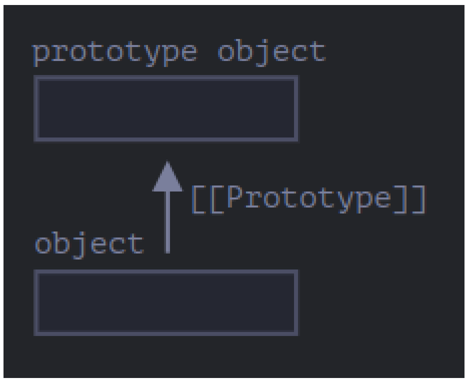
6 - Prototypal inheritance

Prototypal inheritance

In programming, we often want to take something and extend it. For instance, we have a user object with its properties and methods, and want to make admin and guest as slightly modified variants of it. We'd like to reuse what we have in user, not copy/reimplement its methods, just build a new object on top of it. **Prototypal inheritance** is a language feature that helps in that.

[[Prototype]]

In JavaScript, objects have a special hidden property [[Prototype]] (as named in the specification), that is either null or references another object. That object is called "a prototype":



When we read a property from object, and it's missing, JavaScript automatically takes it from the prototype. In programming, this is called "prototypal inheritance". And soon we'll study many examples of such inheritance, as well as cooler language features built upon it. The property [[Prototype]] is internal and hidden, but there are many ways to set it.

One of them is to use the special name **proto**, like this:

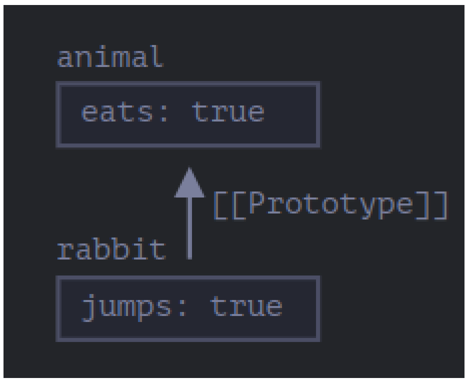
```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // sets rabbit.[[Prototype]] = animal
```

Now if we read a property from rabbit, and it's missing, JavaScript will automatically take it from animal.

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// we can find both properties in rabbit now:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
```

Here the line (*) sets animal to be the prototype of rabbit.

Then, when alert tries to read property rabbit.eats (**), it's not in rabbit, so JavaScript follows the [[Prototype]] reference and finds it in animal (look from the bottom up):



Here we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal". So if animal has a lot of useful properties and methods, then they become automatically available in rabbit. Such properties are called "inherited".

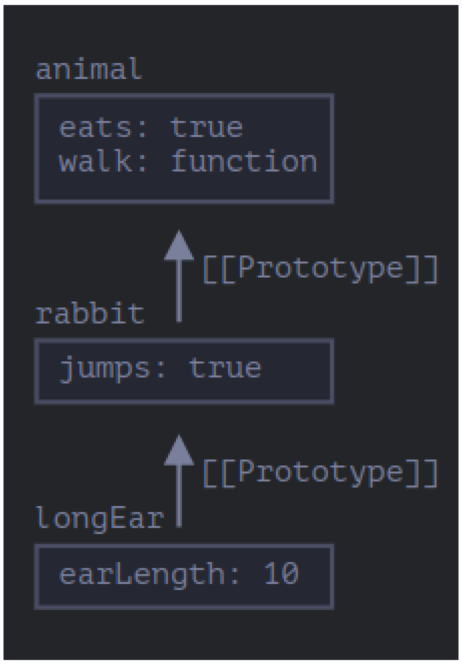
If we have a method in animal, it can be called on rabbit:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};
```

```
// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

The method is automatically taken from the prototype. THe prototype chain can be longer:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```



Now if we read something from longEar, and it’s missing, JavaScript will look for it in rabbit, and then in animal.
There are only two limitations:

- The references can’t go in circles. JavaScript will throw an error if we try to assign **proto** in a circle.
- The value of **proto** can be either an object or null. Other types are ignored.
  Also it may be obvious, but still: there can be only one [[Prototype]]. **An object may not inherit from two others.**

proto is a historical getter/setter for [[Prototype]]
It’s a common mistake of novice developers not to know the difference between these two.

Please note that **proto** is not the same as the internal [[Prototype]] property. It’s a getter/setter for [[Prototype]]. Later we’ll see situations where it matters, for now let’s just keep it in mind, as we build our understanding of JavaScript language.

The **proto** property is a bit outdated. It exists for historical reasons, modern JavaScript suggests that we should use Object.getPrototypeOf/Object.setPrototypeOf functions instead that get/set the prototype. We’ll also cover these functions later. By the specification, **proto** must only be supported by browsers. In fact though, all environments including server-side support **proto**, so we’re quite safe using it. As the **proto** notation is a bit more intuitively obvious, we use it in the examples.

Writing does not use prototypes

The prototype is only used for reading properties.
Write/delete operations work directly with the object.
In the example below, we assign its own walk method to rabbit:

