

# Основные понятия



Алексей  
Кулагин



# Алексей Кулагин

Технический руководитель, системный архитектор,  
разработчик в «Штрихпунктир»



[Алексей Кулагин](#)



# План занятия

1. [Операторы и операнды](#)
2. [Условные конструкции и циклы](#)
3. [Числа](#)
4. [Строки](#)
5. [Типы null и undefined](#)

---

## Вспомним то, что мы уже знаем

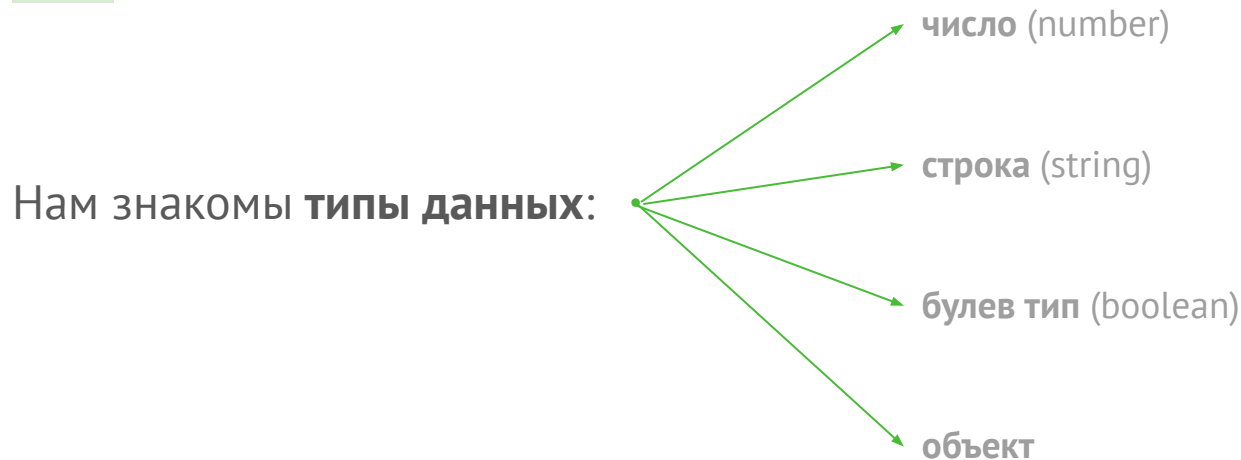


Несколько вопросов на повторение:

- что представляет из себя **переменная**, как её объявить?
- какие **типы данных** мы уже знаем?
- что представляет из себя **функция**?
- что представляет из себя **объект**?
- можно ли **объявить** две переменные с одинаковым именем?

# Переменные и типы данных

К данному моменту мы умеем объявлять переменные с помощью `let` и `const`.



Ещё мы знакомы с **массивами**, которые также являются объектами, *как и многое другое в js*

# JS — язык с динамической типизацией

JS называется языком с **динамической типизацией**, потому что переменные в JS могут менять свой тип в ходе выполнения кода.

Для определения типа используется оператор ***typeof***:

```
let name;  
console.log(typeof name); // "undefined"  
name = 'hello';  
console.log(typeof name); // "string"  
name = 17;  
console.log(typeof name); // "number"  
name = {};  
console.log(typeof name); // "object"  
name = true;
```

Тип переменной меняется на каждой строчке

---

# Переменные и типы данных

Давайте проверим ваши знания. Что делает **каждая строка кода**? Будут ли ошибки?

```
let name;  
name = 'Ann';  
let age = 17;  
let age = 20;  
let study = true;  
study = false;  
const country;  
country = 'Russia';  
const gender = 'female';  
gender = 'male';  
var city = 'Moscow';
```



# Переменные и типы данных

Давайте проверим ваши знания. Что делает **каждая строка кода**? Будут ли ошибки?

```
let name;      // Объявили
name = 'Ann';  // Инициализировали (строка)
let age = 17;  // Объявили и инициализировали (число)
let age = 20;  // Ошибка: нельзя объявить 2 переменных с одним именем
let study = true; // (булево значение)
study = false; // Присвоили другое значение
const country; // Ошибка: нельзя объявить константу без значения
country = 'Russia';
const gender = 'female'; // Правильное объявление констант
gender = 'male' // Ошибка: нельзя присваивать новые значения константам
var city = 'Владимир' // Устаревший способ объявления (подробнее в advanced)
```



# Константа и объекты

**Константы** накладывают ограничения **только** на присваивание значения, то есть оператор `=` и все производные: `+=`, `-=`, `*=` и так далее:

```
const friends = ['Маша, Таня, Оля, Вика']
friends.push('Константин')
console.log(friends); // ['Маша, Таня, Оля, Вика, Константин'];

const info = {
  city: 'Moscow',
  age: 31,
}
info.gender = 'male'
```

