CS472 WAP
Constructor Function

Maharishi International University - Fairfield, Iowa

Wholeness: Inheritance is a fundamental feature of object-oriented programming. Common code is kept in a base component.   Specialized components ‘inherit’ the common code from the more general base component.

Science of Consciousness: An archetype is a fundamental pattern or law of nature that gives rise to many variations and realizations at more expressed levels of nature. Deeper levels of awareness make us more connected with these fundamental patterns.

Prototypal inheritance

- In programming, often want to take something and extend it.
  - user object with its properties and methods,
  - make admin and guest as slightly modified variants of it.
  - reuse what we have in user, not copy/reimplement its methods

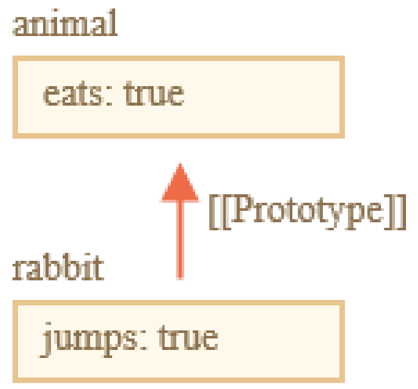
- Prototypal inheritance is a language feature that helps in that

[[Prototype]]

- every object has special hidden property `[[Prototype]]`
  - either null or references another object.
  - object is called “a prototype”:
  - Browsers implements using `__proto____`

- read a property from object, and it’s missing,
  - JavaScript automatically takes it from the prototype.
  - called “prototypal inheritance”.
  - property `[[Prototype]]` is internal and hidden, but there are many ways to set it.

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

Object.setPrototypeOf(rabbit, animal);
rabbit.__proto__ = animal;
```

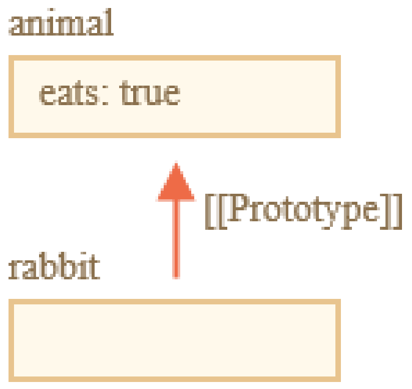


Object.create versus proto

- `__proto__` is considered outdated and “sort of” deprecated

- `Object.create(proto)` sets `[[Prototype]]/__proto__` without needing a constructor function.
  - creates an empty object with given proto as `[[Prototype]]`
  - `Object.create` should be used instead of `__proto__`

```
let animal = {
    eats: true
};
// create a new object with animal as a prototype
let rabbit = Object.create(animal);
alert(rabbit.eats); // true
```

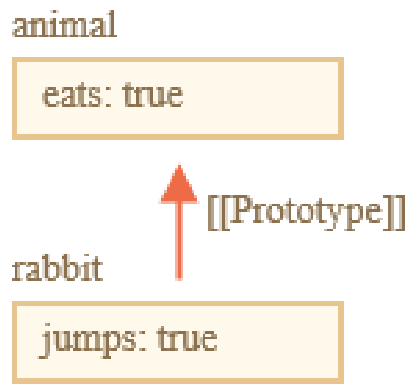


Inherit properties

- If look for a property in `rabbit`, and it's missing, JavaScript automatically takes it from `animal`.

- line (*) sets `animal` to be a prototype of `rabbit`.
- `alert` tries to read property `rabbit.eats (**)`,
  - it's not in `rabbit`,
  - JavaScript follows the `[[Prototype]]` reference and finds it in `animal`

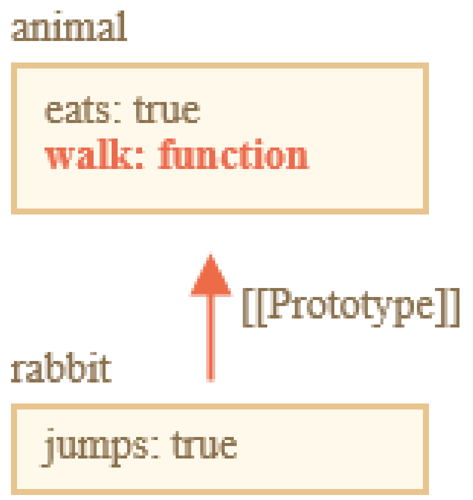
```
let   animal   = { eats: true };

let rabbit = Object.create(animal); //(*)
rabbit.jumps = true;