```
let animal = {
  eats: true,
  walk() {
    /* this method won't be used by rabbit */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

From now on, rabbit.walk() call finds the method immediately in the object and executes it, without using the prototype. Accessor properties are an exception, as assignment is handled by a setter function. So writing to such a property is actually the same as calling a function.

```
let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// setter triggers!
admin.fullName = "Alice Cooper"; // (**)

alert(admin.fullName); // Alice Cooper, state of admin modified
alert(user.fullName); // John Smith, state of user protected
```

Here in the line (*) the property admin.fullName has a getter in the prototype user, so it is called. And in the line (**) the property has a setter in the prototype, so it is called.

Value of this

An interesting question may arise in the example above: what's the value of this inside set fullName(value)? Where are the properties this.name and this.surname written: into user or admin?
The answer is simple: this is not affected by prototypes at all.

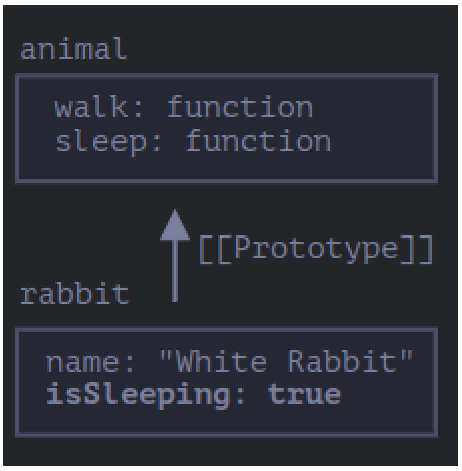
No matter where the method is found: in an object or its prototype. In a method call, this is always the object before the dot.

So, the setter call admin.fullName= uses admin as this, not user.
That is actually a super-important thing, because we may have a big object with many methods, and have objects that inherit from it. And when the inheriting objects run the inherited methods, they will modify only their own states, not the state of the big object.
For instance, here animal represents a "method storage", and rabbit makes use of it.

The call rabbit.sleep() sets this.isSleeping on the rabbit object:

```
// animal has methods
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// modifies rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (no such property in the prototype)
```

This is the result:



If we had other objects, like bird, snake, etc., inheriting from animal, they would also gain access to methods of animal. But this in each method call would be the corresponding object, evaluated at the call-time (before dot), not animal. So when we write data into this, it is stored into these objects.

As a result, methods are shared, but the object state is not.

for...in loop

The for..in loop iterates over inherited properties too.

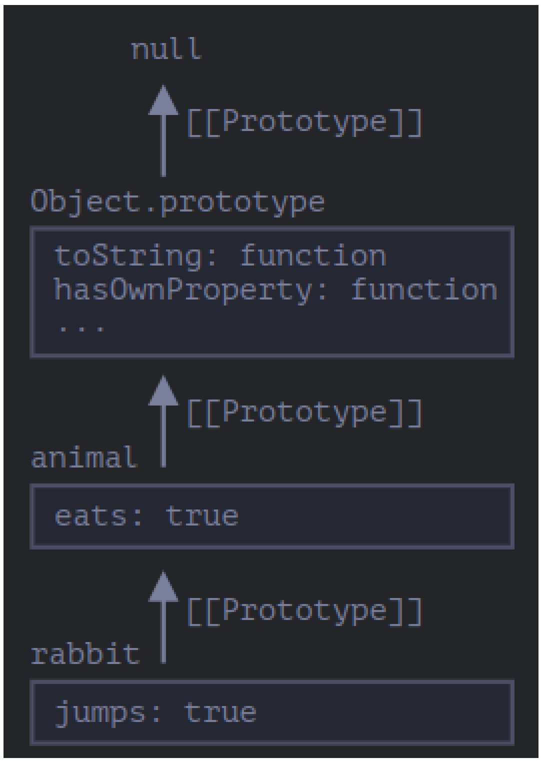
```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys only returns own keys
alert(Object.keys(rabbit)); // jumps

// for..in loops over both own and inherited keys
for(let prop in rabbit) alert(prop); // jumps, then eats
```

If that's not what we want, and we'd like to exclude inherited properties, there's a built-in method obj.hasOwnProperty(key): it returns true if obj has its own (not inherited) property named key.

