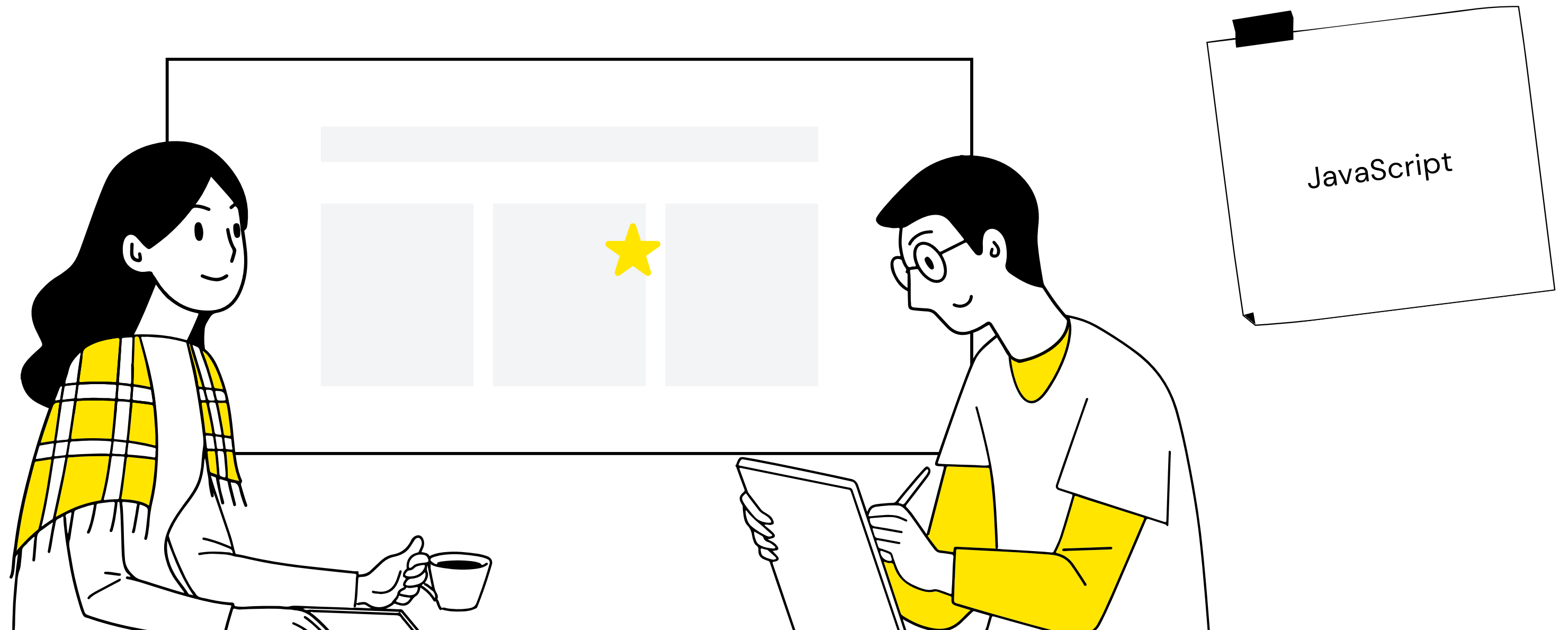


# JavaScript



# План

1

Объекты

2

Копирование объектов и  
ссылки

3

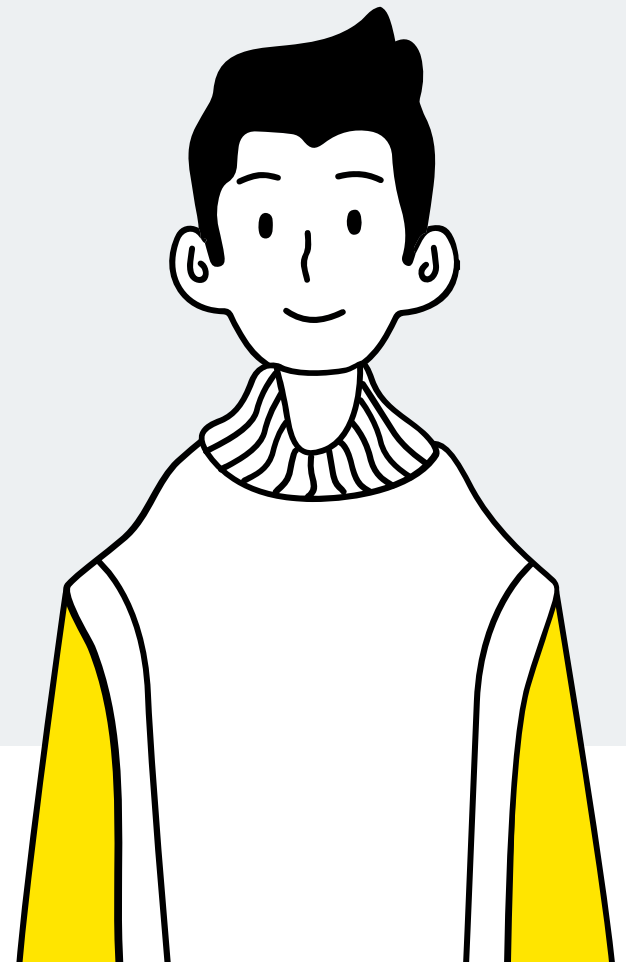
Опциональная цепочка

4

Методы объекта, this

# Объекты

Объекты в JavaScript представляют собой совокупности пар ключ-значение, которые позволяют нам хранить и управлять данными. Объекты могут содержать разные типы значений, включая строки, числа, массивы, функции и другие объекты. Объекты могут быть простыми или сложными, в зависимости от того, какие данные они содержат и как они используются.



# Объекты

Основная структура объекта в JavaScript выглядит следующим образом:

В этом примере `myObject` – это объект, содержащий четыре пары ключ-значение. Каждый ключ уникален в рамках объекта и связан с определенным значением. Значение может быть любого типа, включая функции и массивы, как показано в примере.

```
let myObject = {  
  key1: 'value1',  
  key2: 'value2',  
  key3: function() {  
    // do something  
  },  
  key4: ['item1', 'item2', 'item3']  
};
```

# Объекты

Вы можете получить доступ к значениям объекта с помощью точечной нотации:

```
console.log(myObject.key1); // outputs: 'value1'
```

Или с помощью скобочной нотации:

```
console.log(myObject['key1']); // outputs: 'value1'
```

Вы также можете изменять значения, используя ту же нотацию:

```
myObject.key1 = 'new value';  
console.log(myObject.key1); // outputs: 'new value'
```