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



Inherit methods

- method in `animal`, it can be called on `rabbit`

```
let  animal  =  {
    eats:  true,
    walk:   function()  {
        alert("Animal walk");
    }
};
let  rabbit  =  Object.create(animal);
rabbit.jumps  =  true;

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



Prototype chain

- prototype chain can be longer
- restrictions:
  - references can’t go in circles..
  - value of `__proto__` can be either an `object` or `null`.
  - there can be only one `[[Prototype]]`. An object may not inherit from two others.

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

let rabbit = Object.create(animal);
rabbit.jumps = true;

let longEar = Object.create(rabbit);
longEar.earLength = 10;

longEar.walk();
```

Own properties do not use prototype chain

- Properties declared on an object work directly with the object
  - “shadow/override” anything further up the prototype chain

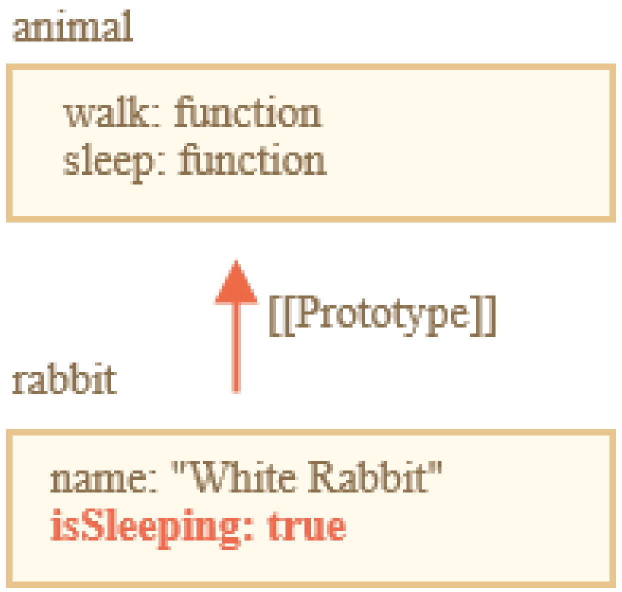
```
let animal = {
    eats: true,
    walk: function () {  /* this method won't be used by rabbit */ }
};
let rabbit = Object.create(animal);
rabbit.walk = function () {
    alert("Rabbit! Bounce-bounce!");
};
```

- From now on, `rabbit.walk()` call finds the method in the object without using prototype

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

methods often shared, object state generally not

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

let rabbit = Object.create(animal);
rabbit.name = "White Rabbit";

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



Constructor functions, operator “new”

- Object literal {...}syntax creates a single object.
- Often need to create many similar objects,
  - multiple users or menu items and so on.
- Use constructor functions and the "new" operator
- Constructor functions technically are regular functions.
- Two conventions:
  - start with capital letter
  - executed only with "new" operator

```
function User(name) {
   this.name = name;
   this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); //Jack
alert(user.isAdmin); // false
```

`new User(...)` does the following steps:

1. A new empty object is created and assigned to `this`.
2. The function body executes. Usually it modifies `this`, adds new properties to it.
3. The value of `this` is returned.

