1. **Деструктурирующее присваивание**

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array. Объекты позволяют создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда они передаются в функцию, то ей может понадобится не объект/массив целиком, а элементы по отдельности.

*Деструктурирующее присваивание* – это специальный синтаксис, который позволяет преобразовать массивы или объекты в кучу переменных, так как иногда они более удобны. Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию, и так далее.

[**Деструктуризация массива**](https://learn.javascript.ru/destructuring-assignment#destrukturizatsiya-massiva)

Пример:

let arr = ["Steve", "Jobs"]

// firstName=arr[0], surname=arr[1]

let [firstName, surname] = arr;

alert(firstName); // Steve

alert(surname);  // Jobs

Теперь можно использовать переменные вместо элементов массива. Удобно использовать в сочетании со split или другими методами, возвращающими массив:

let [firstName, surname] = "Steve Jobs".split(' ');

 Деструктурирующее присваивание не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.

Ненужные элементы массива также могут быть отброшены через запятую:

// второй элемент не нужен

let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul

В примере выше второй элемент массива пропускается, а третий присваивается переменной title, оставшиеся элементы массива также пропускаются (так как для них нет переменных).

Можно использовать любой перебираемый объект, не только массивы:

Если надо не просто получить первые значения, но и собрать все остальные – мы можем добавить ещё один параметр, который получает остальные значения, используя троеточие "...":

let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius

alert(name2); // Caesar

alert(rest[0]); // Consul

alert(rest[1]); // of the Roman Republic

alert(rest.length); // 2

Переменная rest является массивом из оставшихся элементов. Вместо rest можно использовать любое другое название переменной, и она должна находится на последнем месте в деструктурирующем присваивании.

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения считаются неопределёнными:

let [firstName, surname] = [];

alert(firstName); // undefined

alert(surname); // undefined

Если необходимо указать значения по умолчанию, то можно использовать =:

let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (из массива)

alert(surname); // Anonymous (значение по умолчанию)

Значения по умолчанию могут быть гораздо более сложными выражениями или даже функциями. Они выполняются, только если значения отсутствуют. Например, в в примере ниже используется функция prompt для указания двух значений по умолчанию. Но она будет запущена только для отсутствующего значения:

let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name);    // Julius (из массива)

alert(surname); // результат prompt

[**Деструктуризация объекта**](https://learn.javascript.ru/destructuring-assignment#destrukturizatsiya-obekta)

Деструктурирующее присваивание также работает с объектами. Имена переменных и ключи должны совпадать. Синтаксис:

let {var1, var2} = {var1:…, var2:…}

У нас есть существующий объект с правой стороны, который необходимо разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в {...}. Например:

let options = {

  title: "Menu",

  width: 100,

  height: 200

};

let {title, width, height} = options;

alert(title);  // Menu

alert(width);  // 100

alert(height); // 200

Свойства options.title, options.width и options.height присваиваются соответствующим переменным. Порядок не имеет значения. Вот так тоже работает:

// изменён порядок в let {...}

let {height, width, title} = { title: "Menu", height: 200, width: 100 }

Шаблон с левой стороны может быть более сложным и определять соответствие между свойствами и переменными.

Если нужно присвоить свойство объекта переменной с другим названием, например, свойство options.width присвоить переменной *w*, то можно использовать двоеточие:

let options = {

  title: "Menu",

  width: 100,

  height: 200

};

// { sourceProperty: targetVariable }

let {width: w, height: h, title} = options;

// width -> w, height -> h, title -> title

alert(title);  // Menu

alert(w);      // 100

alert(h);      // 200

Двоеточие показывает «что : куда идёт». В примере выше свойство width сохраняется в переменную *w*, свойство height сохраняется в *h*, а title присваивается одноимённой переменной.

Для потенциально отсутствующих свойств можно установить значения по умолчанию, используя "=":

let options = {

  title: "Menu"

};

let {width = 100, height = 200, title} = options;

alert(title);  // Menu

alert(width);  // 100

alert(height); // 200

Как и в случае с массивами, значениями по умолчанию могут быть любые выражения или даже функции. Они выполнятся, если значения отсутствуют.