Переменная `friends` ссылается на тот же массив, что раньше. Присваивание не вызывалось, поэтому ошибка не произошла. Это касается и объекта: то есть мы можем менять внутреннее содержимое

# Требования к именам и правила

Имя может состоять из **букв, цифр, символов** `\$` и `\_`

Первый символ не должен быть **цифрой**

**Регистр** символов имеет значение. `Time`, `time` и `TIME` — разные переменные

**Зарезервированные ключевые слова** нельзя использовать в качестве имени переменной: `let`, `const` и другие

Используйте **английский язык** при выборе имени переменной или константы. Никакого транслита

Используйте **нижнийВерблюжийРегистр** (**lowerCamelCase**) для переменных, название которых состоит из нескольких слов. В нижнем верблюжьем регистре первое слово пишется с маленькой буквы. *Помимо соглашения camelCase также существуют kebab-case, snake\_case и другие, но в рамках данного курса мы будем использовать camelCase*

**Самое главное правило** — имя переменной должно чётко отражать информацию, которая хранится в переменной, и быть понятным не только автору кода



# Операторы и операнды

---

## Вспомним то, что мы уже знаем



- Какие операторы вы помните?

---

# Оператор и операнд



- Какие операторы вы помните?

**Оператор** — это команда, которая на письме выглядит как простой символ. С помощью операторов выполняются некоторые действия над данными.

Операторы бывают:

- **унарные** (англ. unary)
- **бинарные** (англ. binary)
- **тернарные** (англ. ternary)

Данные, расположенные справа и слева от оператора, называются **операндами**

---

# Унарный оператор

**Унарные** оперируют только левым или правым операндом:

```
let temp = -10;      // берет отрицательное число
let result = !false; // меняет true на false и обратно
let age = +'10';     // преобразует к числу
```

# Бинарный оператор

**Бинарные** оперируют левым и правым операндом:

- оператор **присвоения** `=`, `+=`, `-=`
- оператор **сложения** *либо конкатенации строк* `+`
- оператор **деления** `/`
- оператор **умножения** `*`
- оператор **вычитания** `-`
- оператор **взятия остатка** `%`
- операторы **сравнения** `>`, `<`, `>=`, `<=`, `!=`, `==`. *Строки сравниваются побуквенно*

# Нестрогое сравнение

**Нестрогое сравнение** `==`, `!=` использовать **крайне не рекомендуется**, так как это ведёт к множеству проблем: *происходит преобразование типов*. Вы можете использовать нестрогое сравнение только в том случае, если точно знаете, зачем оно вам нужно.

Таблица результатов нестрогого сравнения:

<https://dorey.github.io/JavaScript-Equality-Table>





# Парадоксы при нестрогом сравнении

```
[0]==0    //true  
0==[]     //true  
[]==' '   //true  
' '==[0] //false
```

## Вывод:

- большая таблица для запоминания
- большой потенциал для ошибки

# Тернарный оператор

Иногда нужно в зависимости от условия присвоить переменную:

```
условие ? значение1 : значение2
```

Как это работает?



# Тернарный оператор

Иногда нужно в зависимости от условия присвоить переменную:

```
let age = 18  
access = (age > 14) ? true : false;  
greeting = (age >= 18) ? "Здравствуйте" : "Привет";
```

В первом случае можно было бы обойтись и без оператора '?', так как сравнение само по себе уже возвращает true, false

---

Вопросительный знак – единственный оператор, у которого есть три аргумента, в то время как у обычных операторов их один–два. Поэтому его называют тернарным оператором

---

# Строковые шаблоны: `template literals`

**Шаблоны** — это строки, в которые можно подставить значения переменных.

Для этого используется новый тип кавычек и особый синтаксис:

```
` ${имяПеременной} `
```

Обратите внимание, что для шаблонов используются **апострофы** ```, а не одинарные кавычки `'`

# Строковые шаблоны: template literals

Допустим, у нас есть **задача рассчитать**, через какое время встретятся поезда, следующие навстречу друг другу:

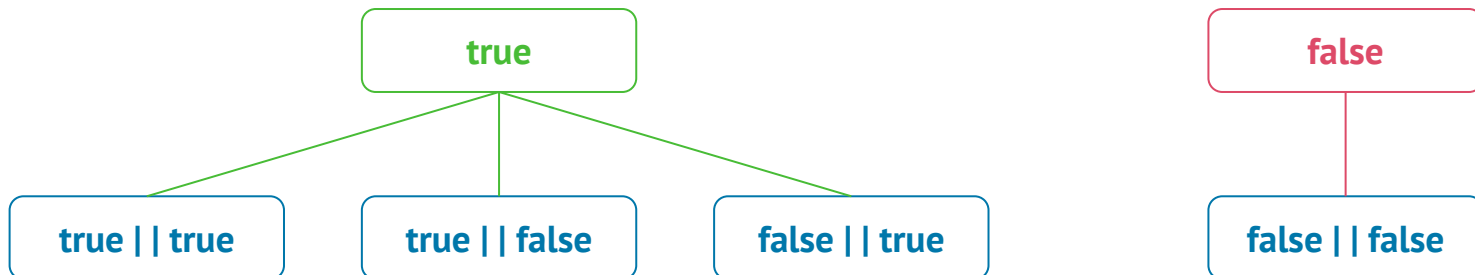
```
let train1Speed = 112;  
let train2Speed = 73;  
let distance = 1056;  
let time = distance / (train1Speed + train2Speed);  
let meetDistance = train1Speed * time;  
console.log(`Поезда встретятся через ${time} часов`);  
console.log(`За это время поезд 1 пройдет ${meetDistance} км`);
```

# Логические операторы

Оператор `||` (ИЛИ). Оператор вам уже знаком:

```
let milk = 60;  
let bread = 30;  
let fruits = 80;  
console.log(milk+bread < 70 || fruits <= 100);
```

Логическое **ИЛИ** в классическом программировании работает следующим образом: если хотя бы один из аргументов true, то возвращает true, иначе false:



# Логические операторы

**Оператор `||` (ИЛИ).** JavaScript вычисляет несколько **ИЛИ** слева направо. Чтобы экономить ресурсы, используется так называемый *короткий цикл вычисления*:

```
console.log(15 > 100 || false || 'some value' || true); // `some value`
```

Допустим, вычисляются несколько **ИЛИ** подряд: `a || b || c || ...`. Если первый аргумент `true`, то результат заведомо будет `true` (*хотя бы одно из значений – true*), и остальные значения игнорируются. Если все значения ложные, то `||` вернёт последнее из них

---

*Ложных значений всего 6, и они также доступны в таблице*

*(<https://dorey.github.io/JavaScript-Equality-Table/>)*

# Логические операторы

**Оператор && (И).** Оператор **И** вам уже знаком. К **нему** применим тот же принцип короткого цикла вычислений, но немного по-другому, чем к **ИЛИ**:

```
console.log(true && 'write me' && 10); // 10
console.log(10 && false && 'write me'); // false
```

Если левый аргумент — false, оператор **И** возвращает его и заканчивает вычисления. Иначе вычисляет и возвращает правый аргумент



# Логические операторы

**Оператор ! (НЕ).** Оператор **НЕ** — самый простой. Он получает один аргумент.  
Синтаксис:

```
let result = !value;
```

**Принцип действия:**

- сначала приводит аргумент к логическому типу true, false
- затем возвращает противоположное значение:

```
console.log(!true); // false  
console.log(!0); // true
```

# Логические операторы

**Оператор ! (НЕ).** Оператор *НЕ* — самый простой. Он получает один аргумент.  
Синтаксис:

```
let result = !value;
```

В частности, двойное *НЕ* используют для **преобразования значений** к логическому типу:

```
console.log(!!{name: "Vasia", age: 36}); // true
```

---

*Как происходит приведение к `true/false`, можно посмотреть в той же таблице  
(<https://dorey.github.io/JavaScript-Equality-Table/>)*



# Условные конструкции и циклы

# Условные конструкции

Вам уже знакома **условная конструкция**:

```
let age = +prompt('Сколько вам лет?');  
// prompt() показывает окно для ввода и возвращает строку  
// + преобразует к числу  
  
if (age >= 18) {  
    console.log('Доступ разрешен');  
} else if (age > 21) {  
    console.log('Вам даже в США можно все!');  
} else if (age > 60) {  
    console.log('Пенсия... хотя может быть и нет :)');  
} else {  
    console.log('Маловат еще');  
}
```

**Вопрос:** как вы думаете, что происходит?

# Конструкция и синтаксис switch

Конструкция **switch** заменяет собой сразу несколько **if**. Она представляет собой более наглядный способ сравнения выражения сразу с несколькими вариантами:

```
switch(x) {  
    case 'value1': // if (x ===  
        'value1')  
        ...  
        break;  
    case 'value2': // if (x ===  
        'value2')  
        ...  
        break;  
    default:  
        ...  
        break;  
}
```

- переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее
- если соответствие установлено, switch начинает выполняться от соответствующей директивы case и далее до ближайшего break *или до конца switch*
- если ни один case не совпал, выполняется, *если есть*, вариант default