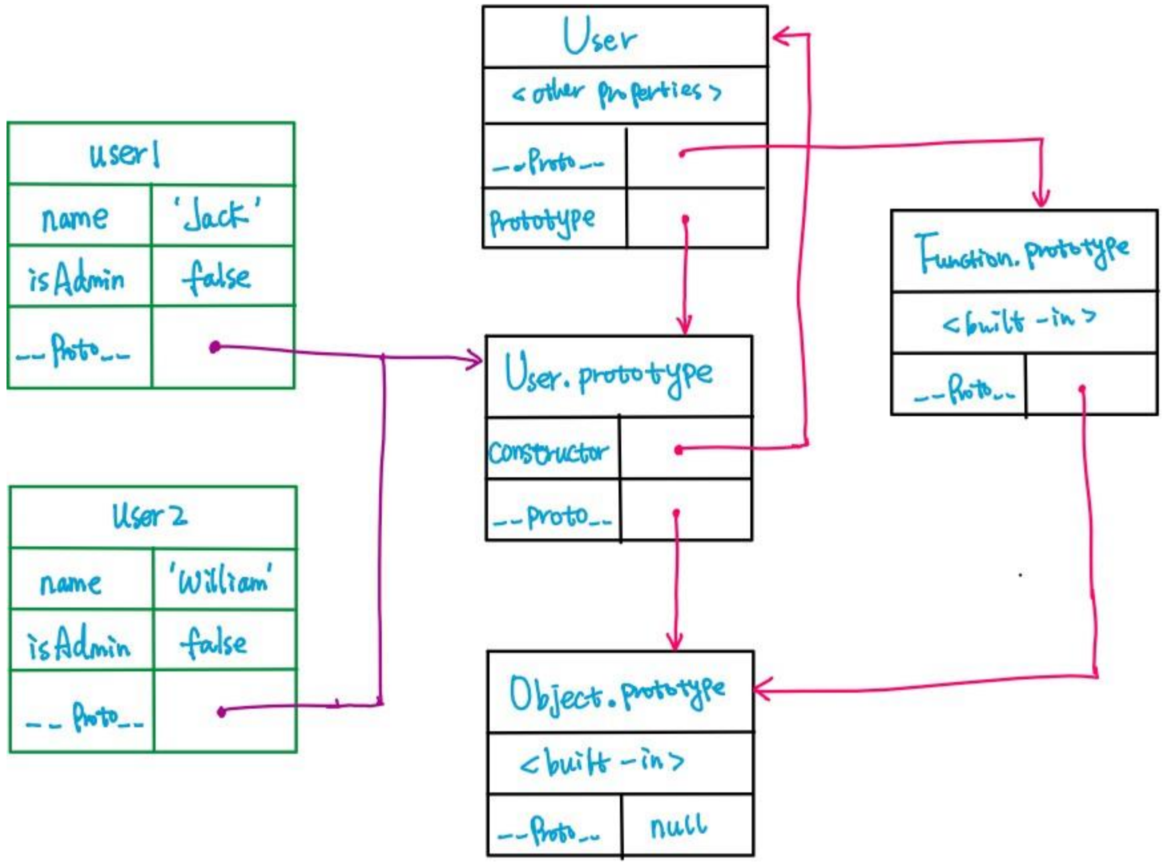
➢ In other words, `new User(...)` does something like:

```
function User(name) {
    // this = {};  (implicitly)

    // add properties to this
    this.name = name;
    this.isAdmin = false;

    // return this;  (implicitly)
}
new User('John');
```

F.prototype -- Set [[Prototype]] using constructor function

- `F.prototype` is a regular property named "`prototype`" on `F`.
  - This is not the 'special hidden' `[[Prototype]]/__proto__` property
- `F.prototype` is an object,
  - `new` operator uses it to set `[[Prototype]]/__proto__` for the new object.

