**1. Типы данных**

a. Какие типы данных существуют в JavaScript и как они классифицируются?  
В JavaScript есть 8 типов данных:

* Примитивные: **number**, **string**, **boolean**, **null**, **undefined**, **symbol**, **bigint**
* Объектный тип: **object** (включает массивы, функции, даты и другие сложные структуры)

b. Что такое примитивные типы данных в JavaScript и какие они бывают?  
Примитивные типы — это простейшие значения, которые не являются объектами и не имеют методов. Всего их семь:  
**number**, **string**, **boolean**, **null**, **undefined**, **symbol**, **bigint**.

c. Каким образом можно проверить тип данных переменной в JavaScript?  
Для этого используется оператор **typeof**. Например:

javascript

let x = 42;

console.log(typeof x); // "number"

d. Что такое объекты в JavaScript и как они отличаются от примитивных типов данных?  
Объекты — это коллекции данных и более сложные структуры. Они могут содержать свойства и методы.  
В отличие от примитивов, объекты хранятся по ссылке, а не по значению.

e. Каким образом можно создавать объекты в JavaScript и какие особенности у них есть?  
Создание через литерал:

javascript

let user = { name: 'John', age: 30 };

Или через конструктор **new Object()**:

javascript

let user = new Object();

Особенности: гибкая структура, поддержка методов, хранение данных по ссылке.

f. Каким образом работает оператор typeof в JavaScript и какие значения он может возвращать?  
Оператор **typeof** возвращает строку с названием типа данных. Возможные результаты:  
**"number"**, **"string"**, **"boolean"**, **"object"**, **"function"**, **"undefined"**, **"symbol"**, **"bigint"**.

g. Какой тип данных имеет значение null в JavaScript и каким образом он отличается от undefined?  
Значение **null** имеет тип **"object"** (это ошибка языка). Оно указывает на намеренно пустое значение.  
**undefined** означает, что значение не было назначено или переменная не определена.

**2. Приведение типов**

a. Что такое приведение типов в JavaScript и как оно работает?  
Приведение типов — это процесс преобразования одного типа данных в другой. Может быть явным (программистом) или неявным (движком JS).

b. Каким образом можно преобразовать строку в число в JavaScript и наоборот?  
Явное преобразование:

javascript

Number("123"); // строка в число

String(123); // число в строку

Неявное:

javascript

"123" \* 1; // к числу

123 + ""; // к строке

c. Каким образом можно преобразовать значение в логический тип данных в JavaScript?  
Через **Boolean()** или двойное отрицание:

javascript

Boolean(""); // false

!!"hello"; // true

d. Что такое неявное приведение типов в JavaScript и когда оно происходит?  
Это автоматическое преобразование типов при выполнении операций. Происходит, например, при использовании оператора **+** со строкой и числом.

e. Как работает приведение типов при использовании оператора "+" в JavaScript?  
Если один из операндов — строка, то второй также приводится к строке:

javascript

1

1 + "2" => "12"

Если оба числа — обычное сложение.

f. Каким образом можно проверить тип данных и избежать ошибок при приведении типов в JavaScript?  
Использовать строгую проверку (**===**, **typeof**) и избегать неявного преобразования там, где важно поведение.

g. Какие проблемы могут возникать при неявном приведении типов в JavaScript и как их избежать?  
Проблемы:

* Сравнение **==** может давать неожиданные результаты (например, **0 == ""** → **true**).  
  Решение: использовать строгое сравнение **===**.

**3. Области видимости**

a. Что такое область видимости в JavaScript и как она работает?  
Область видимости — это контекст выполнения, в котором переменные и функции доступны для использования. JS использует лексическую область видимости.

b. Каким образом можно создавать области видимости в JavaScript?  
Через объявление функций или блоков кода **{}** (начиная с ES6, с использованием **let** и **const**).

c. Как работает механизм лексического окружения в JavaScript и как он связан с областями видимости?  
Лексическое окружение — это внутренняя структура, которая хранит переменные и ссылается на внешнее окружение. Это основа работы областей видимости.

d. Что такое глобальная область видимости в JavaScript и как она работает?  
Глобальная область видимости — это уровень вне всех функций и блоков. Переменные, объявленные здесь, доступны везде.

e. Каким образом можно создавать локальные области видимости в JavaScript?  
Через функции или блоки кода:

javascript

function example() {

let local = "local";

}

f. Что такое замыкание в JavaScript и как оно связано с областями видимости?  
Замыкание — это функция, которая имеет доступ к переменным внешней функции даже после её завершения. Это возможно благодаря областям видимости.

