Entwicklung Web-basierter Anwendungen

Prof. Dr. Stefan Linus Zander

Einführung in JavaScript

Outline

- Foundational Concepts
- Overview of Datatypes
- Primitive Types
- Reference Types and Reference Values
- Arrays
- Constructors
- Properties

Foundational Concepts

Object-based ≠ Object-oriented

- OO developers often get disoriented when working with JavaScript since
  - JS has no formal OO concept of classes (at least until ECMAScript 6)
  - JS classes serve as syntactic sugar for object types
  - The **biggest mistake** in learning JS is trying to transfer your OO-knowledge to JS

JavaScript ↔ Flexibility

- JavaScript is an incredibly **flexible** and **powerfull language**
  - You can just start coding; no formal class or package structure
  - Code along the pay-as-you-go principle
  - With great flexibility comes great responsibility

→ Objects ←

- JavaScript makes **Objects** the central part of the language
  - All data in JS are either objects or can be accessed through objects
  - Even functions are objects(!!!)
  - Objects are extremely flexible and can be modified at any time

So, working with and understanding objects is key to understanding JavaScript as a language.

JavaScript employs two Types of Datatypes

Primitive Types

- Primitive types are stored as **simple data types**
  - ie., the variable object holds the actual value
- JS has 5 primitive types
  - **Boolean** – true or false
  - **Number** – integer or floating point number
  - **String** – character or sequence of characters
  - **Null** – a primitive type with only one value `null`
  - **Undefined** – value assigned to unitialized variables
- Primitive types `Boolean`, `Number`, and `String` are treated as reference types to make the language more consistent
- All primitive types have literal value representations

Reference Types

- Reference types are stored as **objects**
  - i.e., the variable object holds a reference to the memory location where the actual object data are stored
- Objects are the main building blocks of JavaScript
- Objects are extremely **flexible** and **powerful**
  - Since JavaScript has no formal concept of classes, objects resemble the role of both **types** and **instances**
- Reference types serve as **blueprints** for reference values
- Reference values are **instances** of reference types
- JavaScript contains a set of built-in reference types such as `Array`, `Date`, `Error`, `Function`, `RegExp`

Each variable in JavaScript is associated with a specific primitive or reference type

Working with Primitive Types

A) Defining Primitive Types

- A variable holding a primitive directly contains the primitive value
- Values are represented as literals and stored directly in the variable object

```
let color1 = "red";
let color2 = color1;

console.log(color1); // red
console.log(color2); // red

color1 = "blue";

console.log(color1); // blue
console.log(color2); // red
```

B) Identifying Primitive Types

- The `typeof` operator identifies a primitive type
- It returns the type in form of a string

```
console.log(typeof "Nicholas"); // "string"
console.log(typeof 10);         // "number"
console.log(typeof 5.1);        // "number"
console.log(typeof true);       // "boolean"
console.log(typeof undefined);  // "undefined"
```

Primitive Wrapper Types

The primitive types

- `String`
- `Number`
- `Boolean`

have **primitive wrapper types**[1] that allow them to be used like objects.

Primitive wrapper types are special reference types that are automatically created whenever one of the tree types are read.

This proces is called autoboxing.

Examples

```
let name = "Nicholas";
let lowercaseName = name.toLowerCase();   // convert to lowercase
let firstLetter = name.charAt(0);         // get first character
let middleOfName = name.substring(2, 5);  // get characters 2-4

let count = 10;
let fixedCount = count.toFixed(2);        // convert to "10.00"
let hexCount = count.toString(16);        // convert to "a"

let flag = true;
let stringFlag = flag.toString();         // convert to "true"

name.last = "zakas";
console.log(name.last);                   // undefined
```

[1] Primitive wrapper types are **special reference types** for String, Number, and Boolean primitive types that allow to use them as if they were reference values without introducing a new syntax.