So we can filter out inherited properties (or do something else with them):

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}
```

Here we have the following inheritance chain: rabbit inherits from animal, that inherits from Object.prototype (because animal is a literal object {...}, so it's by default), and then null above it:



Note, there's one funny thing. Where is the method rabbit.hasOwnProperty coming from? We did not define it. Looking at the chain we can see that the method is provided by Object.prototype.hasOwnProperty. In other words, it's inherited. But why does hasOwnProperty not appear in the for..in loop like eats and jumps do, if for..in lists inherited properties?
The answer is simple: **it's not enumerable.** Just like all other properties of Object.prototype, it has enumerable:false flag. And for..in only lists enumerable properties. That's why it and the rest of the Object.prototype properties are not listed.
Almost all other key/value-getting methods ignore inherited properties, such as Object.keys, Object.values and so on ignore inherited properties.
They only operate on the object itself. Properties from the prototype are not taken into account.

Summary

- In JavaScript, all objects have a hidden [[Prototype]] property that's either another object or null.
- We can use obj.**proto** to access it (a historical getter/setter, there are other ways, to be covered soon).
- The object referenced by [[Prototype]] is called a "prototype".
- If we want to read a property of obj or call a method, and it doesn't exist, then JavaScript tries to find it in the prototype.
- Write/delete operations act directly on the object, they don't use the prototype (assuming it's a data property, not a setter).
- If we call obj.method(), and the method is taken from the prototype, this still references obj. So methods always work with the current object even if they are inherited.
- The for..in loop iterates over both its own and its inherited properties. All other key/value-getting methods only operate on the object itself.

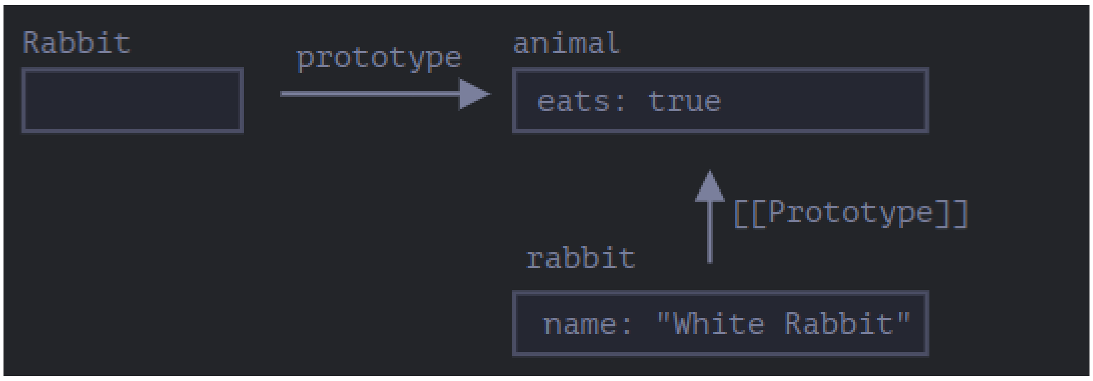
Exercises for this section can be found here: https://javascript.info/prototype-inheritance#tasks

F.prototype

Remember, new objects can be created with a constructor function, like new F().
If F.prototype is an object, then the new operator uses it to set [[Prototype]] for the new object.
JavaScript had prototypal inheritance from the beginning. It was one of the core features of the language.
But in the old times, there was no direct access to it. The only thing that worked reliably was a "prototype" property of the constructor function, described in this chapter. So there are many scripts that still use it. Please note that F.prototype here means a regular property named "prototype" on F. It sounds something similar to the term “prototype”, but here we really mean a regular property with this name.

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal

alert( rabbit.eats ); // true
```

Setting Rabbit.prototype = animal literally states the following: "When a new Rabbit is created, assign its [[Prototype]] to animal".



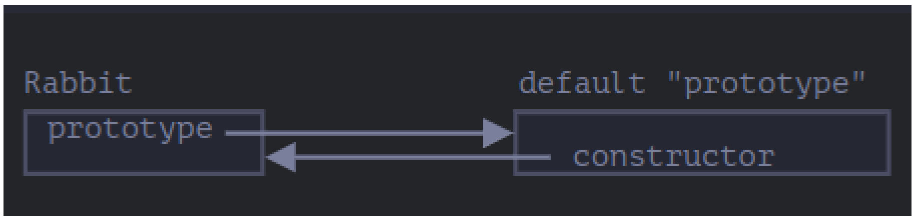
On the picture, "prototype" is a horizontal arrow, meaning a regular property, and [[Prototype]] is vertical, meaning the inheritance of rabbit from animal. F.prototype property is only used when new F is called, it assigns [[Prototype]] of the new object.
If, after the creation, F.prototype property changes (F.prototype = <another object>), then new objects created by new F will have another object as [[Prototype]], but already existing objects keep the old one.

Default F.prototype, constructor property

Every function has the "prototype" property even if we don't supply it.
The default "prototype" is an object with the only property constructor that points back to the function itself.