**4. Взаимодействие с DOM**

a. Что такое DOM в JavaScript и как он работает?  
DOM (Document Object Model) — это представление HTML-документа в виде дерева объектов, с которым можно взаимодействовать с помощью JS.

b. Каким образом можно получить доступ к элементам DOM в JavaScript?  
Методы:

javascript

document.getElementById("id");

document.querySelector(".class");

document.querySelectorAll("div");

c. Как работает механизм обратных вызовов (callback) в JavaScript и как он используется при работе с DOM?  
Callback — это функция, переданная в другую функцию и вызванная позже. Используется часто при обработке событий:

javascript

button.addEventListener("click", function() {

alert("Clicked!");

});

d. Каким образом можно изменять свойства элементов DOM в JavaScript?  
Например:

javascript

element.style.color = "red";

element.textContent = "New text";

element.setAttribute("src", "image.jpg");

e. Как работает механизм событий в JavaScript и как он используется при работе с DOM?  
События — это сигналы, возникающие при действиях пользователя (клик, нажатие клавиши и т.д.). С ними работают через обработчики событий.

f. Каким образом можно регистрировать обработчики событий на элементах DOM в JavaScript?  
Через **addEventListener**:

javascript

1

element.addEventListener("click", handlerFunction);

g. Как работает механизм делегирования событий в JavaScript и как он может помочь при работе с большим количеством элементов DOM?  
Делегирование — это установка одного обработчика на родителя, который обрабатывает события всех потомков. Удобно для динамических списков.

h. Каким образом можно создавать и удалять элементы DOM в JavaScript?  
Создание:

javascript

1

let div = document.createElement("div");

Добавление:

javascript

1

parent.appendChild(div);

Удаление:

javascript

1

parent.removeChild(div);

**5. Обработка событий**

a. Что такое событие в JavaScript и как оно обрабатывается?  
Событие — это действие, происходящее в браузере (клик, загрузка и т.д.). Обрабатывается через обработчики, зарегистрированные на элементах.

b. Каким образом можно зарегистрировать обработчик события в JavaScript?  
С помощью **addEventListener**:

javascript

1

element.addEventListener("click", function() {});

c. Как работает механизм всплытия событий в JavaScript и как он может повлиять на обработку событий?  
После срабатывания события внутри элемента, оно всплывает вверх по дереву DOM. Можно остановить через **event.stopPropagation()**.

d. Каким образом можно отменить всплытие события в JavaScript?  
Использовать метод **event.stopPropagation()** внутри обработчика.

e. Как работает механизм делегирования событий в JavaScript и как он может упростить обработку событий?  
Делегирование позволяет обрабатывать события на уровне родителя, что удобно для динамически добавляемых элементов.

f. Каким образом можно использовать параметры события в обработчике событий в JavaScript?  
Передаваемый объект **event** содержит информацию о событии:

javascript

element.addEventListener("click", function(event) {

console.log(event.target);

});

**6. Замыкания**

a. Что такое замыкание в JavaScript и как оно работает?  
Замыкание — это функция, которая имеет доступ к переменным внешней функции даже после того, как эта функция завершилась.

b. Каким образом можно создавать замыкания в JavaScript?  
Создать функцию внутри другой функции и вернуть её:

javascript

function outer() {

let count = 0;

return function() { return ++count; }

}

c. Как работает механизм доступа к внешним переменным из замыкания в JavaScript?  
Замыкание сохраняет ссылку на внешнюю область видимости и может читать/изменять переменные из неё.

d. Каким образом можно использовать замыкания для создания приватных свойств и методов в JavaScript?  
Скрытие переменных внутри функции:

javascript

⌄

function Counter() {

let count = 0;

this.increment = function() { return ++count; }

}

e. Каким образом можно передавать замыкания в качестве аргументов в функции JavaScript?  
Как обычную функцию:

javascript

⌄

function doSomething(callback) {

callback();

}

doSomething(function() { console.log("Done"); });

f. Как работает механизм возврата замыканий из функций в JavaScript и как он может быть полезен?  
Функция возвращает внутреннюю функцию, которая "запоминает" окружение. Полезно для реализации счетчиков, модулей и других шаблонов.