `g` видно только внутри функции, а снаружи переменная `g` не определена.

```
var f = function g(){ return 23; };
alert( typeof g() );
```

Приоритет вызова функции выше, чем `typeof`, так что браузер попытается вызвать функцию `g()`, а такой переменной нет.

Старый IE<9 выдаст `'number'`, т.к. в нём это ограничение видимости не поддерживается.

Решение задачи: JS-вопросник

Ответ: **4**

```
var y = 1;
var x = y = typeof x;

alert(x + 1);
```

1. До начала выполнения кода обе переменные равны `undefined`.
2. Затем выполняется присваивание `y = 1`.
3. Присваивание `x = y = typeof x` выполняется справа налево (как и все тройные присваивания). Сначала `y = typeof x`, и так как `x` равен `undefined`, то в `y` записывается `typeof x == "undefined"` — строка.

   Затем идёт присвоение`x = y`. В итоге получаем, что `x == "undefined"`.
4. Последняя строка прибавляет `1`. Так как слева строка, то оператор `"+"` преобразует и `1` к строке.
5. Ответ: `"undefined1"`.

Решение задачи: JS-вопросник

Ответ: **4** (ошибка).

```
"use strict";

var f, user;

user = {
  sayHi: function(){ alert(this.fullName); },
  fullName: "Василий Петрович"
};

(f = user.sayHi)();
```

Строка `(f = user.sayHi)()` идентична последовательности операций:

```
f = user.sayHi; // присвоить
f();              // вызвать
```

При этом в функцию передаётся `null` в качестве `this`. А в `null` свойства `fullName` нет.

Решение задачи: JS-вопросник

Ответ: **4** (выведет `12`).

```
alert( [] + 1 + 2 );
```

Первым делом объект `[]` превращается в примитив. У него нет `valueOf` (с примитивным результатом), так что вызывается `toString` и возвращает список элементов через запятую, т.е. пустую строку.

Получается сложение `"" + 1 + 2`. Далее, так как один из аргументов — строка, то оператор `"+"` преобразует второй тоже к строке, и в итоге получится строка `"12"`.

Решение задачи: JS-вопросник

Ответ: **1**.

```
alert( null == undefined );
```

Эти два значения равны друг другу и больше ничему не равны.

В частности:

```
alert( null == 0 || undefined == 0);   // false
alert( null == '' || undefined == ''); // false
```

Преобразования типов здесь не происходит. Оператор `==` работает с `null/undefined` без преобразования, все варианты прописаны в спецификации.

Решение задачи: JS-вопросник

Ответ: **6** (зависит от браузера).

```
var x = 1;
if (function f(){}) {
  x += typeof f; // (*)
}
alert(x);
```

Во всех, кроме IE<9, в `if` находится Named Function Expression, так что имя `f` доступно только из него, а снаружи не видно. Таким образом, переменная `f` в строке `(*)` не определена и к `x` прибавляется строка `"undefined"` (будет вариант 3).

В старых IE функция `f` будет видна везде, так что в этих браузерах получится `1function` (вариант 2).

Решение задачи: JS-вопросник

Ответ: **ошибка**.

```
function f() {
  var a = 5;
  return new Function('b', 'return a + b');
}

alert( f()(1) );
```

Функция, созданная через `new Function` получает в `[[Scope]]` ссылку на `window`, а не на текущий контекст. Поэтому новая функция не увидит `a` и выдаст ошибку.

Решение задачи: JS-вопросник

Ответ: **4** (`true`, `true`).

```
function F(){ }

alert( F instanceof Function );
alert( new F() instanceof Object );
```

1. Функция является объектом встроенного класса `Function`, так что `true`.
2. Вспомним, как работает оператор `new`. Он генерирует пустой объект `{}` (`new Object`) и передаёт его функции-конструктору. В конце работы функции, если нет явного вызова `return obj`, где `obj` — какой-то другой объект, то возвращается `this`.

   В данном случае функция ничего не делает, так что возвращается `this`, равный пустому объекту `{}` (`new Object`). Результатом проверки `new Object instanceof Object` является `true`.

Решение задачи: JS-вопросник

Ответ: **2** (`false`, `true`).

```
function F(){ return F; }

alert( new F() instanceof F );
alert( new F() instanceof Function );
```

Если функция, запущенная через `new`, возвращает объект (не примитив!), то именно этот объект служит результатом, вместо `this`. Так что результатом `new F` является сама функция `F`.

Поэтому получается, что первый `alert` проверяет: `F instanceof F`. Это неверно, т.к. `a instanceof B` проверяет, был ли объект `a` создан конструктором `B` (с учетом прототипов, но здесь это неважно). Функция `F` не была создана собой же, а является объектом встроенного класса `Function`, поэтому `false`.

Следующая строка идентична такой проверке: `F instanceof Function`. Это верно.