[**Остаток объекта «…»**](https://learn.javascript.ru/destructuring-assignment#ostatok-obekta)

Если в объекте больше свойств, чем переменных, то можно использовать троеточие, так же как для массивов. В некоторых старых браузерах (IE) это не поддерживается, необходимо использовать полифилы. Например:

let options = {

  title: "Menu",

  height: 200,

  width: 100

};

let {title, ...rest} = options;

// title="Menu", rest={height: 200, width: 100}

alert(rest.height);  // 200

alert(rest.width);   // 100

В примерах выше переменные были объявлены в присваивании: let {…} = {…}. Если использовать существующие переменные и не указывать let, то это не будет работать. Проблема в том, что JavaScript обрабатывает {...} в основном потоке кода (не внутри другого выражения) как блок кода. Такие блоки кода могут быть использованы для группировки операторов, например:

{

  // блок кода

  let message = "Hello";

  // ...

  alert( message );

}

Так что здесь JavaScript считает, что видит блок кода, отсюда и ошибка. На самом-то деле здесь деструктуризация. Чтобы показать JavaScript, что это не блок кода, можно заключить выражение в скобки (...):

let title, width, height;

 ({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu

[**Вложенная деструктуризация**](https://learn.javascript.ru/destructuring-assignment#vlozhennaya-destrukturizatsiya)

Если объект или массив содержит другие вложенные объекты или массивы, то можно использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства. В приведённом ниже коде options хранит другой объект в свойстве size и массив в свойстве items. Шаблон в левой части присваивания имеет такую же структуру, чтобы извлечь данные из них:

let options = {

  size: {

    width: 100,

    height: 200

  },

  items: ["Cake", "Donut"],

  extra: true

};

let {

  size: {

    width,

    height

  },

  items: [item1, item2],

  title = "Menu"

} = options;

alert(title);  // Menu

alert(width);  // 100

alert(height); // 200

alert(item1);  // Cake

alert(item2);  // Donut

Весь объект options, кроме свойства extra, присваивается в соответствующие переменные. В итоге есть width, height, item1, item2 и title со значением по умолчанию. Заметьте, что переменные для size и items отсутствуют, так как сразу использовали их содержимое.

[**Умные параметры функций**](https://learn.javascript.ru/destructuring-assignment#umnye-parametry-funktsiy)

Есть ситуации, когда функция имеет много параметров, большинство из которых не обязательны. Это особенно верно для пользовательских интерфейсов. Представьте себе функцию, которая создаёт меню. Она может иметь ширину, высоту, заголовок, список элементов и так далее. Вот так – плохой способ писать подобные функции:

function showMenu(title = "Untitled", width = 200, height = 100, items = []) {

  // ...

}

В реальной жизни проблема заключается в том, как запомнить порядок всех аргументов. Обычно IDE помогают в этом, особенно если код хорошо документирован. Но есть и другая проблема, которая заключается в том, как вызвать функцию, когда большинство параметров передавать не надо, и значения по умолчанию вполне подходят. Разве что вот так:

// undefined там, где подходят значения по умолчанию

showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])

Это выглядит плохо. И становится нечитаемым, когда имеется большое количество параметров. Решить эту проблему помогает деструктуризация. Можно передать параметры как объект, и функция немедленно деструктурирует его в переменные:

let options = {

  title: "My menu",

  items: ["Item1", "Item2"]

};

function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {

  alert( `${title} ${width} ${height}` ); // My Menu 200 100

  alert( items ); // Item1, Item2

}

showMenu(options);

Также можно использовать более сложное деструктурирование с вложенными объектами и двоеточием:

let options = {

  title: "My menu",

  items: ["Item1", "Item2"]

};

function showMenu({

  title = "Untitled",

  width: w = 100,

  height: h = 200,

  items: [item1, item2]

}) {

  alert( `${title} ${w} ${h}` ); // My Menu 100 200

  alert( item1 ); // Item1

  alert( item2 ); // Item2

}

showMenu(options);

Полный синтаксис – такой же, как для деструктурирующего присваивания:

function({

  incomingProperty: varName = defaultValue

  ...

