Оглавление

В типичных программах JS примитивы `Symbol` используются не так часто. В основном они встречаются в низкоуровневом коде: библиотеках, фреймворках и т. д.

Массивы и объекты

Кроме примитивов в JS также используются объектные значения.

Как упоминалось ранее, массивы представляют собой особую разновидность объекта — упорядоченный список данных с числовыми индексами:

```
var names = [ "Frank", "Kyle", "Peter", "Susan" ];

names.length;
// 4

names[0];
// Frank

names[1];
// Kyle
```

Массивы JS позволяют хранить значения любых типов — как примитивы, так и объекты (включая другие массивы). Как будет показано в конце главы 3, даже функции являются значениями, которые могут храниться в массивах или объектах.

Функции, как и массивы, являются особой разновидностью (подтипом) объектов. Вскоре функции будут рассмотрены более подробно.

Объекты имеют более общую природу: они являются неупорядоченными наборами произвольных значений с доступом по ключу. Иначе говоря, вы обращаетесь к элементам по строковому имени (ключу или свойству) вместо числовой позиции (как в случае с массивами). Пример:

```
var me = {
    first: "Kyle",
    last: "Simpson",
    age: 39,
    specialties: [ "JS", "Table Tennis" ]
};

console.log(`My name is ${ me.first }.`);
```

Здесь `me` представляет объект, а `first` представляет имя, определяющее местонахождение информации в объекте (коллекции значений). Также для обращения информации в объекте может использоваться свойство/ключ в квадратных скобках:

```
me["first"]
```

Определение типа значения

Чтобы вы могли различать значения, оператор `typeof` возвращает встроенный тип значения для примитивов или `"object"` в противном случае:

```
typeof 42;                  // "number"
typeof "abc";               // "string"
typeof true;                // "boolean"
typeof undefined;           // "undefined"
typeof null;                // "object" -- ошибка!
typeof { "a": 1 };          // "object"
typeof [1,2,3];             // "object"
typeof function hello(){}; // "function"
```

К сожалению, `typeof null` возвращает `"object"` вместо ожидаемого `"null"`. Также `typeof` возвращает `"function"` для функций, но не возвращает `"array"` для массивов, как можно было бы предположить.

Преобразование между типами будет более подробно рассмотрено в этой главе.

Примитивные и объектные значения по-разному ведут себя при присваивании или передаче. Все эти нюансы рассматриваются в приложении А, раздел «Значения и ссылки».

Объявление и использование переменных

Пожалуй, стоит сказать то, что могло остаться неочевидным из предыдущего раздела: в программах JS значения либо присутствуют в виде литералов (как во многих предыдущих примерах), либо содержатся в переменных; переменные можно рассматривать как контейнеры для значений.

Чтобы переменная могла использоваться в программе, ее необходимо объявить (создать). Существуют различные синтаксические формы объявления переменных (идентификаторов), и каждая форма подразумевает свое поведение.

Для примера возьмем команду `var`:

```
var myName = "Kyle";
var age;
```

Ключевое слово `var` объявляет переменную для использования в этой части программы и допускает необязательное присваивание исходного значения.

Также существует похожее ключевое слово `let`:

```
let myName = "Kyle";
let age;
```

Ключевое слово `let` отличается от `var`. Самое очевидное различие заключается в том, что оно открывает более ограниченный доступ к переменной по сравнению с `var`. Это называется блоковой видимостью, в отличие от обычной (функциональной) видимости.

Пример:

```
var adult = true;

if (adult) {
    var myName = "Kyle";
    let age = 39;
    console.log("Shhh, this is a secret!");
}

console.log(myName);
// Kyle

console.log(age);
// Ошибка!
```

Попытка обратиться к `age` вне команды `if` приводит к ошибке, потому что переменная `age` имеет блоковую область видимости, которая ограничивается командой `if`, тогда как с переменной `myName` дело обстоит иначе.

Блоковая область видимости чрезвычайно удобна для ограничения распространения объявлений переменных в программах, так как она способствует предотвращению случайного перекрытия имен.

Однако ключевое слово `var` тоже полезно: оно означает «эта переменная должна быть видимой в более

широкой области видимости (всей функции)». Обе формы объявления могут оказаться уместными в той или иной части программы в зависимости от обстоятельств.

Очень часто приходится слышать, что от `var` следует полностью отказаться в пользу `let` (или `const`) — обычно из-за воображаемой путаницы с тем, как изменялось поведение `var` в отношении областей видимости с первых дней существования JS. Я считаю, что такой подход создает слишком серьезные ограничения и в конечном итоге бесполезен. Он предполагает, что вы не способны изучить и правильно использовать некоторую возможность языка в сочетании с другими возможностями. На мой взгляд, вы *можете* и *должны* изучать доступные возможности и использовать их там, где они уместны!

Третья форма объявления — `const`. Она похожа на `let`, но с дополнительным ограничением: ее значение должно быть задано в момент объявления и ей не может быть присвоено другое значение позднее.

Пример:

```
const myBirthday = true;
let age = 39;

if (myBirthday) {
    age = age + 1;       // OK!
    myBirthday = false; // Ошибка!
}
```

Изменение константы `myBirthday` и присваивание ей другого значения невозможно.