# Конструкция switch. Группировка case (демо)

Несколько значений `case` можно **группировать**. В примере ниже `case 11` и `case 13` выполняют один и тот же код:

```
let a = 5+7;

switch (a) {
  case 12:
    alert('Верно!');
    break;

  case 11:
  case 13:
    alert('Неверно!');
    alert('Немного ошиблись, бывает. ');
    break;
  default:
    alert('Странный результат, очень странный. ');
}
```

# Циклы

Вспомним уже знакомые нам циклы *for* и *while*:

```
let result = 0, i = 1;
while (i < 5) {
  result += i;
  i++;
}
console.log(result);
```

```
let names = ["Sasha", "Katya", "Vika", "Maria"];
for (let i = 0; i < names.length; i = i+2) {
  console.log(names[i]);
}
```

# Прерывание цикла: break

**Выйти из цикла** можно не только при проверке условия, но и вообще в любой момент. Эту возможность обеспечивает директива **break**.

Например, задача — найти первое чётное число:

```
let numbers = [1, 23, 33, 46, 25, 13, 58];
for (let i = 0; i < numbers.length; i = i+2) {
  if(numbers[i] % 2 === 0 ) {
    console.log('Первое найденное четное число равно: ', numbers[i])
    break;
  }
}
```

*Вообще сочетание «бесконечный цикл + break» — отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале и конце цикла, а посередине*



## Следующая итерация: continue

Директива **continue** прекращает выполнение текущей итерации цикла. Она в некотором роде младшая сестра директивы **break**: **прерывает не весь цикл, а только текущее выполнение** его тела, как будто оно закончилось.

Её используют, если на текущем шаге цикла делать больше ничего не требуется:

```
let i = 0;
let result = 0;
while (i < 5) {
  i++;
  if (i === 3) {
    continue;
  }
  result += i;
}
```

*Суммируем числа от 0 до 5,  
пропуская 3*

---

## Доступ по значению и по ссылке: демо

**Вопрос:** что будет выведено в результате выполнения кода?

```
let a = 4;  
let b = a;  
b += 5;  
console.log(a, b);  
  
let c = {a: 1, b: "foo"};  
let d = c;  
d.a = false;  
console.log(c, d);
```

Какой будет результат?



# Доступ по значению и по ссылке: демо

**Вопрос:** что будет выведено в результате выполнения кода?

```
let a = 4;
let b = a;
b += 5;
console.log(a, b);

let c = {a: 1, b: "foo"};
let d = c;
d.a = false;
console.log(c, d);
```

**Ответ:**

```
> let a = 4;
   let b = a;
   b += 5;
   console.log(a, b);

4 9

< undefined

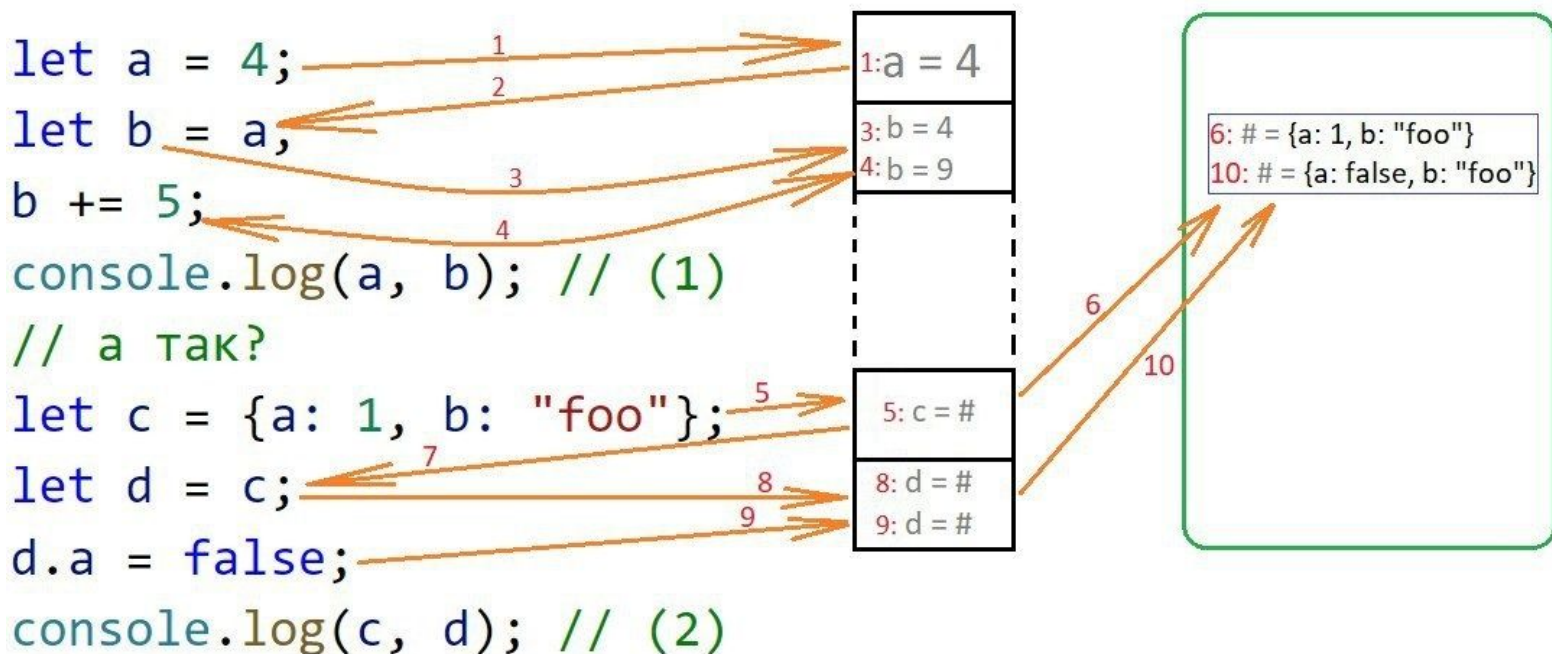
> let c = {a: 1, b: "foo"};
   let d = c;
   d.a = false;
   console.log(c, d);

{a: false, b: "foo"} {a: false, b: "foo"}
```