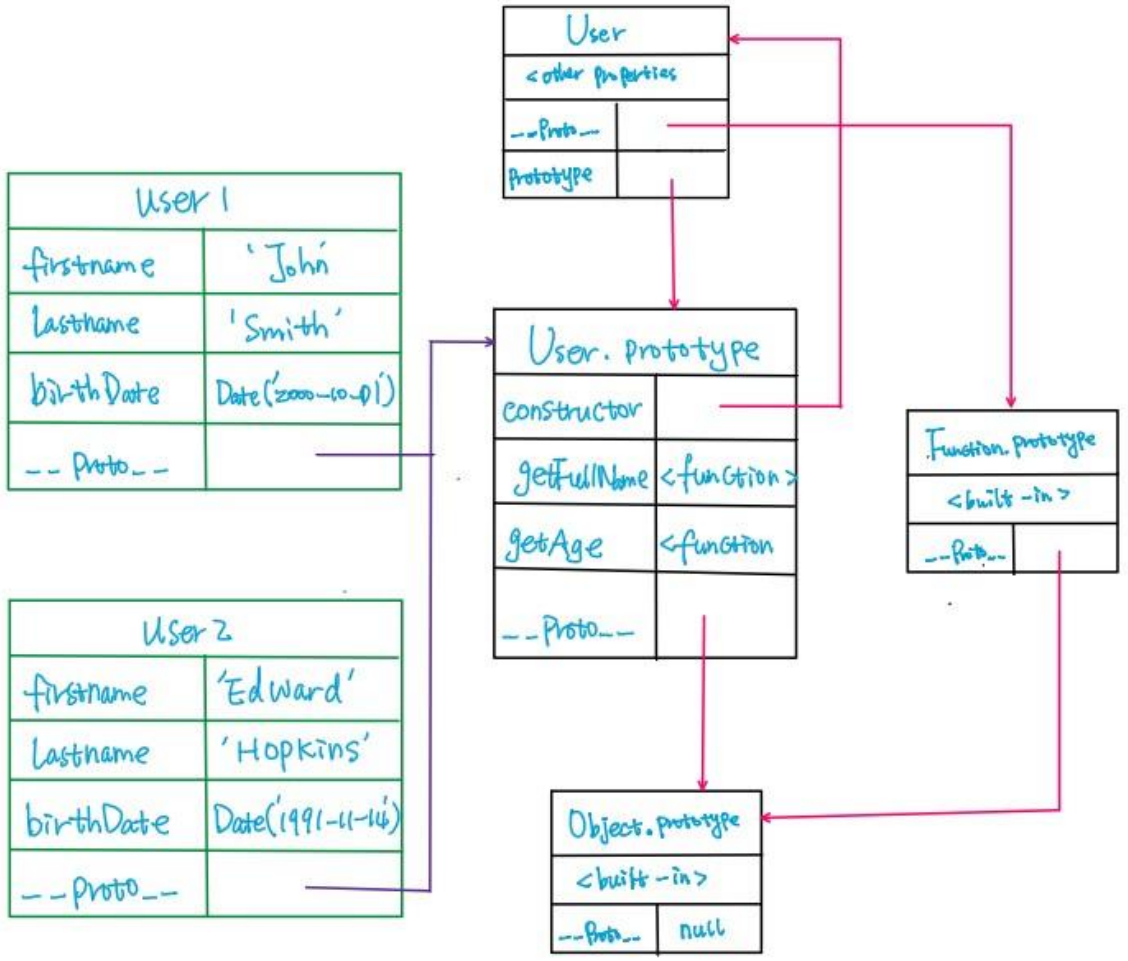
```
function User(name) {
    this.name = name;
    this.isAdmin = false;
}
let user1 = new User("Jack");
let user2 = new User("William");
```