# Удаление свойства

Оператор delete в JavaScript используется для удаления свойства из объекта. После его использования свойство больше не существует.

```
let obj = {  
  name: 'John',  
  age: 30  
};  
  
console.log(obj.name); // 'John'  
  
delete obj.name;  
  
console.log(obj.name); // undefined
```

В этом примере свойство name удаляется из объекта obj, и когда мы пытаемся обратиться к нему, возвращается undefined.

# Проверка существования свойства

Оператор `in` позволяет проверить, существует ли определенное свойство в объекте. Это работает даже если значение свойства `undefined` или `null`

```
let obj = {  
  name: 'John',  
  age: null  
};  
  
console.log('name' in obj); // true  
console.log('age' in obj); // true  
console.log('nonexistent' in obj); // false
```

В этом примере мы видим, что `'name' in obj` и `'age' in obj` возвращают `true`, потому что эти свойства существуют в объекте, даже если `age` равен `null`. Однако `'nonexistent' in obj` возвращает `false`, потому что такого свойства нет в объекте.

# Перебор свойств объекта: цикл for..in

Оператор for...in в JavaScript позволяет перебрать все свойства объекта (включая те, что наследуются от прототипа, если они есть). Это может быть полезно, если вы хотите выполнить действие для каждого свойства объекта.

```
let obj = {  
  name: 'John',  
  age: 30,  
  city: 'New York'  
};  
  
for (let key in obj) {  
  console.log(`Key is: ${key}, value is: ${obj[key]}`);  
}
```

В этом примере мы перебираем каждое свойство в объекте obj и выводим имя свойства и его значение.