# Доступ по значению и по ссылке: демо

Вопрос: что будет выведено в результате выполнения кода?

Пояснения:





# Числа

# Числа

Все числа в JavaScript, как целые, так и дробные, имеют **тип Number** и хранятся в **64-битном формате**:

```
let a = 54;  
let b = 0xFF; // 255 в шестнадцатеричной системе  
let c = 3e5; // в научной форме:  $3 * 10^5$  т.е. 300000  
let d = 3e-5; // здесь  $3 * 10^{-5}$  0.00003
```

---

Дополнительная информация: [https://ru.wikipedia.org/wiki/IEEE\\_754-2008](https://ru.wikipedia.org/wiki/IEEE_754-2008)

---

# Неточные вычисления

Какое значение переменной *result* после вычисления?

```
let result = (0.1 + 0.2 == 0.3);
```

Какой будет результат?



# Неточные вычисления

Операции над числами могут привести к **неожиданным результатам**:

```
alert(0.1 + 0.2); // 0.30000000000000004  
alert(9999999999999999); // 10000000000000000
```

Всё дело в том, что в стандарте IEEE 754 на число выделяется ровно **8 байт** (*64 бита*) — не больше и не меньше



# Как избежать неточных вычислений?

Функция `toFixed(digits)` форматирует число, используя запись с фиксированной запятой и возвращает строку:

```
let a = (9.699932).toFixed(3)
```

“9.699” // обратите внимание, тип строка

---

*digits* — количество цифр после десятичной запятой. Может быть значением между 0 и 20 включительно, хотя реализации могут поддерживать и больший диапазон значений. Если аргумент опущен, он считается равным 0

---

# Округление чисел

Рекомендуется помнить следующие методы работы с числами с плавающей точкой:

```
Math.floor(); // Округляет в меньшую сторону  
Math.round(); // Округляет к ближайшему целому  
Math.ceil(); // Округляет в большую сторону  
Math.trunc(); // Убирает дробную часть
```

# Округление чисел. Пример

Что будет выведено на консоль?

```
console.log(Math.floor(1.1));  
console.log(Math.floor(1.9));
```

```
console.log(Math.round(3.1));  
console.log(Math.round(3.9));
```

```
console.log(Math.ceil(3.1));  
console.log(Math.ceil(3.9));
```

```
console.log(Math.ceil(-3.1));  
console.log(Math.ceil(-3.9));
```

Какой результат?



# Округление чисел. Пример

Что будет выведено на консоль?

```
console.log(Math.floor(1.1)); // 1  
console.log(Math.floor(1.9)); // 1
```

```
console.log(Math.round(3.1)); // 3  
console.log(Math.round(3.9)); // 4
```

```
console.log(Math.ceil(3.1)); // 4  
console.log(Math.ceil(3.9)); // 4
```

```
console.log(Math.ceil(-3.1)); // -3  
console.log(Math.ceil(-3.9)); // -3
```

---

# Особенные числовые значения

**Вопрос:** какое значение мы получим, если попытаемся какое-либо **число** разделить на ноль?

```
console.log(1/0);
```



# Значение Infinity

**Вопрос:** какое значение мы получим, если попытаемся какое-либо **число** разделить на ноль?

`console.log(1/0);` → Infinity

Это значение ведёт себя как **математическая бесконечность**.