Global Scope and Local Scope

Global Scope

- A variable declared at the top of a program or outside of a function is considered a global scope variable.

```
let myName = "Thomas";

function sayHello() {
  console.log("Hello, my name is " + myName);
}

sayHello()    // Hello, my name is Thomas
```

Local Scope (aka Function Scope)

- A variable declared in a function body is only visible within the function body and its inner-functions.
- It will be deleted when the function is removed from the call stack.

```
function sayHello() {
  let myName = "Thomas";
  console.log("Hello, my name is " + myName);
}

function sayGoodbye() {
  console.log("Have a nice day " + myName);
}

sayHello()    // Hello, my name is Thomas
sayGoodbye()  // Error
```

```
let myName = "Thomas";

function sayHello() {
  let myName = "Georg";
  console.log("Hello, my name is " + myName);
}

function sayGoodbye() {
  console.log("Have a nice day " + myName);
}

sayHello()    // Hello, my name is Georg
sayGoodbye()  // Have a nice day Thomas
```

Declaring Variables using `let`, `var`, and `const`

Var

- `var` declares a variable in a global scope
- The variable is always hoisted to the top of the scope (e.g. the function in which `var` is used.)
- Should not be used

```
function getValue(condition) {
    if (condition) {
        value  = "green";
        // value is hoisted to the top of the function
        var value
    }
    // value = "green"; // value should not exist here
    console.log(value); // displays 'green'
}
```

Let

- Introduced with ES2015 (ES6) and preferred notion
- `let` declares a variable in the current scope (function of block – indicated by `{` and `}`)
- `let` declarations are NOT hoisted, so declare them at the beginning of a block

```
function getValue(condition) {
    if (condition) {
        value  = "green";
        // value is hoisted to the top of the function
        let value
    }
    console.log(value); // Reference Error
}
```

Const

- `const` is used to define constants, the value of which can not be changed once declared
- Constants are only valid in their block

```
if (condition) {
    const maxItems = 5;
}
// maxItems isn't accessible here
```

Comparisons

- `===` Comparison without coercing the variables to another type
- `==` Comparison with type coercion

```
console.log("5" == 5);              // true
console.log("5" === 5);             // false

console.log(undefined == null);     // true
console.log(undefined === null);    // false
```

Equality of Primitives and Objects

- **Primitives**: comparing the actual value
- **Objects**: comparing the memory location

Reference Types and Reference Values

What are Reference Types ?

- Objects in JavaScript exist as both **reference types** and **reference values**
- The **role** objects play depends on their usage
  - an object in literal form serves as instance
  - an object used as constructor serves as reference type and resembles the class-concept of OO languages
- Reference types can be specified in different notations
  - **Literal Form**: used to define a reference value, ie., instance of a reference type
  - **Constructor Function**: the reference type serves as blueprint for other reference values of the same type
- Objects in JavaScript consists of an **unordered list of properties**
- **Properties** are key-value pairs where
  - the **key** is always a string and serves as name for the value
  - the **value** can hold any kind of primitive or reference value

Naming Convention
Although reference types are objects, it is useful to distinguish them from reference values. Hence, when we talk about objects, we refer to instances of reference types, ie., reference values. When we talk about reference types, we use this denominator.

Instantiating Objects

A) Usign the `new` Operator with a Constructor

- A constructor is a function that uses `new` to create an object
- Any function can be a constructor[1]
- Such functions are called **Constructor functions**

```
function Book(name, year) {
  this.name = name;
  this.year = year;
}
let b = new Book("ECMAScript 6", 2015);
```

B) Using the `Object()` Function as Constructor

```
  let book = new Object();
  book.name = "The Principles of Object-Oriented JavaScript";
  book.year = 2014;
```

C) Using the Literal Notation

- Objects (=reference values) can be directly created using the literal notation syntax without `new` operator and constructor.

```
let book = {
  name: "The Principles of Object-Oriented JavaScript",
  year: 2014
};
```

- Property names can also be represented as string literals

```
let book = {
  "name": "The Principles of Object-Oriented JavaScript",
  "year": 2014
};
```

Please note the different syntax of the literal notation!

[1] Constructors start with an upper case (capital letter) by convention to distinguish them from non-constructor functions.

