

Путь к карьере Frontend Fullstack разработчика

Модуль 1. WEB CORE

Уровень 18. Функции, массивы и объекты.
Часть 1.



Преобразование типов в JavaScript

В JavaScript, как и во многих других языках программирования, данные имеют различные типы. Часто нам приходится преобразовывать данные из одного типа в другой. Например, когда пользователь вводит свой возраст в форму, это значение по умолчанию будет строкой. Но для математических операций нам нужно, чтобы это было число.

Приведение типов (или преобразование) — это процесс изменения типа данных. В JavaScript существуют два основных способа преобразования типов:

- явное (ручное)
- неявное (автоматическое)

Сравнение типов:

- **Строгое равенство (`===`):** Сравнивает значения и типы.
- **Нестрогое равенство (`==`):** Сравнивает значения, при необходимости преобразуя типы.

Явное преобразование типов

Это процесс, при котором мы вручную указываем JavaScript, какой тип данных мы хотим получить в результате. Это позволяет нам точно контролировать процесс преобразования и избежать неожиданных результатов, которые могут возникнуть при неявном преобразовании.

Преобразование в строку:

- `String(value)`: Универсальный метод, преобразующий любой тип данных в строку.
- `value.toString()`: Метод объекта, который также преобразует в строку. Однако у некоторых объектов (например, `null` и `undefined`) он может быть недоступен или вести себя неожиданно.

```
let num = 42;
let str = "100";
let bool = true;

// Преобразование в строку
let numAsString = String(num); // "42"
let boolAsString = bool.toString(); // "true"
```

Преобразование в число

- `Number(value)`: Преобразует значение в число. Если преобразование невозможно, возвращает `NaN`.
- `parseInt(value, base)`: Преобразует строку в целое число, указав основание системы счисления.
- `parseFloat(value)`: Преобразует строку в число с плавающей точкой.

```
let num = 42;
let str = "100";
let bool = true;

// Преобразование в число
let strAsNumber = Number(str); // 100
let boolAsNumber = Number(bool); // 1

let str = "123";
let num = parseInt(str, 10); // "number"

let str = "123.45";
let num = parseFloat(str); // "number"
```

Особенности:

- Пустая строка преобразуется в `0`.
- Строка, начинающаяся с числа, преобразуется в число.
- Строка, начинающаяся с нечислового символа, преобразуется в `NaN`.

Преобразование в логическое значение

Boolean(value): Преобразует значение в логическое. Пустые строки, 0, null, undefined и NaN преобразуются в **false**, все остальные значения - в **true**.

```
Boolean('') // false
Boolean('string') // true
Boolean('false') // true
Boolean(0) // false
Boolean(42) // true
Boolean(-42) // true
Boolean(NaN) // false
Boolean(null) // false
Boolean(undefined) // false
Boolean({}) // true
Boolean({ key: 42 }) // true
```

Неявное преобразование типов

Это автоматический процесс, когда JavaScript преобразует значения из одного типа в другой в зависимости от контекста. Хотя это удобно, оно может привести к неожиданным результатам, если не понимать, как оно работает.

Сложение со строками: Любое значение, сложенное со строкой, будет преобразовано в строку.

```
let num = 42;  
let result = "Результат: " + num;  
// result будет равно "Результат: 42"
```

Арифметические операции: При выполнении арифметических операций (кроме сложения) строки, содержащие числа, будут преобразованы в числа.

```
let strNum = "100";  
let result = strNum - 50; // result будет равно 50
```

Логические контексты: В условных выражениях, циклах и других логических контекстах значения преобразуются в логические.

```
let userInput = ""; // Пустая строка  
  
if (!userInput) {  
    console.log("Пользователь не ввел данные.");  
// Неявное преобразование строки в булево значение  
}
```

Строгое и нестрогое равенство

Неявное преобразование также используется, когда мы сравниваем значения через нестрогое равенство `==`. В отличие от строгого равенства (`===`), в нём интерпретатор пробует привести типы к одному, чтобы сравнить.

Хорошей практикой считается использовать только строгое сравнение, чтобы избежать неявного преобразования типов при сравнении.

Есть большая матрица, которая показывает, «что чему равно» при строгом и нестрогом равенстве.