Например, любое положительное число, умноженное на *Infinity*, даёт *Infinity*, а любое число, поделённое на *Infinity*, даёт 0

---

# Особенные числовые значения

**Вопрос:** какое значение мы получим, если попытаемся **строку умножить** или **разделить на число**?


```
console.log("десять" * 3);
```



# Особенные числовые значения NaN

**Вопрос:** какое значение мы получим, если попытаемся **строку умножить** или **разделить на число**?

```
console.log("десять" * 3);
```



```
NaN
```

**NaN** является значением, представляющим не-число (*Not-A-Number*).

**NaN** возникает, когда математические функции **не могут** вернуть значение, например, при вызове `Math.sqrt(-1)`, или когда функция считывания числа из строки **не может** это сделать, потому что в строке не-число (`parseInt('blabla')`)



# Особенные числовые значения NaN

```
console.log(typeof NaN); // "number"  
console.log(NaN === NaN); // false  
console.log(Number.isNaN(NaN)); // true
```

Обратите внимание, `NaN` — число.

Строгое сравнение `NaN === NaN` дает **false**. Немного неожиданное, но вполне объяснимое поведение.

Для проверки, что переменная — `NaN`, следует использовать `Number.isNaN()`



# Строки

# Строки

В JavaScript любые текстовые данные являются **строками**. Строки создаются при помощи двойных или одинарных кавычек:

```
let text = "это строка";  
let anotherText = 'ещё одна строка';  
let str = "012345";  
let result = `Привет, меня зовут ${name}, мне ${10+9} лет, я ${job}`);
```

# Специальные символы

Как решить проблему, если необходимо вставить символ, который **отсутствует на клавиатуре**, или необходимо добавить кавычки?

| Символ              | Описание                                      |
|---------------------|---|
| <code>\n</code>     | Перевод на новую строку                       |
| <code>\t</code>     | Символ табуляции                              |
| <code>\uNNNN</code> | Любой юникод символ с шестнадцатеричным кодом |
| <code>\'</code>     | Экранирование одинарной кавычки               |

# Специальные символы

Давайте посмотрим на пару примеров:

```
console.log('I\'m a JavaScript programmer'); // I'm a JavaScript programmer  
console.log('\u262D \u262A \u2766 \u2713 \u262F \u2328'); // 🍷 🌙 🍀 🍀 🍀 🍀
```

# Доступ к символам

**Строка является массивом символов**, следовательно, можно получить символ или воспользоваться методом `charAt()`:

```
let myString = "моя строка";  
console.log(myString[4]); // 'с'  
console.log(myString.charAt(4)); // 'с'
```

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| м | о | я |   | с | т | р | о | к | а |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

## Различия между обращением как к массиву и функцией charAt()

Предпочтительным способом доступа к символам является `string[i]`.

На практике почти всегда так. Использование `charAt` может привести к неожиданным результатам: <https://stackoverflow.com/a/54827850/2546720>

```
let myString = "моя строка";  
console.log(myString[30]); // undefined  
console.log(myString.charAt(30)); // ""
```

# Длина строки

Одно из самых частых действий со строкой — получение её длины. **Длина строки** находится в свойстве `length`:

```
let myString = "моя строка";  
console.log(myString.length); // 10
```

---

*Если нам необходимо отправить значение на сервер для сохранения в базе данных, стоит задуматься о валидации этого значения, чтобы оно не было слишком длинным*



---

# Смена регистра

Методы `toLowerCase()` и `toUpperCase()` меняют регистр строки на нижний, верхний.

Если необходимо **сравнить** ввод пользователя с ключевой фразой без учёта регистра:

```
console.log("моя строка".toUpperCase()); // "МОЯ СТРОКА"  
console.log("Моя СтРоКа".toLowerCase()); // "моя строка"
```

# Поиск строки в строке

Для **поиска строки в строке** есть метод:

```
indexOf(искомая_строка[, начальная_позиция])
```

Он возвращает позицию, на которой находится подстрока, или **-1**, если ничего не найдено:

```
console.log("моя строка".indexOf("о")); // 1  
console.log("Моя СтРоКа".indexOf("о", 4)); // 7
```

---

## Взятие подстроки

В JavaScript существуют **3 метода для взятия подстроки** с небольшими отличиями между ними: `substring`, `substr` и `slice`. Наиболее универсальным и современным, а также применимым к массивам, является `slice`



# Использование slice

```
str.slice(beginIndex[, endIndex])
```

- **beginIndex** — индекс, с которого нужно начинать извлечение. Если аргумент отрицателен, то трактуется как «с конца». Если **beginIndex** больше или равен **str.length**, возвращается пустая строка
- **endIndex** — индекс, перед которым нужно заканчивать извлечение.  
*Нумерация при этом начинается с нуля.* Символ по этому индексу не будет включён. Если **endIndex** опущен или является **undefined**, **slice()** извлечёт всё до конца строки. Если аргумент отрицательный, то трактуется как «с конца»