})

Тогда для объекта с параметрами, будет создана переменная varName для свойства с именем incomingProperty, по умолчанию равная defaultValue. Обратите внимание, что такое деструктурирование подразумевает, что в showMenu() будет обязательно передан аргумент. Если нужны все значения по умолчанию, то следует определить пустой объект:

showMenu({}); // все значения по умолчанию

showMenu(); // ошибка

Можно исправить это, сделав {} значением по умолчанию для всего объекта параметров:

function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {

  alert( `${title} ${width} ${height}` );

}

showMenu(); // Menu 100 200

В приведённом коде выше весь объект аргументов по умолчанию равен {}, поэтому всегда есть что-то, что можно деструктурировать.

1. **Остаточные параметры и оператор расширения.**

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов. Например: Math.max(arg1, arg2, ..., argN) – вычисляет максимальное число из переданных; Object.assign(dest, src1, ..., srcN) – копирует свойства из исходных объектов src1..N в целевой объект dest и др.

[**Остаточные параметры (...)**](https://learn.javascript.ru/rest-parameters-spread-operator#ostatochnye-parametry)

Вызывать функцию можно с любым количеством аргументов, независимо от того, как она была определена. Например:

function sum(a, b) {

  return a + b;

}

alert( sum(1, 2, 3, 4, 5) );

Лишние аргументы не вызовут ошибку, но приняты будут только первые два.

Остаточные параметры могут быть обозначены через три точки «...». Суть его в том, что оставшиеся параметры помещаются в массив. Например, соберём все аргументы в массив args:

function sumAll(...args) {

  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

Можно положить первые несколько параметров в переменные и собрать в массив остальные. В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив titles:

function showName(firstName, lastName, ...titles) {

  alert( firstName + ' ' + lastName ); // Юлий Цезарь

  // titles = ["Консул", "Император"]

  alert( titles[0] ); // Консул

  alert( titles[1] ); // Император

  alert( titles.length ); // 2

}

showName("Юлий", "Цезарь", "Консул", "Император");

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них. Это вызовет ошибку:

function f(arg1, ...rest, arg2) { // Ошибка

}

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами. Например:

function showName() {

  alert( arguments.length );

  alert( arguments[0] );

  alert( arguments[1] );

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined

showName("Илья");

Раньше в языке не было остаточных параметров, и получить все аргументы функции можно было только с помощью arguments. Этот способ всё ещё работает, его можно найти в старом коде. Но у него есть один недостаток. Хотя arguments похож на массив, и он тоже перебираемый, это всё же не массив. Он не поддерживает методы массивов, поэтому нельзя, например, вызвать arguments.map(...). К тому же, arguments всегда содержит все аргументы функции – нельзя получить их часть. А остаточные параметры позволяют это сделать.

Соответственно, для более удобной работы с аргументами лучше использовать остаточные параметры.

У стрелочных функций нет своего объекта arguments. Если обратиться к arguments из стрелочной функции, то получим аргументы внешней обычной функции. Пример:

function f() {

  let showArg = () => alert(arguments[0]);

  showArg(2);

}

f(1); // 1

[**Оператор расширения**](https://learn.javascript.ru/rest-parameters-spread-operator#spread-operator)

Иногда нужно массив преобразовать в список параметров. Например, есть встроенная функция [Math.max](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/max). Она возвращает наибольшее число из списка:

alert( Math.max(3, 5, 1) ); // 5

Если вызвать эту функцию для массива чисел [3, 5, 1], то она его не обработает, так как ожидает список параметров:

let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN

Чтобы преобразовать массив в список необходимо использовать оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов. Для Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5

Этим же способом можно передать несколько итерируемых объектов и комбинировать оператор расширения с обычными значениями:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

Оператор расширения можно использовать и для слияния массивов:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15

Оператора расширения работает с любым перебираемым объектом. Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

Оператор расширения использует итераторы, чтобы собирать элементы. Так же, как это делает for..of. Цикл for..of перебирает строку как последовательность символов, поэтому из ...str получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: [...str].

Для этой задачи можно использовать и Array.from. Он тоже преобразует перебираемый объект (такой как строка) в массив:

let str = "Привет";

alert( Array.from(str) ); // П,р,и,в,е,т

Результат аналогичен [...str]. Но между Array.from(obj) и [...obj] есть разница: Array.from работает как с псевдомассивами, так и с итерируемыми объектами; оператор расширения работает только с итерируемыми объектами. Таким образом, если нужно сделать из чего угодно массив, Array.from – более универсальный метод.