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

Хранение данных в JavaScript

В JavaScript, переменные хранят не только значения, но и ссылки на них.

```
// Прimitives types (storage by value)
let num1 = 10;
let num2 = num1;
num2 = 20;
console.log(num1); // Outputs 10

// Reference types (storage by reference)
let array1 = [1, 2, 3];
let array2 = array1;
array2.push(4);
console.log(array1); // Outputs [1, 2, 3, 4]
```

Хранение по значению:

- **Примитивы:** Числа, строки, булевы, null, undefined.
- **Копирование:** При присвоении создается точная копия значения.
- **Изменения:** Изменение одной переменной не влияет на другую.

Хранение по ссылке:

- **Объекты, массивы, функции.**
- **Ссылка:** Переменная хранит адрес в памяти, где находится объект.
- **Изменения:** Изменение объекта через одну переменную влияет на все переменные, ссылающиеся на него.

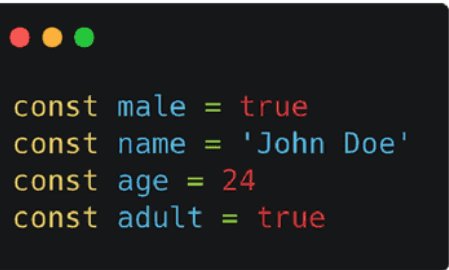
Как хранятся примитивные типы в памяти?

Представьте примитивное значение как небольшую коробочку, в которой лежит конкретный предмет. Когда вы создаете переменную и присваиваете ей примитивное значение, вы создаете новую коробочку и кладете в нее этот предмет.

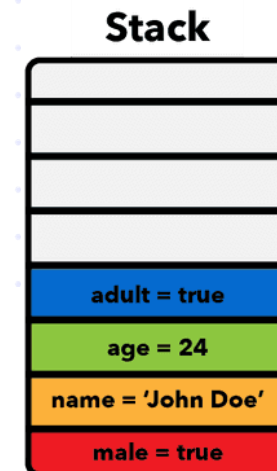
Если вы потом скопируете значение в другую переменную, вы создаете еще одну коробочку и кладете в нее точную копию предмета из первой коробки.

При сравнении двух примитивных значений сравниваются сами значения. Если значения равны, то результат сравнения будет true.

```
let x = 5;  
let y = 5;  
console.log(x === y); // true
```

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner.

```
const male = true  
const name = 'John Doe'  
const age = 24  
const adult = true
```



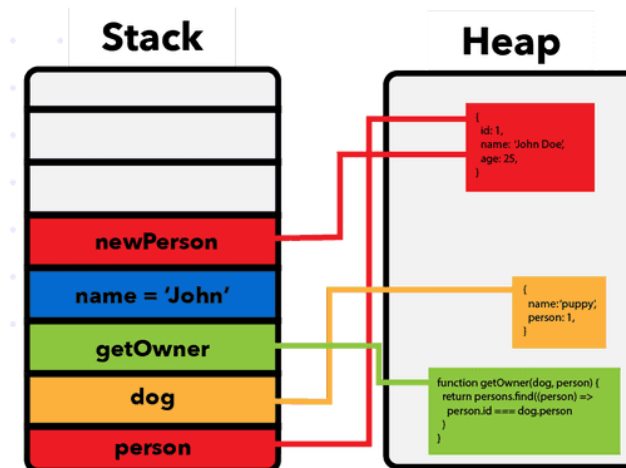
Ссылочные типы данных

В отличие от примитивных типов, которые хранят непосредственно значение, ссылочные типы хранят ссылку на место в памяти, где находится это значение.

Представьте, что у вас есть ящик, в котором лежит не сам предмет, а записка с адресом, где этот предмет находится. Когда вы хотите получить предмет, вы идете по этому адресу. Точно так же переменная, содержащая ссылку на объект, указывает на место в памяти, где хранится этот объект.

```
let person = { name: 'Alice', age: 30 };  
// Создаем еще одну ссылку на тот же объект  
let anotherPerson = person;  
  
anotherPerson.age = 31;  
  
console.log(person);  
// { name: 'Alice', age: 31 }  
console.log(anotherPerson);  
// { name: 'Alice', age: 31 }
```

```
const person = {  
  id: 1,  
  name: 'John',  
  age: 25,  
}  
  
const dog = {  
  name: 'puppy',  
  personId: 1,  
}  
  
function getOwner(dog, persons) {  
  return persons.find((person) =>  
    person.id === dog.personId  
  )  
}  
  
const name = 'John';  
  
const newPerson = person;
```



Тип `undefined`

`undefined` - это примитивный тип данных в JavaScript, который указывает на отсутствие присвоенного значения. Другими словами, если переменная была объявлена, но ей еще не присвоено конкретное значение, то ее значение будет `undefined`.