# Объекты

JavaScript также поддерживает более сложные объекты, такие как встроенные объекты `Date`, `RegExp` и другие. Он также поддерживает создание собственных объектов с помощью конструкторов и прототипов, что позволяет разработчикам создавать сложные структуры данных и переиспользовать код.

Важно помнить, что в JavaScript почти все является объектами. Даже функции являются объектами и могут содержать свои собственные свойства и методы. Это делает JavaScript очень гибким языком, который может быть использован для создания широкого спектра веб-приложений.



# Копирование объектов

**Копирование по ссылке vs копирование по значению:** В JavaScript, объекты копируются по ссылке, а не по значению. Это означает, что когда вы копируете объект и изменяете его, изменения отражаются на всех копиях этого объекта, потому что все они ссылаются на одно и то же место в памяти. С другой стороны, примитивные типы данных, такие как строки, числа и булевы значения, копируются по значению.

# Копирование объектов

Примитивные типы данных в JavaScript включают строки, числа, булевы значения, null и undefined. Когда вы копируете примитивные типы, вы фактически создаете новую копию значения.

```
1 let a = 10;
2 let b = a; // b получает копию значения a
3
4 a = 20; // изменяем значение a
5
6 console.log(a); // выведет 20
7 console.log(b); // выведет 10, потому что b сохранил копию исходного значения a
```

# Копирование объектов

Объекты в JavaScript включают объекты, массивы и функции. Когда вы копируете объекты, вы фактически копируете ссылку на объект, а не сам объект.

```
let obj1 = { name: 'John' };
let obj2 = obj1; // obj2 получает ссылку на obj1

obj1.name = 'Pete'; // изменяем свойство объекта

console.log(obj1.name); // выведет 'Pete'
console.log(obj2.name); // также выведет 'Pete', потому что obj2 ссылается на тот же объект, что и obj1
```

В этом примере, когда мы меняем свойство name в obj1, это изменение видно и в obj2, потому что обе переменные ссылаются на один и тот же объект.

# Сравнение объектов

**Сравнение объектов:** Два объекта считаются равными только в том случае, если они ссылаются на одно и то же место в памяти. Даже если два объекта имеют одинаковую структуру и значения, они не будут равны, если они не ссылаются на одно и то же место в памяти.



# Сравнение объектов

В этом примере, даже если `obj1` и `obj2` имеют одинаковые свойства и значения, они считаются разными объектами, потому что они ссылаются на разные места в памяти.

```
let obj1 = { name: 'John' };  
let obj2 = { name: 'John' };  
  
console.log(obj1 == obj2); // выведет false  
console.log(obj1 === obj2); // выведет false
```