---

# Использование slice

```
const str = 'The quick brown fox jumps over the lazy dog.';

console.log(str.slice(31));
// expected output: "the lazy dog."

console.log(str.slice(4, 19));
// expected output: "quick brown fox"

console.log(str.slice(-4));
// expected output: "dog."

console.log(str.slice(-9, -5));
// expected output: "lazy"
```

# Сравнение символов

Символы **сравниваются** в алфавитном порядке `'А' < 'Б' < 'В' < ... < 'Я'`, но с некоторыми особенностями:

```
console.log('а' > 'Я'); // true
console.log('ё' > 'я'); // true, т.к. ё находится после строчных
```

Все строки имеют **внутреннюю кодировку Unicode**. По этому коду сравниваются строки

# Сравнение строк

Сравнение строк работает **лексикографически**, т. е. посимвольно.

При вводе капчи необходимо **проверить** пользовательский ввод с правильным результатом.

Сравнение строк `str1` и `str2` обрабатывается по следующему **алгоритму**:

- сравниваются первые символы: `str1[0]` и `str2[0]`. Если они разные, то сравниваем их и, в зависимости от результата их сравнения, возвращаем true или false. Если же они одинаковые, то

# Сравнение строк

Сравнение строк работает **лексикографически**, т. е. посимвольно.

При вводе капчи необходимо проверить пользовательский ввод с правильным результатом.

Сравнение строк `str1` и `str2` обрабатывается по следующему **алгоритму**:

- Сравниваются первые символы: `str1[0]` и `str2[0]`
- Сравниваются вторые символы `str1[1]` и `str2[1]`



# Сравнение строк

Сравнение строк работает **лексикографически**, т. е. посимвольно.

При вводе капчи необходимо проверить пользовательский ввод с правильным результатом.

Сравнение строк `str1` и `str2` обрабатывается по следующему **алгоритму**:

- Сравниваются первые символы: `str1[0]` и `str2[0]`
- Сравниваются вторые символы `str1[1]` и `str2[1]`
- Затем третьи `str1[2]` и `str2[2]` и так далее, пока символы не будут наконец разными, и тогда какой символ больше — та строка и больше

# Сравнение строк

Сравнение строк работает **лексикографически**, т. е. посимвольно.

При вводе капчи необходимо проверить пользовательский ввод с правильным результатом.

Сравнение строк `str1` и `str2` обрабатывается по следующему **алгоритму**:

- Сравниваются первые символы: `str1[0]` и `str2[0]`
- Сравниваются вторые символы `str1[1]` и `str2[1]`
- Затем третьи `str1[2]` и `str2[2]` и так далее
- Если же в какой-либо строке закончились символы, то считаем, что она меньше, а если закончились в обеих — они равны

# Сравнение строк с учётом языка

Правильным способом сравнения строк с учётом алфавита языка сравнения является `str.localeCompare(str2,[locale])`:

```
console.log('ä'.localeCompare('z', 'de'));  
// отрицательное значение: в немецком буква ä идёт перед буквой z  
  
console.log('ä'.localeCompare('z', 'sv'));  
// положительное значение: в шведском буква ä следует после буквы z  
  
console.log('Привет!'.localeCompare('Андрей', 'ru'));  
// положительное сравнение
```

# Получение символа по коду и код по символу

Метод `String.fromCharCode(code)` возвращает символ по коду:

```
console.log(String.fromCharCode(8381)); // ₽
```

Метод `str.charCodeAt(pos)` возвращает код символа на позиции pos:

```
console.log("строка".charCodeAt(0)); // 1089, код 'с'
```



# Типы undefined и null

---

# Вспоминаем прошлые занятия

**Вопрос:** чему равно **значение по умолчанию** объявленной переменной?



# Когда используется undefined

Переменная, **не имеющая присвоенного значения**, обладает значением **undefined**.

Также возвращают **undefined** **метод** или **инструкция**, если переменная, участвующая в вычислениях, не имеет присвоенного значения. **Функция** возвращает **undefined**, если она не возвращает какое-либо значение:

```
let a; // undefined
let b = console.log(2+2); // выведет 4
console.log(b); // выведет undefined
```

---

*В явном виде undefined **никогда** не присваивают, так как это противоречит его смыслу. Для записи в переменную пустого или неизвестного значения используется null*

# Логические сравнения undefined

Если переменная вообще **не существует**, можно ли к ней обратиться? *Нет:*

```
typeof x === 'undefined'; // для переменной, не определенной ранее, вернет true
x === undefined; // выведет ReferenceError
```

```
let x;
x === undefined; // true
```



---

## Тип null

Мы используем тип **null**, если хотим показать **отсутствие значения**.