Когда возникает `undefined`:

```
// Несуществующие свойства объектов:  
let obj = {};  
console.log(obj.property); // undefined  
  
// Несуществующие элементы массивов:  
const numbers = [1, 2, 3];  
console.log(numbers[5]); // undefined
```

Важные моменты:

- `undefined` - это значение, а не ошибка.
- Тип данных: `typeof undefined` возвращает `'undefined'`.
- Сравнение: `undefined` строго равен только самому себе: `undefined === undefined`.

Тип `null`

Тип `null` представляет намеренное отсутствие значения. Это значение часто используется для инициализации переменных, которые будут позже присвоены объектом, или для очистки значения переменной.

Когда использовать `null`:

- **Инициализация переменных:** Когда вы хотите создать переменную, но пока не знаете, какое значение ей присвоить.
- **Очистка значений:** Когда вы хотите явно указать, что переменная больше не ссылается на объект.
- **Возврат из функций:** Когда функция не должна возвращать никакого значения.

```
// Инициализация
let user = null;

// Очистка
user = { name: 'Alice' };
user = null;

// Возврат функции
function getUser() {
    // Если пользователь не найден, возвращаем null
    return null;
}

// Проверка условий
if (user === null) {
    console.log('Пользователь не найден');
}
```

Объявление функций в JavaScript

Функция — это блок кода, который выполняет определенную задачу. Она может принимать входные данные (аргументы) и возвращать результат. Такие функции «поднимаются» (hoisted) на вершину их области видимости, что позволяет их вызывать до объявления в коде.

Функция объявляется с помощью ключевого слова **function**, за которым следует имя функции, список аргументов в круглых скобках и тело функции в фигурных скобках.

```
function name (arguments) {  
    // function body  
}
```

- **name**: Имя функции, по которому она вызывается.
- **arguments**: Список параметров, которые функция принимает.
- **function body**: Блок кода, который выполняется при вызове функции.

```
// Объявление функции  
function greet(name) {  
    return `Hello, ${name}!`;  
}  
  
// Вызов функции  
console.log(greet('Alice')); // Выведет: Hello, Alice!
```

Выражения функций

Выражения функций создают анонимные функции (функции без имени) и могут быть присвоены переменной. Такие функции не «всплывают», поэтому их можно вызвать только после объявления. Это означает, что вы должны объявить выражение функции перед его использованием.

```
const nameФункции = function(параметры) {  
  // тело функции  
};
```

```
// Объявление функции  
function greet(name) {  
  return `Hello, ${name}!`;  
}  
  
// Выражение функции  
const greet2 = function(name) {  
  return `Hi, ${name}!`;  
};
```

Что такое функции-стрелки?

Функции-стрелки — это компактный синтаксис для создания функций в JavaScript. Они были введены в ES6 и стали популярны благодаря своей лаконичности и особенностям контекста **this**.

Ключевые особенности:

- **Контекст **this**:** Функции-стрелки не имеют собственного **this**. Они наследуют **this** из окружающего лексического окружения. Это делает их особенно полезными в обработчиках событий и методах массивов.
- **ИмPLICITНЫЙ ВОЗВРАТ:** Если тело функции состоит из одного выражения, то результат этого выражения автоматически возвращается.
- **Краткая запись:** Для функций с одним параметром можно опустить круглые скобки, а для функций с одним выражением — фигурные скобки и ключевое слово **return**.

```
const name = (arguments) => {  
  // function body  
}
```

```
// Полная запись  
const square = (x) => {  
  return x * x;  
};  
  
// Сокращенная запись  
const square2 = x => x * x;  
  
// Без явного возврата  
const greet = name => `Hello, ${name}!`;
```

Параметры функций

Параметры функций — это переменные, которые определяются при объявлении функции и используются для приема данных, передаваемых при вызове функции.

Параметры указываются в круглых скобках после имени функции. При вызове функции значения этих параметров передаются в соответствующие переменные внутри функции.

```
function sum(a, b) {  
    return a + b;  
}  
  
const result = sum(3, 5); // result будет равен 8
```

Обработка неопределенных параметров:

Если при вызове функции какой-то параметр не указан, его значение будет **undefined**. Это может привести к ошибкам. Чтобы избежать этого, можно использовать проверку на **undefined**.

