→ The JavaScript language → Classes

30th November 2019

Class checking: "instanceof"

The `instanceof` operator allows to check whether an object belongs to a certain class. It also takes inheritance into account.

Such a check may be necessary in many cases. Here we'll use it for building a *polymorphic* function, the one that treats arguments differently depending on their type.

The instanceof operator

The syntax is:

```
obj instanceof Class
```

It returns `true` if `obj` belongs to the `Class` or a class inheriting from it.

For instance:

```
class Rabbit {}
let rabbit = new Rabbit();

// is it an object of Rabbit class?
alert( rabbit instanceof Rabbit ); // true
```

It also works with constructor functions:

```
// instead of class
function Rabbit() {}

alert( new Rabbit() instanceof Rabbit ); // true
```

…And with built-in classes like `Array` :

```
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

Please note that `arr` also belongs to the `Object` class. That's because `Array` prototypically inherits from `Object` .

Normally, `instanceof` examines the prototype chain for the check. We can also set a custom logic in the static method `Symbol.hasInstance`.

The algorithm of `obj instanceof Class` works roughly as follows:

1. If there's a static method `Symbol.hasInstance`, then just call it: `Class[Symbol.hasInstance](obj)`. It should return either `true` or `false`, and we're done. That's how we can customize the behavior of `instanceof`.

   For example:

   ```
   // setup instanceOf check that assumes that
   // anything with canEat property is an animal
   class Animal {
     static [Symbol.hasInstance](obj) {
       if (obj.canEat) return true;
     }
   }

   let obj = { canEat: true };

   alert(obj instanceof Animal); // true: Animal[Symbol.hasInstance](obj) is
   ```

2. Most classes do not have `Symbol.hasInstance`. In that case, the standard logic is used: `obj instanceOf Class` checks whether `Class.prototype` is equal to one of the prototypes in the `obj` prototype chain.

   In other words, compare one after another:

   ```
   obj.__proto__ === Class.prototype?
   obj.__proto__.__proto__ === Class.prototype?
   obj.__proto__.__proto__.__proto__ === Class.prototype?
   ...
   // if any answer is true, return true
   // otherwise, if we reached the end of the chain, return false
   ```

   In the example above `rabbit.__proto__ === Rabbit.prototype`, so that gives the answer immediately.

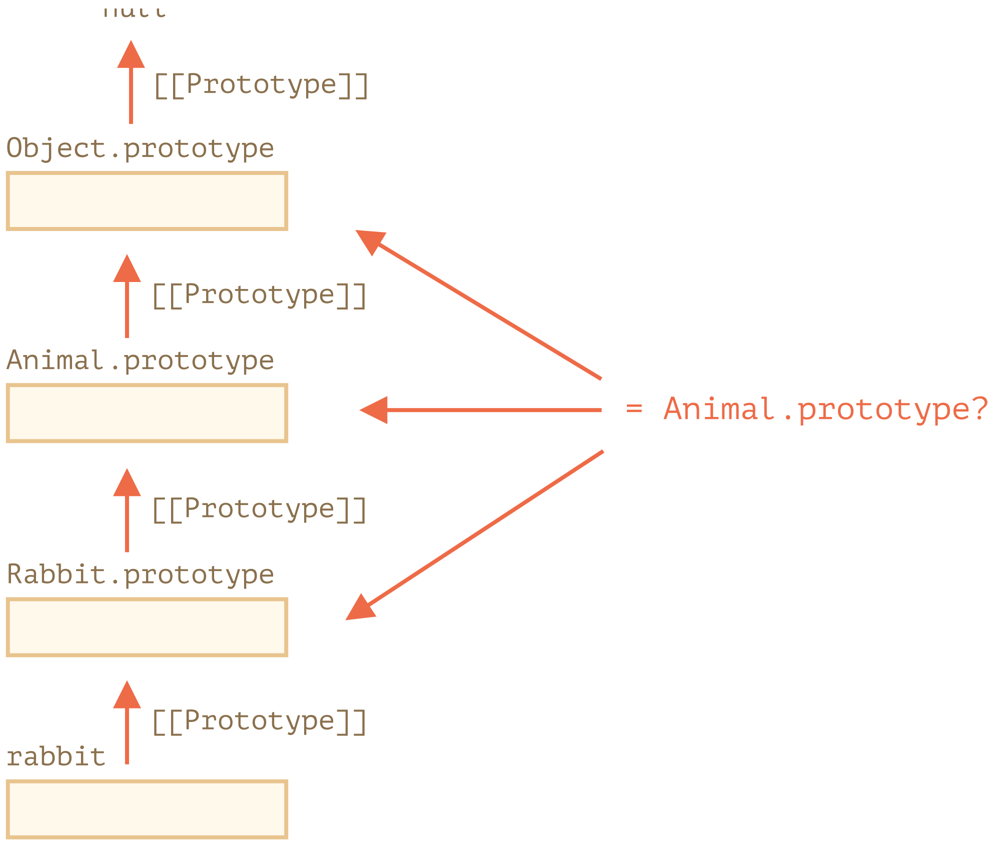
   In the case of an inheritance, the match will be at the second step:

   ```
   class Animal {}
   class Rabbit extends Animal {}