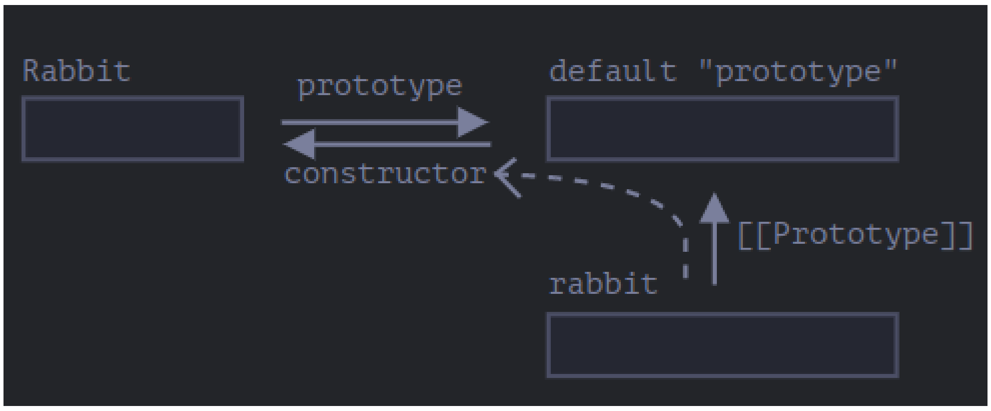
Like this:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/

alert( Rabbit.prototype.constructor == Rabbit ); // true
```



Naturally, if we do nothing, the constructor property is available to all rabbits through [[Prototype]]:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // true (from prototype)
```



We can use constructor property to create a new object using the same constructor as the existing one.

```
function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");
```

That’s handy when we have an object, don’t know which constructor was used for it (e.g. it comes from a 3rd party library), and we need to create another one of the same kind.
But probably the most important thing about "constructor" is that…
…JavaScript itself does not ensure the right "constructor" value.
Yes, it exists in the default "prototype" for functions, but that’s all. What happens with it later – is totally on us.
In particular, if we replace the default prototype as a whole, then there will be no "constructor" in it.

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

So, to keep the right "constructor" we can choose to add/remove properties to the default "prototype" instead of overwriting it as a whole:

```
function Rabbit() {}

// Not overwrite Rabbit.prototype totally
// just add to it
Rabbit.prototype.jumps = true
// the default Rabbit.prototype.constructor is preserved
// Another way is to recreate the constructor property manually:

Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// now constructor is also correct, because we added it
```

Summary

In this chapter we briefly described the way of setting a [[Prototype]] for objects created via a constructor function. Later we’ll see more advanced programming patterns that rely on it.
Everything is quite simple, just a few notes to make things clear:

- The F.prototype property (don't mistake it for [[Prototype]]) sets [[Prototype]] of new objects when new F() is called.
- The value of F.prototype should be either an object or null: other values won't work.
- The "prototype" property only has such a special effect when set on a constructor function, and invoked with new.
  On regular objects the prototype is nothing special:

```
let user = {
  name: "John",
  prototype: "Bla-bla" // no magic at all
};
```

By default all functions have F.prototype = { constructor: F }, so we can get the constructor of an object by accessing its "constructor" property.

Exercises for this section can be found at: https://javascript.info/function-prototype#tasks

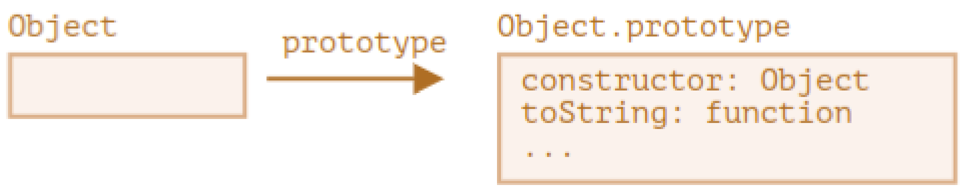
Native Prototypes

The prototype property is widely used by javascript itself. All built-in constructor funcion use it.

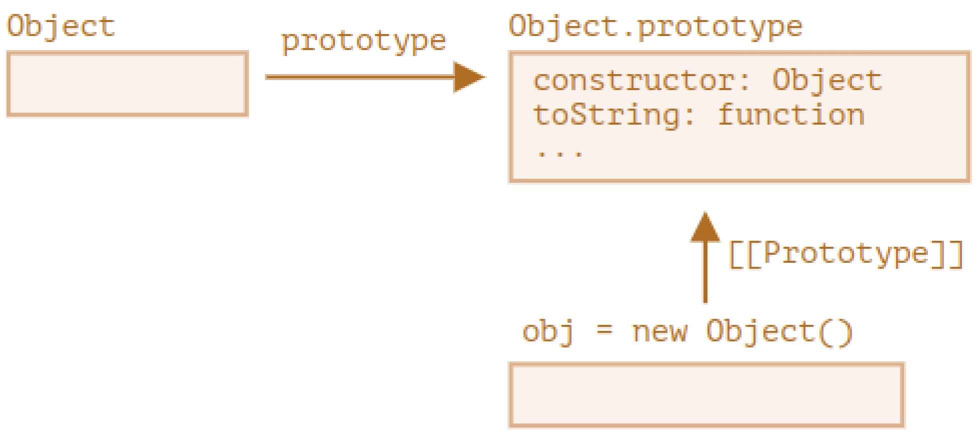
Object.prototype

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

Where's the code that generates the string "[object Object]"? That's a built-in toString method, but where is it? The obj is empty! But the short notation obj = {} is the same as obj = new Object(), where Object is a built-in object constructor function, with its own prototype referencing a huge object with toString and other methods.