В функции **greet()** проверяется, передан ли параметр **name**. Если нет, используется строка **Guest**.

```
function greet(name) {  
    if (name === undefined) {  
        return 'Hello, Guest!';  
    }  
    return `Hello, ${name}!`;  
}  
  
console.log(greet()); // "Hello, Guest!"  
console.log(greet('Bob')); // "Hello, Bob!"
```


Параметры по умолчанию

Параметры по умолчанию позволяют задавать значения для параметров функции, которые будут использоваться, если при вызове функции эти параметры не были переданы или были переданы со значением **undefined**.

```
// Простая функция с параметром по умолчанию
function greet(name = 'Guest') {
    return `Hello, ${name}!`;
}

// Параметры по умолчанию, зависящие от других параметров
function createProduct(name, price, discount = price * 0.1) {
    return { name, price, finalPrice: price - discount };
}

// Параметры по умолчанию, зависящие от других параметров:
function createProduct(name, price, discount = price * 0.1) {
    return { name, price, finalPrice: price - discount };
}
```

Важные моменты:

- Значения по умолчанию вычисляются только один раз, при определении функции.
- Параметры с значениями по умолчанию должны идти после параметров без значений по умолчанию.
- Значения по умолчанию можно переопределить при вызове функции.

Остаточные параметры

Остаточные параметры (rest parameters) — это синтаксическая конструкция в JavaScript, которая позволяет собирать все оставшиеся аргументы функции в массив.

Это особенно полезно, когда количество аргументов заранее неизвестно или когда нужно разделить аргументы на две группы: фиксированные и переменное количество остальных.

```
function name (arg1, arg2, ...argArray) {  
  // function body  
}
```

Как работают остаточные параметры:

- Остаточные параметры всегда должны быть последними в списке параметров функции.
- Все аргументы, передаваемые после фиксированных параметров, собираются в массив, который присваивается переменной, обозначенной тремя точками (...).

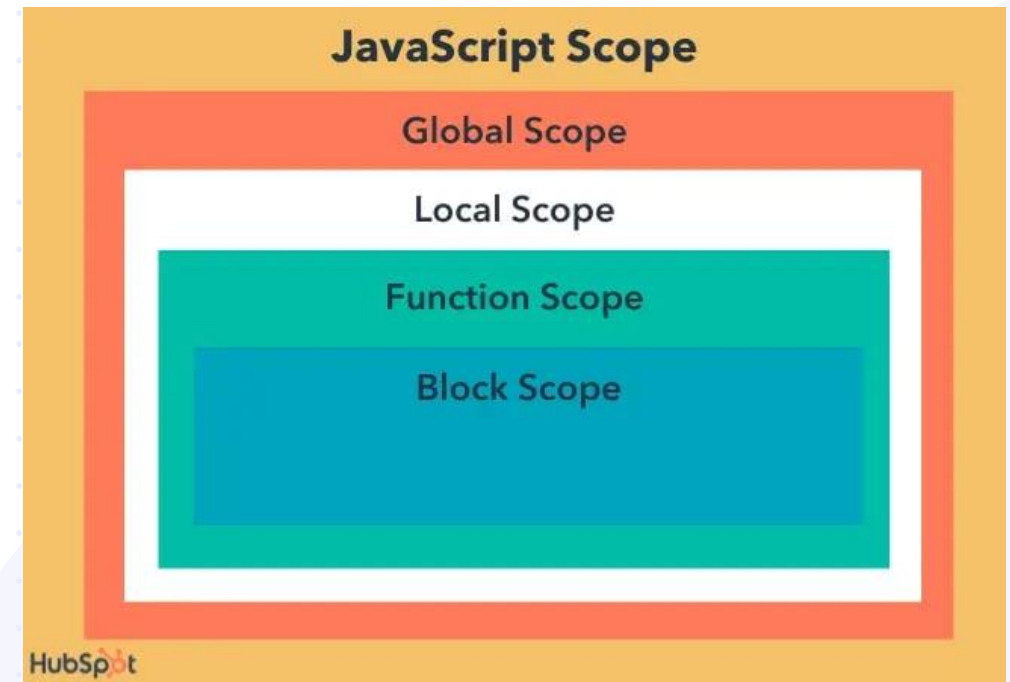
```
function sumAll(...numbers) {  
  return numbers.reduce((sum, num) => sum + num, 0);  
}  
  
console.log(sumAll(1, 2, 3));           // Выведет: 6  
console.log(sumAll(4, 5, 6, 7, 8));    // Выведет: 30
```

Области видимости

Области видимости (scopes) – это часть программы, в которой мы можем обратиться к переменной, функции или объекту.