Решение задачи: JS-вопросник

Ответ: **3**.

```
alert( typeof null );
```

Это особенность спецификации JavaScript. Значением оператора `typeof null` является строка `"object"`.

Решение задачи: JS-вопросник

Ответ: **2**.

```
alert( 20e-1['toString'](2) );
```

1. Запись числа `20e-1` означает `20`, сдвинутое на 1 знак после запятой, т.е. `2`:

   ```
   alert( 20e-1 ); // 2
   ```

2. У каждого числа есть метод `toString(radix)`, который преобразует число в строку, используя `radix` как основание системы счисления. В этом коде `radix = 2`. Так что возвращается `2` в двоичной системе: `"10"`.

Решение задачи: JS-вопросник

Ответ: **4**.

```
var a = (1,5 - 1) * 2;
alert(a);
```

Оператор «запятая», вычисляемый в скобках, возвращает последнее вычисленное значение. То есть, `1` игнорируется и возвращается `4` (`5-1`). Затем идёт умножение на два, в результате — `8`.

Решение задачи: JS-вопросник

Ответ: **4**.

```
var a = [1,2]

(function() { alert(a) })()
```

Всё дело в пропущенной точке с запятой после `[1,2]`.

Браузер интерпретирует это так:

```
var a = [1,2](function() { alert(a) })()
```

..То есть, пытается вызвать массив как функцию, что, естественно, не выходит. Отсюда и ошибка.

Решение задачи: JS-вопросник

Ответ: **1** (`true`).

```
alert("ёжик" > "яблоко");
```

В JavaScript строки сравниваются посимвольно. Символы соотносятся как их коды.

В кодировке Unicode код буквы `"ё"` больше кода буквы `"я"`. Поэтому `"ёжик"` больше.

Решение задачи: JS-вопросник

Правильный ответ: пункт 3 (т.е. `=2`):

```
alert(null + {0:1}[0] + [,[1],][1][0]);
```

Подробнее вычисления:

1. `null` превращается в `0`
2. `{0:1}` — это объект, у которого ключ `0` имеет значение `1`

   ```
   {0:1}[0] == 1
   ```

3. Второе слагаемое `[,[1],]` — это массив с 3-мя элементами. Элемент с индексом `1` — это массив `[1]`. Берём от него первый элемент:

   ```
   [,[1],][1][0] == 1
   ```

   [js]

Решение задачи: JS-вопросник

Правильный ответ: пункт 4 (т.е. `NaN`):

```
var a = new Array(1,2), b = new Array(3);
alert(a[0] + b[0]);
```

Дело в том, что `new Array(1,2)` создаёт массив из элементов `[1,2]`, а вот `new Array(3)` — это особая форма вызова `new Array` с одним численным аргументом. При этом создаётся массив без элементов, но с длиной. Любой элемент этого массива равен `undefined`.

```
var a = new Array(1,2), b = new Array(3);
alert(a[0]); // 1
alert(b[0]); // undefined
```

При арифметических операциях `undefined` становится `NaN`, поэтому и общий результат — `NaN`.

Решение задачи: JS-вопросник

Правильный ответ: пункт 2 (`false`):

```
var applePrice = 1.15;
var peachPrice = 2.30;

var sum = 3.45;
alert( sum - (applePrice + peachPrice) == 0 ); // false
```

Дело в том, что из-за ошибок округления разность является не нулём, а очень маленьким числом:

```
var applePrice = 1.15;
var peachPrice = 2.30;

var sum = 3.45;
alert( sum - (applePrice + peachPrice) ); // число
```

Решение задачи: JS-вопросник

Правильный ответ: пункт **2** (выведет `1`):

```
var obj = {'1': 0, 1: 1, 0: 2};

alert(obj['1']);
```

При задании объекта через `{...}` — кавычки не обязательны, они нужны лишь в случаях, когда без них нельзя, например:

```
var obj = { "строка с пробелами" : 123 }
```

Все ключи приводятся к строке. Поэтому второй ключ равен первому и перезаписал его.

Решение задачи: JS-вопросник

Правильный ответ: пункт 4 (зависит от браузера):

Браузеры IE<9, Firefox, Safari
: Перебирают ключи в том же порядке, в котором свойства присваивались. Для них: `1`, `0`.

Opera, современный IE, Chrome
: Гарантируют сохранение порядка только для строковых ключей. Численные ключи сортируются и идут до строковых. Для них: `0`, `1`.

Попробуйте сами:

```
for(var key in  {1:1, 0:0}) {
 alert(key);
}
```

Решение задачи: JS-вопросник

Правильный ответ: пункт **1** (есть различия):

Для `x = NaN` (или любого другого значения, которое преобразуется к `NaN`):

```
var x = NaN;
alert( x <= 100 ); // false
alert( !(x > 100) ); // true
```

Решение задачи: JS-вопросник

Правильный ответ: **3** (ошибка).

