Деструктурирующее присваивание

В JavaScript есть две чаще всего используемые структуры данных – это `Object` и `Array`.

Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» 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

Пример деструктуризации массива:

```
// у нас есть массив с именем и фамилией
let arr = ["Ilya", "Kantor"]

// деструктурирующее присваивание
// записывает firstName=arr[0], surname=arr[1]
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname);  // Kantor
```

Теперь мы можем использовать переменные вместо элементов массива.

Отлично смотрится в сочетании со `split` или другими методами, возвращающими массив:

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

«Деструктуризация» не означает «разрушение».

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

Это просто короткий вариант записи:

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

Пропускайте элементы, используя запятые

Ненужные элементы массива также могут быть отброшены через запятую:

```
// второй элемент не нужен
let [firstName, , title] = ["Julius", "Caesar", "C
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).

Работает с любым перебираемым объектом с правой стороны

...На самом деле мы можем использовать любой перебираемый объект, не только массивы:

```
let [a, b, c] = "abc";
let [one, two, three] = new Set([1, 2, 3]);
```

Присваивайте чему угодно с левой стороны

Мы можем использовать что угодно «присваивающее» с левой стороны.

Например, можно присвоить свойству объекта:

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split('
alert(user.name); // Ilya
```

Цикл с .entries()

В предыдущей главе мы видели метод Object.entries(obj).

Мы можем использовать его с деструктуризацией для цикличного перебора ключей и значений объекта:

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

// цикл по ключам и значениям
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:John, затем ag
}
```

...то же самое для map:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

for (let [key, value] of user) {
  alert(`${key}:${value}`); // name:John, затем ag
}
```

Остаточные параметры «...»

Если мы хотим не просто получить первые значения, но и собрать все остальные, то мы можем добавить ещё один параметр, который получает остальные значения, используя оператор «остаточные параметры» – троеточие (`"..."`):

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Con

alert(name1); // Julius
alert(name2); // Caesar

// Обратите внимание, что `rest` является массивом.
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, просто убедитесь, что перед переменной есть три точки и она стоит на последнем месте в деструктурирующем присваивании.

Значения по умолчанию

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 нам необходимо указать значения по умолчанию, то мы можем использовать `=` :

```
// значения по умолчанию
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, здесь мы используем функцию `prompt` для указания двух значений по умолчанию. Но она будет запущена только для отсутствующего значения:

```
// prompt запустится только для surname
let [name = prompt('name?'), surname = prompt('surname?

alert(name);    // Julius (из массива)
alert(surname); // результат prompt
```

Деструктуризация объекта

Деструктурирующее присваивание также работает с объектами.

Синтаксис:

```
let {var1, var2} = {var1:…, var2:…}
```

У нас есть существующий объект с правой стороны, который мы хотим 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: 2
```

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

Если мы хотим присвоить свойство объекта переменной с другим названием, например, свойство `options.width` присвоить переменной `w`, то мы можем использовать двоеточие:

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

```
6
7  // { sourceProperty: targetVariable }
8  let {width: w, height: h, title} = options;
9
10 // width -> w
11 // height -> h
12 // title -> title
13
14 alert(title);  // Menu
15 alert(w);      // 100
16 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 мы можем установить значения по умолчанию, используя `"="`, как здесь:

```
1  let options = {
2    title: "Menu"
3  };
4
5  let {width = 100, height = 200, title} = options;
6
7  alert(title);  // Menu
8  alert(width);  // 100
9  alert(height); // 200
```

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

В коде ниже `prompt` запросит `width`, но не `title`:

```
1  let options = {
2    title: "Menu"
3  };
4
5  let {width = prompt("width?"), title = prompt("title?")
6
7  alert(title);  // Menu
8  alert(width);  // (результат prompt)
```

Мы также можем совмещать `:` и `=`:

```
1  let options = {
2    title: "Menu"
3  };
4
5  let {width: w = 100, height: h = 200, title} = options;
6
7  alert(title);  // Menu
8  alert(w);      // 100
9  alert(h);      // 200
```

Если у нас есть большой объект с множеством свойств, можно взять только то, что нужно:

```
1  let options = {
2    title: "Menu",
3    width: 100,
4    height: 200
5  };
6
7  // взять только title, игнорировать остальное
8  let { title } = options;
```

```
9
10 alert(title); // Menu
```


Остаток объекта «...»

Что если в объекте больше свойств, чем у нас переменных? Можем ли мы взять необходимые нам, а остальные присвоить куда-нибудь?

Можно использовать троеточие, как и для массивов. В некоторых старых браузерах (IE) это не поддерживается, используйте Babel для полифила.

Выглядит так:

```
1  let options = {
2    title: "Menu",
3    height: 200,
4    width: 100
5  };
6
7  // title = свойство с именем title
8  // rest = объект с остальными свойствами
9  let {title, ...rest} = options;
10
11 // сейчас title="Menu", rest={height: 200, width: 100}
12 alert(rest.height);  // 200
13 alert(rest.width);   // 100
```

Обратите внимание на `let`

В примерах выше переменные были объявлены в присваивании: `let {…} = {…}`. Конечно, мы могли бы использовать существующие переменные и не указывать `let`, но тут есть подвох.

Вот так не будет работать:

```
let title, width, height;

// ошибка будет в этой строке
{title, width, height} = {title: "Menu", width: 20
```

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 у нас деструктуризация.

Чтобы показать JavaScript, что это не блок кода, мы можем заключить выражение в скобки `(...)`:

```
let title, width, height;

// сейчас всё работает
({title, width, height} = {title: "Menu", width: 2

alert( title ); // Menu
```

Вложенная деструктуризация

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем 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:

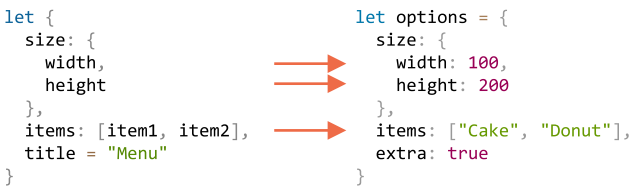
```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// деструктуризация разбита на несколько строк для ясно
let {
  size: { // положим size сюда
    width,
    height
  },
  items: [item1, item2], // добавим элементы к items
  title = "Menu" // отсутствует в объекте (используется
```

```
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

Весь объект `options`, кроме свойства `extra`, которое в левой части отсутствует, присваивается в соответствующие переменные:



В итоге у нас есть `width`, `height`, `item1`, `item2` и `title` со значением по умолчанию.

Заметим, что переменные для `size` и `items` отсутствуют, так как мы взяли сразу их содержимое.

Умные параметры функций

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, heig
  // ...
}
```

В реальной жизни проблема заключается в том, как запомнить порядок всех аргументов. Обычно IDE пытаются помочь нам, особенно если код хорошо документирован, но всё же... Другая проблема 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", "It
```

Это выглядит ужасно. И становится нечитаемым, когда мы имеем дело с большим количеством параметров.

На помощь приходит деструктуризация!

Мы можем передать параметры как объект, и функция немедленно деструктурирует его в переменные:

```
// мы передаём объект в функцию
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// ...и она немедленно извлекает свойства в переменные
function showMenu({title = "Untitled", width = 200, hei
  // title, items – взято из options,
  // width, height – используются значения по умолчанию
  alert( `${title} ${width} ${height}` ); // My Menu 20
  alert( items ); // Item1, Item2
```

```
}

showMenu(options);
```

Мы также можем использовать более сложное деструктурирование с вложенными объектами и двоеточием:

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({
  title = "Untitled",
  width: w = 100,  // width присваиваем в w
  height: h = 200, // height присваиваем в h
  items: [item1, item2] // первый элемент items присваи
}) {
  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`.

Пожалуйста, обратите внимание, что такое деструктурирование подразумевает, что в `showMenu()` будет обязательно передан аргумент. Если нам нужны все значения по умолчанию, то нам следует передать пустой объект:

```
showMenu({}); // ок, все значения - по умолчанию

showMenu(); // так была бы ошибка
```

Мы можем исправить это, сделав `{}` значением по умолчанию для всего объекта параметров:

```
function showMenu({ title = "Menu", width = 100, height
  alert( `${title} ${width} ${height}` );
}

showMenu(); // Menu 100 200
```

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

Итого

- Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.
- Полный синтаксис для объекта:

```
let {prop : varName = default, ...rest} = object
```

Свойства, которые не были упомянуты, копируются в объект `rest`.

- Полный синтаксис для массива:

```
let [item1 = default, item2, ...rest] = array
```

Первый элемент отправляется в `item1`; второй отправляется в `item2`, все остальные элементы попадают в массив `rest`.

- Можно извлекать данные из вложенных объектов и массивов, для этого левая сторона должна иметь ту же структуру, что и правая.

Задачи

Деструктурирующее присваивание

важность: 5

У нас есть объект:

```
let user = {
  name: "John",
  years: 30
};
```

Напишите деструктурирующее присваивание, которое:

- свойство `name` присвоит в переменную `name`.
- свойство `years` присвоит в переменную `age`.
- свойство `isAdmin` присвоит в переменную `isAdmin` (false, если нет такого свойства)

Пример переменных после вашего присваивания:

```
let user = { name: "John", years: 30 };

// ваш код должен быть с левой стороны:
// ... = user

alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

решение

Максимальная зарплата

важность: 5

У нас есть объект `salaries` с зарплатами:

```
let salaries = {
  "John": 100,
  "Pete": 300,
  "Mary": 250
};
```

Создайте функцию `topSalary(salaries)`, которая возвращает имя самого высокооплачиваемого сотрудника.

- Если объект `salaries` пустой, то нужно вернуть `null`.
- Если несколько высокооплачиваемых сотрудников, можно вернуть любого из них.

P.S. Используйте `Object.entries` и деструктурирование, чтобы перебрать пары ключ/значение.

Открыть песочницу с тестами для задачи.

решение

Комментарии

перед тем как писать...