Этой частью может быть функция, блок или вся программа в целом — то есть мы всегда находимся как минимум в одной области видимости.

Области видимости помогают скрывать переменные от нежелательного доступа, управлять побочными эффектами и разбивать код на смысловые блоки.



Global Scope

Глобальная область видимости включает в себя все переменные и функции, объявленные вне каких-либо функций или блоков кода. Эти переменные и функции доступны из любого места в коде.

```
// Глобальная переменная
let message = 'Hello, world!';

function greet(name) {
  console.log(message + ' ' + name); // Доступ к глобальной переменной
}

greet('Alice'); // Выведет: Hello, world! Alice
```

Local Scope

Локальная область видимости – это пространство, ограниченное функцией. Переменные, объявленные внутри функции, доступны только внутри этой функции и не видны снаружи.

Это помогает избежать конфликтов имен и делает код более организованным.

```
function greet(name) {  
    let message = "Hello, "; // Локальная переменная  
    console.log(message + name + "!");  
}  
  
greet("Alice"); // Выведет: Hello, Alice!  
  
console.log(message); // Ошибка: message не определена
```

Локальная область видимости может быть разделена на область видимости функции и область видимости блока. С ES6 введены `let` и `const`, которые создают переменные с блочной областью видимости (например, внутри блоков `if`, `for`).

Function Scope

Область видимости функции ограничивает доступ к переменным на уровне функций, в которых они были объявлены. В JavaScript, переменные, объявленные с использованием ключевого слова **var** внутри функции, не доступны за пределами этой функции.

Переменная **num** объявлена внутри функции **foo** и доступна только в этой области видимости.

```
function foo() {  
    var num = 10; // Локальная переменная функции  
    console.log('inside function:', num);  
}  
  
foo(); // Вывод в консоль: inside function: 10  
console.log(num); // Ошибка: ReferenceError: num is not defined
```

Block Scope

Область видимости блока означает, что переменные, объявленные при помощи ключевых слов `let` и `const`, контролируются фигурными скобками `{}` таких конструкций, как условные инструкции (например, `if`, `switch`) или циклы (`for`, `while`).

Это ограничивает доступ к этим переменным строго внутри данных блоков.

```
function testScope() {  
  if (true) {  
    let blockScopedVar = 'Доступна только в этом блоке';  
    console.log(blockScopedVar); // Вывод: 'Доступна только в этом блоке'  
  }  
  // Попытка доступа к blockScopedVar за пределами блока вызовет ошибку  
  console.log(blockScopedVar); // Ошибка: ReferenceError: blockScopedVar is not defined  
}  
  
testScope();
```

Lexical Scope

В JavaScript функции создаются вместе с их лексическим окружением. Это означает, что каждая функция "запоминает" окружение, в котором она была создана. Таким образом, при вызове функции она имеет доступ к переменным из своей лексической области видимости, включая переменные во всех родительских областях.

В примере внутренняя функция `inner()` имеет доступ к переменной `outerVar`, объявленной в родительской функции `outer()`. Когда функция `inner()` вызывается внутри `outer()`, она выводит значение переменной `outerVar`.

```
function outer() {  
    let outerVar = 'Я переменная из внешней функции (outer)';  
  
    function inner() {  
        // Доступ к переменной внешней функции достигим из внутренней функции  
        console.log(outerVar); // Вывод: 'Я переменная из внешней функции (outer)'  
    }  
  
    inner();  
}  
  
outer();
```


Динамическая область видимости

Динамическая область видимости позволяет переменным быть доступными в зависимости от того, как функция была вызвана, что может приводить к разным результатам в зависимости от контекста вызова функции.

Лексическая область видимости (Lexical Scope) определяется во время написания кода и зависит от структурной организации программы. Динамическая область видимости (Dynamic Scope), в отличие от лексической, определяется в момент выполнения программы, исходя из цепочки вызовов функций.

```
function foo() {  
    console.log(a); // Вывод зависит от типа области видимости  
}  
  
function bar() {  
    var a = 10;  
    foo();  
}  
  
var a = 5;  
bar(); // Лексическая область видимости: выводит 5
```

Домашнее задание

Уровень 18. Функции, массивы и объекты.
Лекции 0-4