When new Object() is called (or a literal object {...} is created), the [[Prototype]] of it is set to Object.prototype according to the rule that we discussed in the previous chapter:

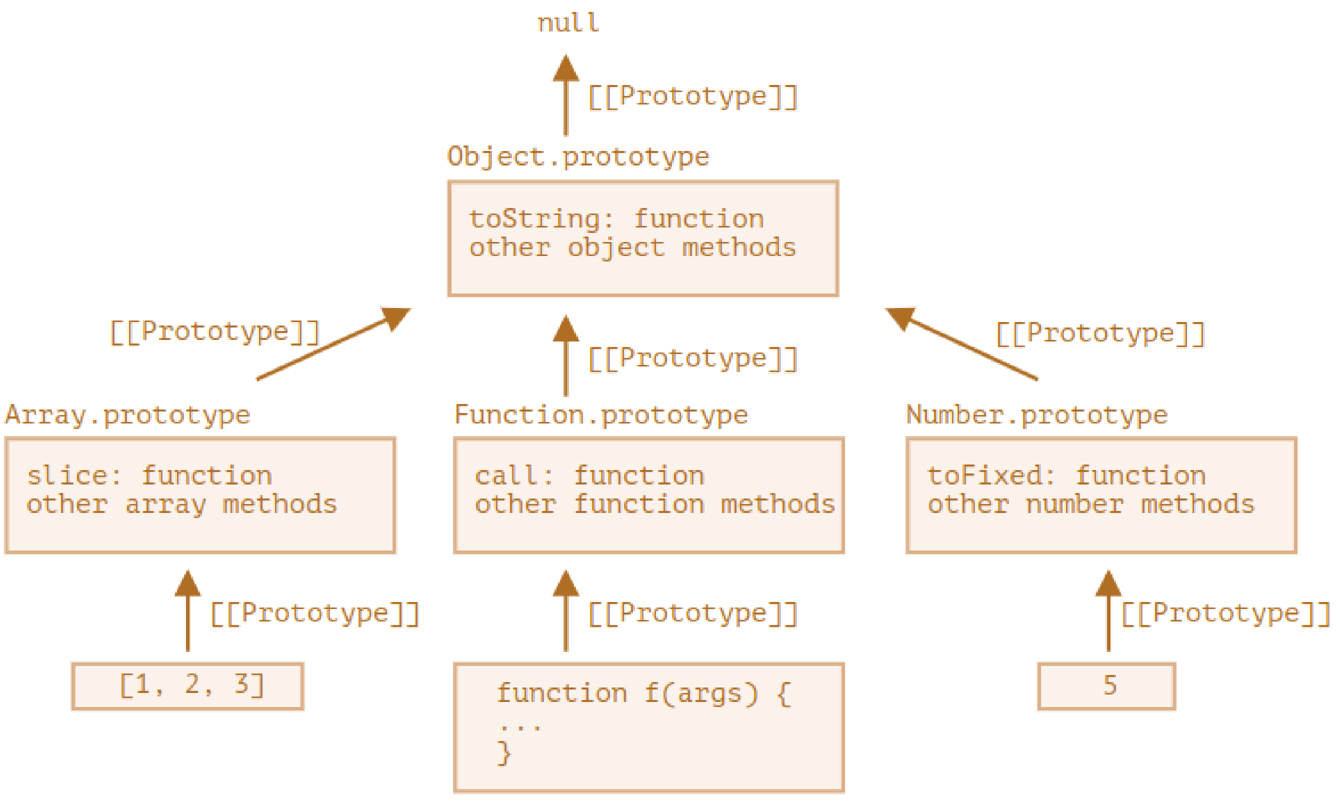


So then when obj.toString() is called the method is taken from Object.prototype

```
let obj = {};

alert(obj.__proto__ === Object.prototype); // true

alert(obj.toString === obj.__proto__.toString); //true
alert(obj.toString === Object.prototype.toString); //true
// Please note that there is no more [[Prototype]] in the chain above
Object.prototype:
alert(Object.prototype.__proto__); // null
```