# Сравнение объектов

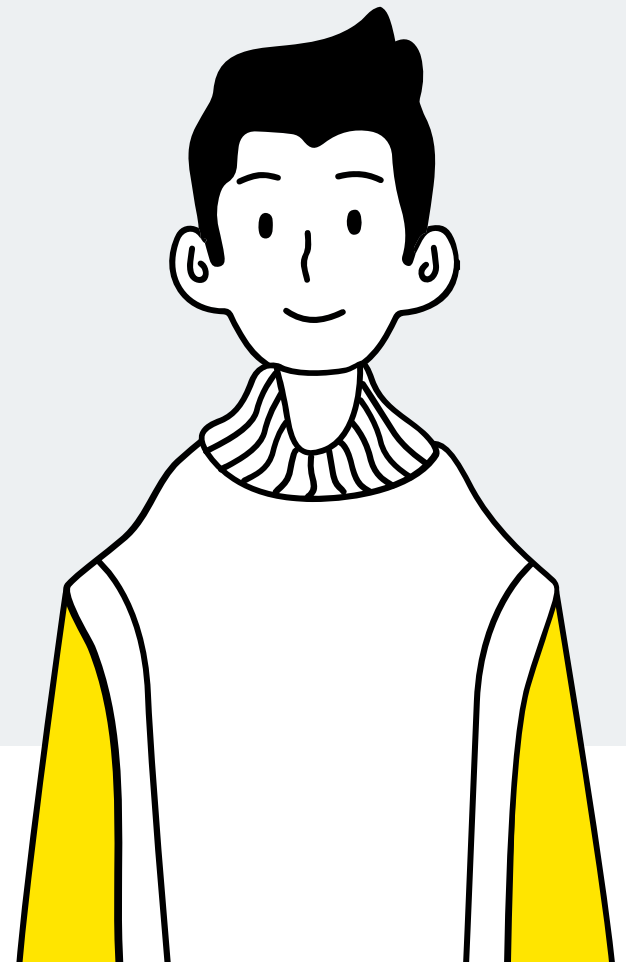
Теперь давайте рассмотрим случай, когда две переменные ссылаются на один и тот же объект:

```
let obj1 = { name: 'John' };  
let obj2 = obj1; // теперь obj2 ссылается на тот же объект, что и obj1  
  
console.log(obj1 == obj2); // выведет true  
console.log(obj1 === obj2); // выведет true
```

В этом примере, поскольку obj1 и obj2 ссылаются на одно и то же место в памяти, они считаются равными.

# Объекты, объявленные как константы

В JavaScript, когда вы объявляете объект как константу с помощью ключевого слова `const`, вы не можете изменить саму константу (то есть, вы не можете заставить её ссылаться на другой объект), но вы можете изменить свойства этого объекта.





# Объекты, объявленные как константы

В этом примере, хотя `obj` объявлен как константа, мы все равно можем изменить его свойство `name`.

```
const obj = { name: 'John' };


console.log(obj.name); // выведет 'John'

obj.name = 'Pete'; // изменяем свойство объекта

console.log(obj.name); // теперь выведет 'Pete'
```

# Объекты, объявленные как константы

Однако, если вы попытаетесь изменить саму константу `obj`, вы получите ошибку:

```
const obj = { name: 'John' };  
  
obj = { name: 'Pete' }; // TypeError: Assignment to constant variable.
```

Этот код вызовет ошибку, потому что мы пытаемся изменить саму константу `obj`, что недопустимо в JavaScript.

# Клонирование объектов

Если вам нужно создать полностью независимую копию (клон) объекта, вы должны создать новый объект и скопировать в него все свойства исходного объекта. В JavaScript нет встроенного метода для этого, но вы можете использовать цикл `for...in` или метод `Object.assign()`.



# Клонирование объектов

Использование цикла for...in:

```
let user = {  
  name: "John",  
  age: 30  
};  
  
let clone = {}; // новый пустой объект  
  
// скопируем все свойства user в clone  
for (let key in user) {  
  clone[key] = user[key];  
}  
  
// теперь clone - это полностью независимый клон объекта user  
clone.name = "Pete"; // изменяем данные в clone  
  
console.log(user.name); // 'John' - свойства в исходном объекте не изменились  
console.log(clone.name); // 'Pete' - свойства в клонированном объекте изменились
```

# Клонирование объектов

Использование метода `Object.assign()`:

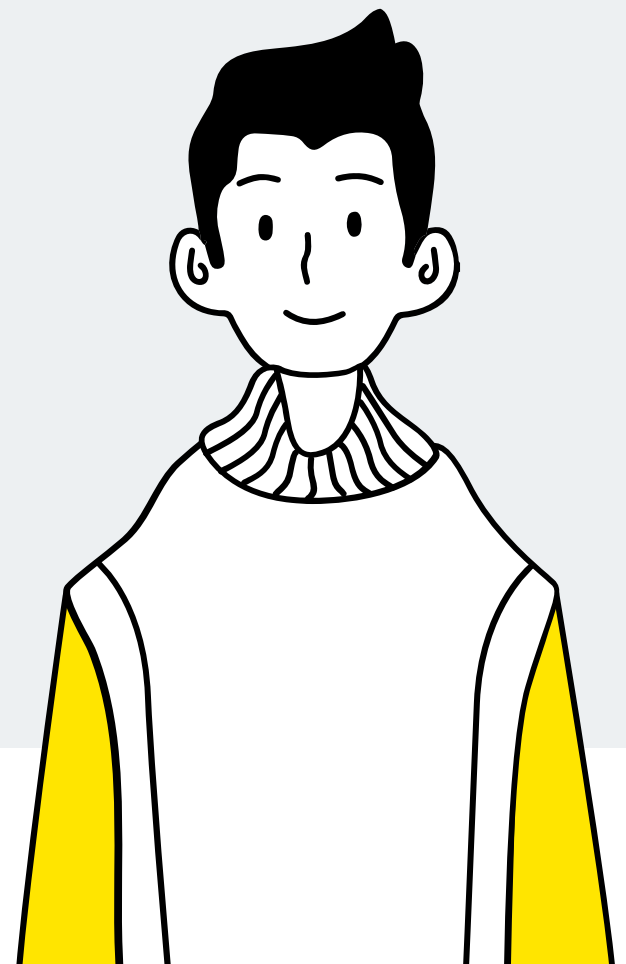
```
let user = {  
  name: "John",  
  age: 30  
};  
  
let clone = Object.assign(target: {}, user);  
  
// теперь clone - это полностью независимый клон объекта user  
clone.name = "Pete"; // изменяем данные в clone  
  
console.log(user.name); // 'John' - свойства в исходном объекте не изменились  
console.log(clone.name); // 'Pete' - свойства в клонированном объекте изменились
```

# Клонирование объектов

Все эти методы создают "поверхностные" клоны объекта. Если исходный объект имеет вложенные объекты, то они будут копироваться по ссылке, а не создавать новые объекты. Для создания "глубоких" клонов (когда вы хотите скопировать и вложенные объекты) можно использовать функции из различных библиотек, таких как Lodash (функция `_cloneDeep()`).

# Методы объекта

Методы объекта – это функции, которые находятся в свойствах объекта. Они обычно представляют действия, которые объект может выполнять.  
Например:



# Методы объекта

В этом примере sayHi является методом объекта user.

```
let user = {  
  name: "John",  
  age: 30,  
  sayHi() {  
    alert("Привет!");  
  }  
};  
  
user.sayHi(); // Привет!
```



# Ключевое слово this

Ключевое слово `this` в методах обычно ссылается на объект, которому принадлежит метод. Это позволяет методам получить доступ к свойствам и методам своего объекта. Например:

```
let user = {  
  name: "John",  
  age: 30,  
  sayHi() {  
    alert(this.name);  
  }  
};
```

```
user.sayHi(); // John
```

# this в разных контекстах

В JavaScript, поведение 'this' может отличаться в зависимости от контекста, в котором оно используется. Например, в стрелочных функциях 'this' не имеет своего собственного значения и будет брать значение из окружающего контекста.

```
let user = {  
  name: "John",  
  greet() {  
    console.log(`Hello, my name is ${this.name}`);  
  }  
};  
  
user.greet(); // Выводит: "Hello, my name is John"
```

В этом примере this в методе greet ссылается на объект user.

# this в разных контекстах

В глобальном контексте: Когда this используется вне любого объекта, оно ссылается на глобальный объект. В браузере глобальный объект – это window.

```
console.log(this === window); // Выводит: true
```

# this в разных контекстах

**В стрелочной функции:** Стрелочные функции не имеют своего собственного this. Вместо этого, this в стрелочной функции берется из окружающего лексического контекста.

```
let user = {  
  name: "John",  
  greet: () => {  
    console.log(`Hello, my name is ${this.name}`);  
  }  
};  
  
user.greet(); // Выводит: "Hello, my name is undefined"
```

В этом примере this в стрелочной функции не ссылается на объект user, как это было бы в обычной функции. Вместо этого, this берется из окружающего контекста, который в данном случае является глобальным объектом, и this.name не определено в глобальном объекте.