Проверьте:

```
var f = function(x) {
  alert(x)
}  // (*)

(function() {
  f(1)
}())
```

Всё дело в том, что в коде отсутствует точка с запятой в месте `(*)`.

Поэтому JavaScript воспринимает его как:

```
var f = function(x) {
  alert(x)
}(function() { f(1) }())
```

То есть, хочет использовать `function() { f(1) }()` как аргумент для вызова первой функции.

Но что такое `function() { f(1) }()`? Это же вызов функции «на месте», который передаёт управление `f(1)`. А переменная `f` ещё не присвоена (равна `undefined`), поэтому её нельзя запустить и выдаётся ошибка об этом.

Решение задачи: JS-вопросник

Ответ: пункт `3`, выведет `4`.

```
var obj = {
 "0": 1,
 0: 2
};

alert( obj["0"] + obj[0] );
```

Дело в том, что у объектов в JavaScript ключи всегда строковые. Если в качестве ключа передано что-то ещё, то оно приводится к строке. Значения обрабатываются в порядке поступления, поэтому `0: 2` перекроет `"0": 1`.

В итоге получится объект с единственным ключом: `{"0" : 2}`.

При доступе к ключу — `obj["0"]` и `obj[0]` эквивалентны, опять же, поскольку ключ приводится к строке.

Решение задачи: JS-вопросник

Ответ: `4`.

```
alert( [] + false - null + true + undefined );
alert( [] + false - null + true );
```

Шаги преобразования:

```
[] + false = "false", т.к массив [] преобразуется к пустой строке, выходит "" + false = "false"
"false" - null = NaN, т.к. вычитание преобразует к числу, получается NaN - 0 = NaN
... дальнейшие вычисления дадут NaN
```

Решение задачи: JS-вопросник

Ответ: **1**.

Если точнее, результат равен `4`:

1. Первый минус превратит строку `"5"` в число `5`
2. Далее идёт сложение с числом 15, записанным в 16-ричной форме.
3. Далее строка содержит число, записанное в научной форме: `1e1 = 10`, минус преобразует эту строку в число.

Получается:

```
4 - 5 + 15 - 10 = 4
```

Ссылки

1. Стандарту http://www.w3.org/TR/html5/webappapis.html#timers
2. Управление порядком обработки, setTimeout(…0) http://learn.javascript.ru/events-and-timing-depth
3. Gmail http://gmail.com
4. Стандарту http://www.w3.org/TR/html5/webappapis.html#timers
5. Управление порядком обработки, setTimeout(…0) http://learn.javascript.ru/events-and-timing-depth
6. SetImmediate(func) http://msdn.microsoft.com/en-us/library/ie/hh773176.aspx
7. PostMessage https://developer.mozilla.org/en-US/docs/DOM/window.postMessage
8. Общение окон с разных доменов: postMessage http://learn.javascript.ru/cross-window-messaging-with-postmessage
9. MessageChannel http://www.w3.org/TR/webmessaging/#channel-messaging
10. Web Workers http://www.w3.org/TR/workers/
11. Bind https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/bind
12. Карринг http://ru.wikipedia.org/wiki/Каррирование
13. Concat http://javascript.ru/Array/concat
14. $.proxy http://api.jquery.com/jQuery.proxy/
15. Func.bind https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/bind
16. Es5-shim https://github.com/kriskowal/es5-shim
17. String.fromCharCode http://javascript.ru/String.fromCharCode
18. Date.parse http://javascript.ru/Date.parse

19. Date http://learn.javascript.ru/datetime
20. String.fromCharCode(code) http://javascript.ru/String.fromCharCode
21. ES5-shim https://github.com/kriskowal/es5-shim
22. Arr.forEach(callback[, thisArg]) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/forEach
23. Arr.filter(callback[, thisArg]) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/filter
24. Arr.map(callback[, thisArg]) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/map
25. Arr.every(callback[, thisArg]) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/every
26. Arr.some(callback[, thisArg]) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/some
27. Arr.reduce(reduceCallback[, initialValue]) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/reduce
28. Arr.reduceRight https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/reduceRight
29. ExecScript http://msdn.microsoft.com/en-us/library/ie/ms536420(v=vs.85).aspx
30. Является глобальный объект http://learn.javascript.ru/closures#scope-Function
31. Формат JSON http://learn.javascript.ru/json
32. ToString(основание системы счисления) https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Number/toString
33. ToString https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Number/toString
34. Error в MDN https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Error
35. Error в MSDN http://msdn.microsoft.com/en-us/library/dww52sbt(v=vs.85).aspx
36. Error http://javascript.ru/Error
37. JSON http://ru.wikipedia.org/wiki/JSON
38. RFC 4627 http://tools.ietf.org/html/rfc4627
39. JSON-js https://github.com/douglascrockford/JSON-js
40. Регулярных выражений http://learn.javascript.ru/regular-expressions-javascript