Other built-in prototypes

Other built-in objects such as Array, Date, Function and others also keep methods in prototypes.
For instance, when we create an array [1, 2, 3], the default new Array() constructor is used internally.
So Array.prototype becomes its prototype and provides methods. That’s very memory-efficient.
By specification, all of the built-in prototypes have Object.prototype on the top. That’s why some people say that “everything inherits from objects”.
Here’s the overall picture (for 3 built-ins to fit):



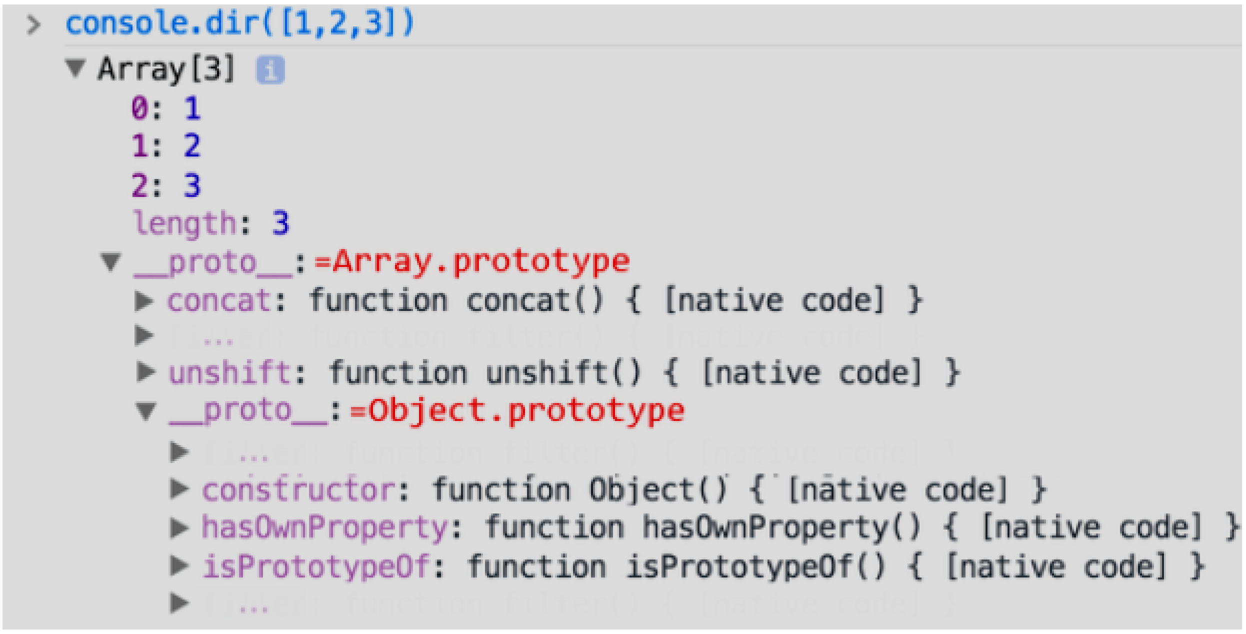
```
let arr = [1, 2, 3];

// it inherits from Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// then from Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// and null on the top.
alert( arr.__proto__.__proto__.__proto__ ); // null
```

Some methods in prototypes may overlap, for instance, Array.prototype has its own toString that lists comma-delimited elements:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3  - The result of Array.prototype.toString
```

As we've seen before, Object.prototype has toString as well, but Array.prototype is closer in the chain, so the array variant is used.
In-browser tools like Chrome developer console also show inheritance (console.dir may need to be

used for built-in objects):



```
> console.dir([1,2,3])
  ▼ Array[3]
      0: 1
      1: 2
      2: 3
      length: 3
    ▼ __proto__: =Array.prototype
      ▶ concat: function concat() { [native code] }
      ▶ ...
      ▶ unshift: function unshift() { [native code] }
      ▼ __proto__: =Object.prototype
        ▶ ...
        ▶ constructor: function Object() { [native code] }
        ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
        ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
        ▶ ...
```

Other built-in objects also work the same way. Even functions – they are objects of a built-in Function constructor, and their methods (call/apply and others) are taken from Function.prototype. Functions have their own toString too.

```
function f() {}

alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true, inherit from
objects
```

Primitives

The most intricate things happens with strings, numbers and booleans. As we remember, they are not objects. But if we try to access their properties, temporary wrapper objects are created using built-in constructors String, Number and Boolean. They provide the methods and disappear.
These objects are created invisibly to us and most engines optimize them out, but the specification describes it exactly this way. Methods of these objects also reside in prototypes, available as String.prototype, Number.prototype and Boolean.prototype.
Special values null and undefined stand apart. They have no object wrappers, so methods and properties are not available for them. And there are no corresponding prototypes either.

Changing native prototypes

Native prototypes can be modified.