# this в разных контекстах

```
function Person(name) {  
  this.name = name;  
  
  this.sayHello = function() {  
    // Стрелочная функция наследует контекст `this` от обычной  
    let arrowFunction = () => {  
      console.log(`Hello, ${this.name}`);  
    };  
  
    arrowFunction();  
  };  
}  
  
let john = new Person( name: 'John');  
john.sayHello(); // Выводит: "Hello, John"
```

В этом примере, Person – это конструктор объекта, который создает объект с именем. У объекта есть метод sayHello, который создает стрелочную функцию arrowFunction и затем вызывает ее. Поскольку стрелочные функции не имеют своего собственного this, они наследуют this из окружающего контекста. В этом случае, окружающий контекст – это функция sayHello, поэтому this в стрелочной функции ссылается на this в функции sayHello. Таким образом, когда мы создаем новый объект john и вызываем john.sayHello(), мы видим вывод "Hello, John".

# Опциональная цепочка '?.'

Опциональная цепочка ?. — это безопасный способ доступа к свойствам вложенных объектов, даже если какое-либо из промежуточных свойств не существует.

# Опциональная цепочка '?.'

Основная проблема, которую решает опциональная цепочка, это обращение к свойствам объекта, которые могут быть не определены. Например, если у вас есть объект `user`, который может иметь или не иметь свойство `address`, и вы попытаетесь получить доступ к `user.address.street`, вы получите ошибку, если `address` не определено.

Опциональная цепочка позволяет вам безопасно обращаться к вложенным свойствам. Вместо `user.address.street` вы можете написать `user?.address?.street`. Если `address` не определено, выражение вернет `undefined`, а не вызовет ошибку.

# Опциональная цепочка '?.'

Опциональная цепочка также работает с вызовами функций и обращениями к элементам массива.

Например, `user.sayHello?.()` вызовет функцию `sayHello`, только если она определена. `user.skills?.[0]` вернет первый элемент массива `skills`, только если он определен.

Однако стоит быть осторожным с использованием опциональной цепочки. Она полезна, когда вы не уверены в структуре объекта, но если вы знаете, что определенное свойство должно быть определено, лучше не использовать опциональную цепочку, чтобы не пропустить ошибки.



# Опциональная цепочка '?.'

Обращение к свойствам объекта:

```
let user = {}; // пользователь без свойства "address"  
console.log(user?.address?.street); // undefined, без ошибки
```

В этом примере у нас есть объект `user`, который не имеет свойства `address`. Без опциональной цепочки, попытка обратиться к `user.address.street` вызвала бы ошибку, потому что мы пытаемся получить доступ к свойству `street` у `undefined`. Однако, с использованием опциональной цепочки, выражение `user?.address?.street` просто возвращает `undefined`, потому что `address` не определено.

# Опциональная цепочка '?.'

Вызов методов:

```
let user = {  
  greet: function() {  
    console.log("Hello!");  
  }  
};  
  
user.greet?(); // "Hello!"  
user.sayBye?(); // undefined, без ошибки
```

В этом примере у нас есть объект `user` с методом `greet`. Мы можем безопасно вызвать этот метод с использованием опциональной цепочки `user.greet?()`. Если бы мы попытались вызвать несуществующий метод `sayBye` без опциональной цепочки, мы получили бы ошибку. Но с опциональной цепочкой `user.sayBye?()` просто возвращает `undefined`.

# Опциональная цепочка '?.'

Обращение к элементам массива:

```
let users = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 }
];

console.log(users[0]?.name); // "Alice"
console.log(users[2]?.name); // undefined, без ошибки
```

В этом примере у нас есть массив `users` с двумя объектами. Мы можем безопасно получить доступ к свойствам этих объектов с использованием опциональной цепочки. `users[0]?.name` возвращает "Alice", а `users[2]?.name` возвращает `undefined`, потому что `users[2]` не определен.

Это основные примеры использования опциональной цепочки в JavaScript. Она очень полезна для работы с объектами и массивами, когда вы не уверены в их структуре или когда некоторые свойства могут быть не определены.

# Thank you!