Переменные, объявленные ключевым словом `const`, не являются немодифицируемыми — просто им невозможно присвоить новое значение. Не рекомендуется использовать `const` с объектными значениями, потому что эти значения все равно могут изменяться, даже без повторного присваивания. В конечном итоге это приводит к потенциальной путанице, так что я считаю, что подобных ситуаций следует избегать:

```
const actors = [
"Morgan Freeman", "Jennifer Aniston"
];

actors[2] = "Tom Cruise"; // OK :(
actors = [];              // Ошибка!
```

Семантика использования `const` лучше всего подходит для ситуации, в которой у вас имеется одно примитивное значение, которому вы хотите присвоить полезное имя. Например, чтобы использовать `myBirthday` вместо `true`. Это упрощает чтение программ.

Если вы будете использовать `const` только с примитивными типами, вы сможете избежать любой путаницы с повторным присваиванием (запрещенным) и модификацией (разрешенной). Это самый надежный и лучший способ использования `const`.

Кроме var/let/const существуют и другие синтаксические формы, объявляющие идентификаторы (переменные) в различных областях видимости. Пример:

```
function hello(myName) {
    console.log(`Hello, ${ myName }.`);
}

hello("Kyle");
// Hello, Kyle.
```

Идентификатор hello создается во внешней области видимости и при этом автоматически связывается со ссылкой на функцию. Но именованный параметр myName создается только внутри функции, и поэтому доступ к нему возможен только в области видимости функции. В целом hello и myName обычно ведут себя как объявленные ключевым словом var.

Также для объявления переменной может использоваться конструкция catch:

```
try {
    someError();
}
catch (err) {
    console.log(err);
}
```

Переменная err имеет блоковую область видимости, которая существует только внутри конструкции catch, как если бы она была объявлена ключевым словом let.

Функции

Слово «функция» имеет много смыслов в области программирования. Например, в мире функционального программирования термин «функция» имеет точное математическое определение и подразумевает жесткий набор правил, которые должны соблюдаться.

В JS смысл функции расширяется до другого взаимосвязанного термина: «процедура». Процедура представляет собой набор команд, который может вызываться один или несколько раз, может получать входные данные и может возвращать одно или несколько значений.

С первых дней существования JS определение функции выглядело так:

```
function awesomeFunction(coolThings) {
    // ..
    return amazingStuff;
}
```

Эта конструкция называется объявлением функции, потому что выглядит как самостоятельная команда, а не как выражение в другой команде.

Связь между идентификатором `awesomeFunction` и значением-функцией устанавливается в фазе компиляции кода, до его выполнения.

В отличие от команды объявления функции, определение и присваивание *функциональных выражений* могут выглядеть так:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

Эта функция является выражением, которое присваивается переменной `awesomeFunction`. В отличие от формы с объявлением функции, функциональное выражение не связывается с идентификатором до момента выполнения команды на стадии выполнения.

Очень важно понимать, что в JS функции являются значениями, которые могут присваиваться (как в этом фрагменте) и передаваться при вызове. Собственно функции JS составляют особую разновидность типов объектных значений. Не во всех языках функции рассматриваются как значения, но для языка очень важно поддерживать этот паттерн функционального программирования, как это делается в JS.

Функции JS могут получать входные данные в параметрах:

```
function greeting(myName) {
    console.log(`Hello, ${ myName }!`);
}

greeting("Kyle"); // Hello, Kyle!
```

В этом фрагменте идентификатор `myName` называется параметром; он действует как локальная переменная внутри функции. Функции могут определяться так, чтобы они получали любое количество параметров, от 0 и далее по вашему усмотрению. Каждому параметру присваивается значение-аргумент, которое передается в соответствующей позиции при вызове (`"Kyle"` в данном случае).

Функции также могут возвращать значения при помощи ключевого слова `return`:

```
function greeting(myName) {
    return `Hello, ${ myName }!`;
}

var msg = greeting("Kyle");

console.log(msg); // Hello, Kyle!
```

Вернуть можно только одно значение, но если вам потребовалось вернуть несколько значений, их можно упаковать в объект/массив. Так как функции являются значениями, их можно присваивать как свойства объекта:

```
var whatToSay = {
    greeting() {
        console.log("Hello!");
    },
    question() {
        console.log("What's your name?");
    },
```

```
    answer() {
        console.log("My name is Kyle.");
    }
};

whatToSay.greeting();
// Hello!
```

В этом фрагменте ссылки на три функции (`greeting()`, `question()` и `answer()`) включаются в объект, хранящийся под именем `whatToSay`. Каждую функцию можно вызвать, обратившись к свойству для получения значения — ссылки на функцию. Сравните этот прямолинейный стиль определения функций в объектах с более сложным синтаксисом классов, который будет рассматриваться позже в этой главе.

Функции принимают в JS много разных форм. Все эти вариации будут рассматриваться в приложении А, раздел «Многоликие функции».

Сравнения

Чтобы принимать решения в программах, необходимо сравнивать значения друг с другом, чтобы определить их отличительные признаки и отношения друг с другом. В JS предусмотрено несколько механизмов сравнения значений.