```
// Adding a method to all strings
String.prototype.show = function() {
  alert(this);
};
```

```
"BOOM!".show(); // BOOM!
```

During the process of development, we may have ideas for new built-in methods we'd like to have, and we may be tempted to add them to native prototypes. But that is generally a bad idea. Prototypes are global, so it's easy to get a conflict. If two libraries add a method String.prototype.show, then one of them will be overwriting the method of the other. So, generally, modifying a native prototype is considered a bad idea. In modern programming, there is only one case where modifying native prototypes is approved. That's polyfilling.
Polyfilling is a term for making a substitute for a method that exists in the JavaScript specification, but is not yet supported by a particular JavaScript engine. We may then implement it manually and populate the built-in prototype with it.

```
if (!String.prototype.repeat) { // if there's no such method
  // add it to the prototype

  String.prototype.repeat = function(n) {
    // repeat the string n times

    // actually, the code should be a little bit more complex than that
    // (the full algorithm is in the specification)
    // but even an imperfect polyfill is often considered good enough
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Borrowing from prototypes

In the chapter Decorators and forwarding, call/apply we talked about method borrowing. That's when we take a method from one object and copy it into another. Some methods of native prototypes are often borrowed.
For instance, if we're making an array-like object, we may want to copy some Array methods to it.

```
let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!
```

It works because the internal algorithm of the built-in join method only cares about the correct indexes and the length property. It doesn’t check if the object is indeed an array. Many built-in methods are like that.
Another possibility is to inherit by setting obj.**proto** to Array.prototype, so all Array methods are automatically available in obj. But that’s impossible if obj already inherits from another object. Remember, we only can inherit from one object at a time. Borrowing methods is flexible, it allows to mix functionalities from different objects if needed.

Summary

- All built-in objects follow the same pattern:
  - The methods are stored in the prototype (Array.prototype, Object.prototype, Date.prototype, etc.)
  - The object itself stores only the data (array items, object properties, the date)
- Primitives also store methods in prototypes of wrapper objects: Number.prototype, String.prototype and Boolean.prototype. Only undefined and null do not have wrapper objects
- Built-in prototypes can be modified or populated with new methods. But it’s not recommended to change them. The only allowable case is probably when we add-in a new standard, but it’s not yet supported by the JavaScript engine

Exercises for this section can be found at : https://javascript.info/native-prototypes#tasks

Prototype methods, objects without "__proto__"

In the first chapter of this section, we mentioned that there are modern methods to setup a prototype.

Setting or reading the prototype with obj.**proto** is considered outdated and somewhat deprecated (moved to the so-called “Annex B” of the JavaScript standard, meant for browsers only).

The modern methods to get/set a prototype are:

- Object.getPrototypeOf(obj) – returns the [[Prototype]] of obj.
- Object.setPrototypeOf(obj, proto) – sets the [[Prototype]] of obj to proto.

The only usage of **proto**, that’s not frowned upon, is as a property when creating a new object: { **proto**: ... }.

Although, there’s a special method for this too:

- Object.create(proto, [descriptors]) – creates an empty object with given proto as [[Prototype]] and optional property descriptors.

For instance:

```
let animal = {
  eats: true
};

// create a new object with animal as a prototype
let rabbit = Object.create(animal); // same as {__proto__: animal}

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}
```

The Object.create method is a bit more powerful, as it has an optional second argument: property descriptors.
We can provide additional properties to the new object there, like this:

```
let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

The descriptors are in the same format as described in the chapter Property flags and descriptors. We can use Object.create to perform an object cloning more powerful than copying properties in for..in:

```
let clone = Object.create(
  Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj)
);
```

This call makes a truly exact copy of obj, including all properties: enumerable and non-enumerable, data properties and setters/getters – everything, and with the right [[Prototype]].

Don't change [[Prototype]] on existing objects if speed matters : Technically, we can get/set [[Prototype]] at any time. But usually we only set it once at the object creation time and don't modify it anymore: rabbit inherits from animal, and that is not going to change. And JavaScript engines are highly optimized for this. Changing a prototype "on-the-fly" with Object.setPrototypeOf or obj.**proto**= is a very slow operation as it breaks internal optimizations for object property access operations. So avoid it unless you know what you're doing, or JavaScript speed totally doesn't matter for you.

"Very plain" objects

As we know, objects can be used as associative arrays to store key/value pairs. But if we try to store user-provided keys in it (for instance, a user-entered dictionary), we can see an interesting glitch: all keys work fine except "**proto**". Check out the example:

```
let obj = {};