Переменная имеет значение *null*, если в ней нет явного значения



# Сравнение null с нулём

Сравним `null` с нулём:

```
null > 0 // false
null == 0 // false
null >= 0 // true
null == undefined // true
```

Дело в том, что алгоритмы проверки равенства `==` и сравнения `>=`, `>`, `<`, `<=` работают по-разному

# Сравнение null с нулём

Сравним `null` с нулём:

```
null > 0 // false
null == 0 // false
null >= 0 // true
null == undefined // true
```

Дело в том, что алгоритмы проверки равенства `==` и сравнения `>=`, `>`, `<`, `<=` работают по-разному:

- **сравнение** честно приводит к числу, получается ноль

# Сравнение null с нулём

Сравним `null` с нулём:

```
null > 0 // false
null == 0 // false
null >= 0 // true
null == undefined // true
```

Дело в том, что алгоритмы проверки равенства `==` и сравнения `>=`, `>`, `<`, `<=` работают по-разному:

- **сравнение** честно приводит к числу, получается ноль
- при проверке **равенства** значения `null` и `undefined` обрабатываются особым образом: они равны друг другу, но не равны чему-то ещё

# Сравнение null с нулём

Сравним `null` с нулём:

```
null > 0 // false
null == 0 // false
null >= 0 // true
null == undefined // true
```

**Вывод:** сравнивайте данные одинаковых типов. Избегайте нестрогого равенства

---

# Отличия между null и undefined

```
typeof null // object (баг в JS, должно быть null)
typeof undefined // undefined
null === undefined // false
null == undefined // true
```

# Преобразования типов

Предположим, что у вас есть **строка**, в которой хранится число, к которому необходимо прибавить число. Как это сделать?

```
console.log("13" + 6);
```

Какой результат?



## Преобразования типов

Предположим, что у вас есть **строка**, в которой хранится число, к которому необходимо прибавить число. Как это сделать?

Ответ:

```
console.log("13" + 6);
```

```
"136"
```

Почему получилось **136**, а не **19**?



# Почему получилось 136, а не 19?

При **сложении строк** движок JavaScript преобразует все параметры к строке и складывает их:

```
console.log("13" + 6); // "13" + "6" = "136"  
console.log("мой " + "дом"); // "мой дом"
```

Это пример **неявного преобразования типов**.

Решение проблемы — **явное преобразование типов**

# Явное преобразования типов

Значение можно явно **преобразовать** из одного типа в другой:

```
//К числу
console.log(parseFloat('3.14')); // 3.14
console.log(parseInt("0xF", 16)); //15
console.log(Number("13") + 6); // 19
console.log(+ "3.13"); //3.13
```

```
//К строке
console.log(String(123)); // "123"
console.log(''+12345); // "12345"
```

```
// К булеву
console.log(Boolean(123)); // true
console.log(!!123); // true
```

# Проблемы с преобразованием к числу

Отличия при преобразовании строки в число с помощью `Number` и `parseInt`:

```
console.log(Number("25abc") + 7);  
// NaN, упс... Вызов Number не примет нечисловые строки  
  
console.log(parseInt("25abc") + 7); // 32, сработало
```

# Новые типы

```
let s = new Symbol("symbol_name"); // Symbol
const theBiggestInt = 9007199254740991n; // BigInt
const alsoHuge = BigInt(9007199254740991); // BigInt
const hugeHex = BigInt("0x1ffffffffffffff"); // BigInt
```

---

*В реальной практике эти типы используются очень редко, поэтому они выходят за рамки данного курса*

## Запуск из Node.js

А вы знали, что JavaScript-код **можно запускать** не только в браузере?

Для запуска кода в среде **Node** следует выполнить команду:

```
`node ./main.js` (./main.js - путь к исполняемому файлу)
```

Если файл находится **в текущей директории**, то можно использовать:

```
`node main.js`.
```



---

## Дополнительные материалы

- [Symbol](#)
- [Symbol на learn.javascript.ru](#)
- [BigInt](#)
- [BigInt — новый тип данных в JS](#)
- [String.prototype.substr\(\)](#)
- [String.prototype.substring\(\)](#)



# Итоги



---

## Чему мы научились?

- Переменные и ограничения, связанные с их применением
- Операторы, операнды, понятия и назначение
- Новые типы данных и преобразования типов



---

# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#):

- вопросы по домашней работе задавайте **в чате** мессенджера Slack
- задачи можно сдавать **по частям**
- зачёт по домашней работе проставляется после того, как **приняты все обязательные задачи**

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Алексей Кулагин**