Проверка класса: "instanceof"

Оператор `instanceof` позволяет проверить, к какому классу принадлежит объект, с учётом наследования.

Такая проверка может потребоваться во многих случаях. Здесь мы используем её для создания *полиморфной* функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

Оператор instanceof

Синтаксис:

```
obj instanceof Class
```

Оператор вернёт `true`, если `obj` принадлежит классу `Class` или наследующему от него.

Например:

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

// это объект класса Rabbit?
alert( rabbit instanceof Rabbit ); // true
```

Также это работает с функциями-конструкторами:

```
// вместо класса
function Rabbit() {}

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

...И для встроенных классов, таких как `Array`:

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

Пожалуйста, обратите внимание, что `arr` также принадлежит классу `Object`, потому что `Array` наследует от `Object`.

Обычно оператор `instanceof` просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода `Symbol.hasInstance`.

Алгоритм работы `obj instanceof Class` работает примерно так:

1. Если имеется статический метод `Symbol.hasInstance`, тогда вызвать его: `Class[Symbol.hasInstance](obj)`. Он должен вернуть либо `true`, либо `false`, и это конец. Это как раз и есть возможность ручной настройки `instanceof`.

   Пример:

   ```
   // проверка instanceof будет полагать,
   // что всё со свойством canEat - животное Animal
   class Animal {
     static [Symbol.hasInstance](obj) {
       if (obj.canEat) return true;
   ```

```
  }
}

let obj = { canEat: true };
alert(obj instanceof Animal); // true: вызван Animal[
```

2. Большая часть классов не имеет метода `Symbol.hasInstance`. В этом случае используется стандартная логика: проверяется, равен ли `Class.prototype` одному из прототипов в прототипной цепочке `obj`.

   Другими словами, сравнивается:

```
obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype
...
// если какой-то из ответов true - возвратить true
// если дошли до конца цепочки - false
```

   В примере выше `rabbit.__proto__ === Rabbit.prototype`, так что результат будет получен немедленно.

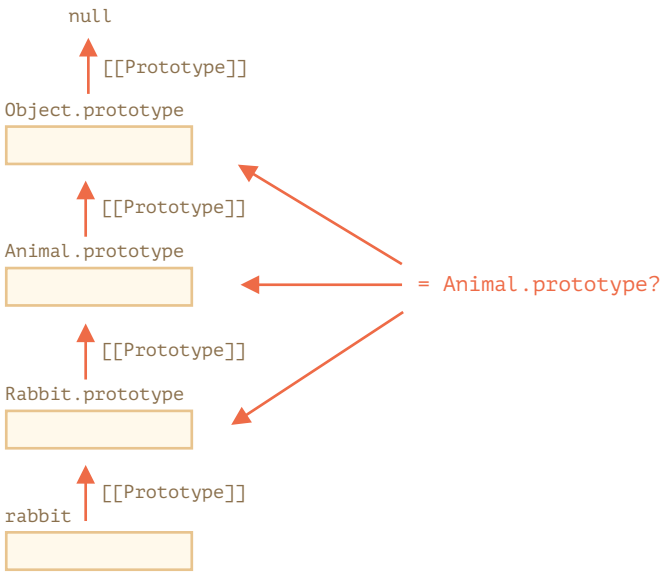
   В случае с наследованием, совпадение будет на втором шаге:

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

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

// rabbit.__proto__ === Rabbit.prototype
// rabbit.__proto__.__proto__ === Animal.prototype (c
```

Вот иллюстрация того как `rabbit instanceof Animal` сравнивается с `Animal.prototype`:



Кстати, есть метод objA.isPrototypeOf(objB), который возвращает `true`, если объект `objA` есть где-то в прототипной цепочке объекта `objB`. Так что `obj instanceof Class` можно перефразировать как `Class.prototype.isPrototypeOf(obj)`.

Забавно, но сам конструктор `Class` не участвует в процессе проверки! Важна только цепочка прототипов `Class.prototype`.

Это может приводить к интересным последствиям при изменении свойства `prototype` после создания объекта.

Как, например, тут:

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

// заменяем прототип
Rabbit.prototype = {};

// ...больше не rabbit!
alert( rabbit instanceof Rabbit ); // false
```

Бонус: Object.prototype.toString возвращает тип

Мы уже знаем, что обычные объекты преобразуется к строке как `[object Object]`:

```
let obj = {};

alert(obj); // [object Object]
alert(obj.toString()); // то же самое
```

Так работает реализация метода `toString`. Но у `toString` имеются скрытые возможности, которые делают метод гораздо более мощным. Мы можем использовать его как расширенную версию `typeof` и как альтернативу `instanceof`.

Звучит странно? Так и есть. Давайте развеем мистику.

Согласно спецификации встроенный метод `toString` может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

- Для числа это будет `[object Number]`
- Для булева типа это будет `[object Boolean]`
- Для `null`: `[object Null]`
- Для `undefined`: `[object Undefined]`
- Для массивов: `[object Array]`
- ...и т.д. (поведение настраивается).

Давайте продемонстрируем:

```
// скопируем метод toString в переменную для удобства
let objectToString = Object.prototype.toString;

// какой это тип?
let arr = [];

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

В примере мы использовали call, как описано в главе Декораторы и переадресация вызова, call/apply, чтобы выполнить функцию `objectToString` в контексте `this=arr`.

Внутри, алгоритм метода `toString` анализирует контекст вызова `this` и возвращает соответствующий результат. Больше примеров:

```
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

Поведение метода объектов `toString` можно настраивать, используя специальное свойство объекта `Symbol.toStringTag`.

Например:

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

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

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

```
// toStringTag для браузерного объекта и класса
alert( window[Symbol.toStringTag]); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] );

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

Как вы можете видеть, результат – это значение `Symbol.toStringTag` (если он имеется) обёрнутое в `[object ...]`.

В итоге мы получили «typeof на стероидах», который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать `{}.toString.call` вместо `instanceof` для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.

Итого

Давайте обобщим, какие методы для проверки типа мы знаем:

	работает для	возвращает
`typeof`	примитивов	строка
`{}.toString`	примитивов, встроенных объектов, объектов с `Symbol.toStringTag`	строка
`instanceof`	объектов	true/false

Как мы можем видеть, технически `{}.toString` «более продвинут», чем `typeof`.

А оператор `instanceof` – отличный выбор, когда мы работаем с иерархией классов и хотим делать проверки с учётом наследования.

Задачи

Странный instanceof

важность: 5

Почему `instanceof` в примере ниже возвращает `true`? Мы же видим, что `a` не создан с помощью `B()`.

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

A.prototype = B.prototype = {};

let a = new A();

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

решение

Комментарии

перед тем как писать...