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // [object Object], not "some value"!
```

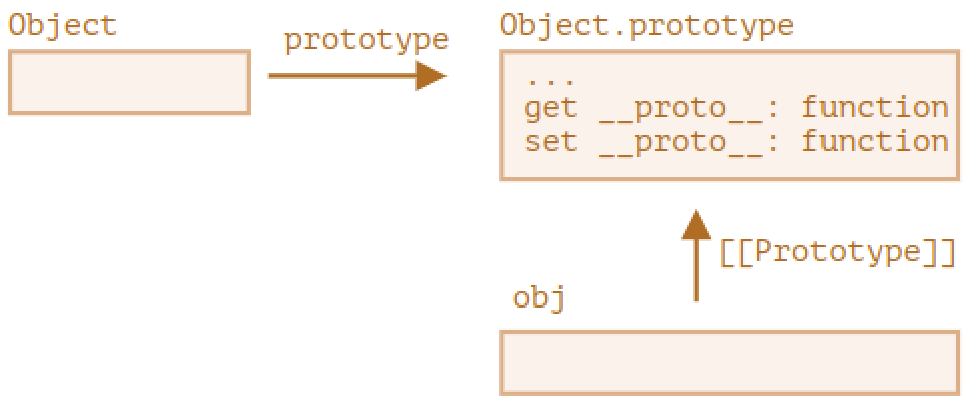
Here, if the user types in **proto**, the assignment in line 4 is ignored!

That could surely be surprising for a non-developer, but pretty understandable for us. The **proto** property is special: it must be either an object or null. A string can not become a prototype. That's why an assignment a string to **proto** is ignored. But we didn't intend to implement such behavior, right? We want to store key/value pairs, and the key named "**proto**" was not properly saved. So that's a bug! Here the consequences are not terrible. But in other cases we may be storing objects instead of strings in obj, and then the prototype will indeed be changed. As a result, the execution will go wrong in totally unexpected ways. What's worse – usually developers do not think about such possibility at all. That makes such bugs hard to notice and even turn them into vulnerabilities, especially when JavaScript is

used on server-side. Unexpected things also may happen when assigning to obj.toString, as it's a built-in object method.
How can we avoid this problem?
First, we can just switch to using Map for storage instead of plain objects, then everything's fine:

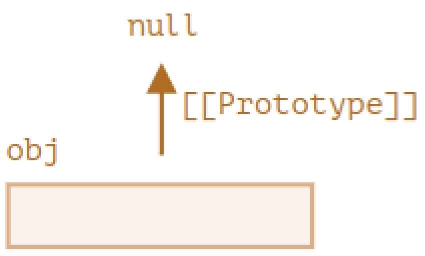
```
let map = new Map();

let key = prompt("What's the key?", "__proto__");
map.set(key, "some value");

alert(map.get(key)); // "some value" (as intended)
```

But Object syntax is often more appealing, as it's more concise.
Fortunately, we can use objects, because language creators gave thought to that problem long ago.
As we know, **proto** is not a property of an object, but an accessor property of Object.prototype:



So, if obj.**proto** is read or set, the corresponding getter/setter is called from its prototype, and it gets/sets [[Prototype]]. As it was said in the beginning of this tutorial section: **proto** is a way to access [[Prototype]], it is not [[Prototype]] itself. Now, if we intend to use an object as an associative array and be free of such problems, we can do it with a little trick:

```
let obj = Object.create(null);
// or: obj = { __proto__: null }

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // "some value"
```



So, there is no inherited getter/setter for **proto**. Now it is processed as a regular data property, so the example above works right.
We can call such objects “very plain” or “pure dictionary” objects, because they are even simpler than the regular plain object {...}. A downside is that such objects lack any built-in object methods, e.g. toString:

```
let obj = Object.create(null);

alert(obj); // Error (no toString)
```

Note that most object-related methods are Object.something(...), like Object.keys(obj) – they are not in the prototype, so they will keep working on such objects:

```
let chineseDictionary = Object.create(null);
chineseDictionary.hello = "你好";
chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye
```

Summary

- To create an object with the given prototype, use:
  - literal syntax: { **proto**: ... }, allows to specify multiple properties
  - or Object.create(proto, [descriptors]), allows to specify property descriptors.

The Object.create provides an easy way to shallow-copy an object with all descriptors:

```
let clone = Object.create(Object.getPrototypeOf(obj),
Object.getOwnPropertyDescriptors(obj));
```

- Modern methods to get/set the prototype are:
  - Object.getPrototypeOf(obj) – returns the [[Prototype]] of obj (same as **proto** getter).
  - Object.setPrototypeOf(obj, proto) – sets the [[Prototype]] of obj to proto (same as **proto** setter). Getting/setting the prototype using the built-in **proto** getter/setter isn’t recommended, it’s now in the Annex B of the specification.
- We also covered prototype-less objects, created with Object.create(null) or {**proto**: null}.
- These objects are used as dictionaries, to store any (possibly user-generated) keys.

- Normally, objects inherit built-in methods and **proto** getter/setter from Object.prototype, making corresponding keys “occupied” and potentially causing side effects. With null prototype, objects are truly empty.

Exercises for this section can be found at: https://javascript.info/prototype-methods#tasks