Constructors

- Constructors are **special functions** that serve as blueprints for objects
- A **constructor** allows to create multiple instances of a reference type
  - those instances contain the same properties and methods as defined in the reference type
- A constructor does not return anything unless otherwise specified
  - The `new` operator creates a new object from a constructor and returns it
  - When a constructor returns an object, that object will be used instead of the newly created instance
- Every object is created with a special **constructor property** that points to its constructor
  - This is the reason why `instanceof` can deduce the type of an object
  - For objects created through the literal notation, the constructor property points to `Object`
- Constructors don't eleminate **code redundancy**
  - e.g. each created object contains a copy of the same method property although the behaviour does not change among instance

Built-in Reference Types

- The `Object`-type is one generic **built-in reference type**
- The other built-in reference types are more specialized in their intended usage
  - `Array` – ordered list of numerically indexed values
  - `Date` – for date and time data
  - `Error` – A runtime error
  - `Function` – a function
  - `RegExp` – a regular expression

```
const items = new Array();
let now = new Date();
let error = new Error("Something bad happened.");
let func = new Function("console.log('Hi');");
let object = new Object();
let re = new RegExp("\\d+");
```

Arrays

```
// Array creation
const colors = ["red", "blue", "green" ];
const points = new Array(40, 100, 1, 5, 25);
// Create an array with one element
const points = [40];
// Create an array with 40 undefined elements
const points = new Array(40);
// reads the 3rd element
let one = points[2];
// overwrites first element
points[0] = 50;
// 40 * 100 * 1 * 5 * 25
let p_str = points.join(" * ");
// removes "25" from the array
let p = points.pop();
// adds "10" to the array
points.push(10);
// stores "40" in p variable
let p = points.shift();
// returns 5
let l = points.length
// deletes "100"; points[1] is now undefined
delete points[1];
// merges two arrays into a new one
```

- Unlike many languages, JavaScript **does not** provide associative arrays
- In JavaScript, arrays always use **numbered indexes**
- Objects, in contrast, use **named indexes**
- It is common practice to declare arrays with the `const` keyword.
- `pop()` removes last element from an array
- `push()` adds an element to the end of an array
- `shift()` removes the first element and moves all other elements to a lower index
- The `length` property returns the number of array items
- `concat()` merges two or more arrays
- `join()` joins all array items into a string with a separator
- `toString()` displays all items as a string
- `delete` removes elements from an array but leaves a **hole** at the index
- Array elements can be **objects**, and hence also functions

Identification

A) Identifying Reference Types

- To identify a specific type, use the `instanceof` operator

```
var items = [];
var object = {};

function reflect(value) {
    return value;
}

console.log(items instanceof Array);          // true
console.log(items instanceof Object);         // true
console.log(object instanceof Object);        // true
console.log(object instanceof Array);         // false
console.log(reflect instanceof Function);     // true
console.log(reflect instanceof Object);       // true

console.log(Array.isArray(items));            // true
console.log(typeof reflect);                  // "function"
```

- Functions can be identified using `typeof` operator
- Arrays can be identified using the `Array.isArray()` function

B) Identifying Reference Values

- The `instanceof` operator also helps in identifying the type of a reference value

```
function Book(name, year) {
  this.name = name;
  this.year = year;
}

let b = new Book("ECMAScript 6", 2015);

console.log(b instanceof Object);     // true
console.log(b instanceof Book);       // true
console.log(b instanceof Person);     // false
```

Example #2

```
console.log(typeof Book);
console.log(typeof b);
```

Think about, what does the `typeof` operator return for each operation?

Properties

Properties

- Properties are the main building blocks of objects
- Objects can thus be perceived as **hash tables**
  - The values of their properties can be accessed in an associative form through their keys.
- Property **keys** are represented as string literals
- Proptery **values** can be both primitive as well as reference values
- When the value of a property is a function, it is called a **method**
- **Methods** are reference values that can be executed
  - They contain a special internal property called `[[call]]` that signals that the value needs to be exectuted

Property Types

A) "Own" Properties

- Own properties are properties that 'belong' exlusively to the object for which they are defined
  - The property is stored directly on the object
  - All operations on the property must be performed on that object

```
var person1 = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name); }
};

console.log("name" in person1);                  // true
console.log(person1.hasOwnProperty("name"));     // true

console.log("toString" in person1);              // true
console.log(person1.hasOwnProperty("toString")); // false
```