   let rabbit = new Rabbit();
   alert(rabbit instanceof Animal); // true

   // rabbit.__proto__ === Rabbit.prototype
   // rabbit.__proto__.__proto__ === Animal.prototype (match!)
   ```

Here's the illustration of what `rabbit instanceof Animal` compares with `Animal.prototype`:



By the way, there's also a method objA.isPrototypeOf(objB), that returns `true` if `objA` is somewhere in the chain of prototypes for `objB`. So the test of `obj instanceof Class` can be rephrased as `Class.prototype.isPrototypeOf(obj)`.

It's funny, but the `Class` constructor itself does not participate in the check! Only the chain of prototypes and `Class.prototype` matters.

That can lead to interesting consequences when a `prototype` property is changed after the object is created.

Like here:

```
function Rabbit() {}
let rabbit = new Rabbit();

// changed the prototype
Rabbit.prototype = {};

// ...not a rabbit any more!
alert( rabbit instanceof Rabbit ); // false
```

Bonus: Object.prototype.toString for the type

We already know that plain objects are converted to string as `[object Object]`:

```
let obj = {};

alert(obj); // [object Object]
alert(obj.toString()); // the same
```

That's their implementation of `toString`. But there's a hidden feature that makes `toString` actually much more powerful than that. We can use it as an extended `typeof` and an alternative for `instanceof`.

Sounds strange? Indeed. Let's demystify.

By specification, the built-in `toString` can be extracted from the object and executed in the context of any other value. And its result depends on that value.

- For a number, it will be `[object Number]`
- For a boolean, it will be `[object Boolean]`
- For `null`: `[object Null]`
- For `undefined`: `[object Undefined]`
- For arrays: `[object Array]`
- …etc (customizable).

Let's demonstrate:

```
// copy toString method into a variable for convenience
let objectToString = Object.prototype.toString;

// what type is this?
let arr = [];

alert( objectToString.call(arr) ); // [object Array]
```

Here we used call as described in the chapter Decorators and forwarding, call/apply to execute the function `objectToString` in the context `this=arr`.

Internally, the `toString` algorithm examines `this` and returns the corresponding result. More examples:

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTag

The behavior of Object `toString` can be customized using a special object property `Symbol.toStringTag`.

For instance:

```
let user = {
  [Symbol.toStringTag]: "User"
};

alert( {}.toString.call(user) ); // [object User]
```

For most environment-specific objects, there is such a property. Here are some browser specific examples:

```
// toStringTag for the environment-specific object and class:
alert( window[Symbol.toStringTag]); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

As you can see, the result is exactly `Symbol.toStringTag` (if exists), wrapped into `[object ...]`.

At the end we have “typeof on steroids” that not only works for primitive data types, but also for built-in objects and even can be customized.

We can use `{}.toString.call` instead of `instanceof` for built-in objects when we want to get the type as a string rather than just to check.

Summary

Let’s summarize the type-checking methods that we know:

	works for	returns
`typeof`	primitives	string
`{}.toString`	primitives, built-in objects, objects with `Symbol.toStringTag`	string
`instanceof`	objects	true/false

As we can see, `{}.toString` is technically a “more advanced” `typeof`.

And `instanceof` operator really shines when we are working with a class hierarchy and want to check for the class taking into account inheritance.

✓ Tasks

Strange instanceof

importance: 5

In the code below, why does `instanceof` return `true`? We can easily see that `a` is not created by `B()`.

```
function A() {}
function B() {}

A.prototype = B.prototype = {};

let a = new A();

alert( a instanceof B ); // true
```

solution

Previous lesson

Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)