B) Prototype Properties

- Prototypes allow objects to **share** common properties
- A prototype is a **blueprint** for objects of the same type
- A prototype is shared among all instances

```
let book = {
  title: "The Principles of Object-Oriented JavaScript"
};

console.log("title" in book);                          // true
console.log(book.hasOwnProperty("title"));             // true
console.log("hasOwnProperty" in book);                 // true
console.log(book.hasOwnProperty("hasOwnProperty"));    // false
console.log(
  Object.prototype.hasOwnProperty("hasOwnProperty")); // true
```

Working with Properties

A) Adding Properties

```
let person1 = new Object();

// Variant #1
person1.surname = "Hans";
// Variant #2
person1["lastname"] = "Haas";

console.log(person1.surname);
console.log(person1["lastname"]);

// variable values can also be used
// to specify property keys
let key = "age";
person1[key] = "82";

console.log(person1[key]);
```

- Properties can be added at any time regardless of the creation method

B) Deleting Properties

```
let person1 = {
  name: "Nicholas"
};

console.log("name" in person1); // true

delete person1.name;

console.log("name" in person1); // false
console.log(person1.name);  // undefined
```

- A successful `delete` operation returns `true`
- Some properties can not be removed

C) Enumerating Properties

```
// Only iterates over OWN properties

let property;

for (property in Person) {
  console.log("Name: " + property);
  console.log("Value: " + Person[property]);
}
```

```
// Includes PROTOTYPE properties

let properties = Object.keys(object);

// if you want to mimic for-in behavior
let i,
    len = properties.length;

for (i=0; i < len; i++) {
  console.log("Name: " + properties[i]);
  console.log("Value: " +
    object[properties[i]]);
}
```

Working with Properties

D) Identifying Properties

- The `in` operator looks for the specified property with the given name and returns `true` in case of its existence
- The `in` operator works on both reference types and values

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.sayName = function() {
    console.log(this.name);
  }
}

console.log("name" in Person);    // true
console.log("age" in Person);     // true
console.log("sayName" in Person); // true
console.log("title" in Person);   // false
```

E) Identifying Own Properties only

- The `in` operator checks for both own properties and prototype properties
- If only own properties should be detected, a combination of `in` with the `hasOwnProperty` method[1] is needed

```
var person1 = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name); }
};

console.log("name" in person1);                  // true
console.log(person1.hasOwnProperty("name"));     // true

console.log("toString" in person1);              // true
console.log(person1.hasOwnProperty("toString")); // false
```

[1] The `hasOwnProperty` method is present on all objects in JavaScript

Working with Properties

F) Identifying Prototype Properties

- Determining whether a property is on the prototype requires a self-written function

```
function hasPrototypeProperty(object, name) {
  return name in object && !object.hasOwnProperty(name);
}

console.log(
  hasPrototypeProperty(book, "title"));          // false
console.log(
  hasPrototypeProperty(book, "hasOwnProperty")); // true
```

- If the property is `in` an object but `hasOwnProperty()` returns `false`, then the property is on the prototype

G) Accessing the `[[Prototype]]` Property

- Every object has an internal `[[Prototype]]` property that points to its prototype

```
let object = {};
let prototype = Object.getPrototypeOf(object);

console.log(prototype === Object.prototype);         // true
console.log(Object.prototype.isPrototypeOf(object)); // true
```

- `Object.getPrototypeOf()` returns the value of the `[[Prototype]]` property
- `isPrototypeOf()` checks whether an object is prototype for another object

[1] The `hasOwnProperty` method is present on all objects in JavaScript

Working with Properties

H) Using Prototypes with Constructors

- The shared nature of prototypes makes them ideal for defining methods once for all objects of a given type
- It is much more efficient to put the methods on the prototype and then use this to access the current instance
- A common pattern for defining prototype properties involves the notation as **object literal** (see example on the right side)
- An alternative notation for prototype properties is

```
Person.prototype.sayName = function() {
  console.log(this.name);
};
```

```
function Person(name) {
  this.name = name;
}

// value is an object literal
Person.prototype = {
  sayName: function() {
    console.log(this.name);
  },

  toString: function() {
    return "[Person " + this.name + "]";
  }
};
```