Extend functionality using `F.prototype` property

- add/remove properties to default ‘`prototype`’ property

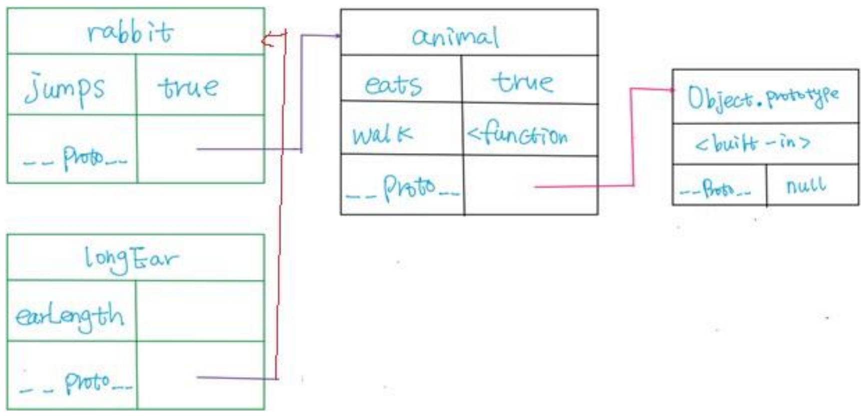
```
function User(firstname, lastname, birthDate) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthDate = birthDate;
}
let user1 = new User('John', 'Smith', new Date('2000-10-01'));
let user2 = new User('Edward', 'Hopkins', new Date('1991-11-14'));

User.prototype.getFullName = function() {
    return this.firstname + ' ' + this.lastname;
}

User.prototype.getAge = function() {
    return new Date().getFullYear() - this.birthDate.getFullYear();
}

console.log(user1.getFullName()); //John Smith
console.log(user1.getAge()); //21
```

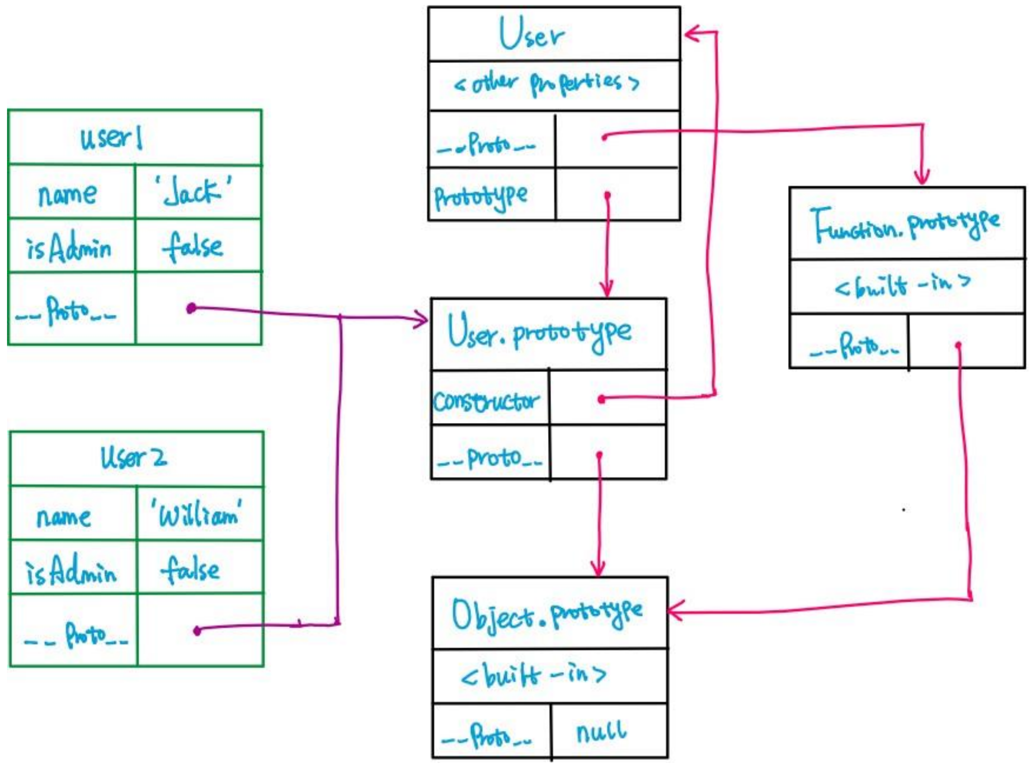


Constructor Function vs object literal

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

let rabbit = Object.create(animal);
rabbit.jumps = true;

let longEar = Object.create(rabbit);
longEar.earLength = 10;

longEar.walk();
```

```javascript
function User(name) {
    this.name = name;
    this.isAdmin = false;
}
let user1 = new User("Jack");
let user2 = new User("William");
```





Native prototypes

- "prototype" property is widely used by core of JavaScript
  - All built-in constructor functions use

```
const a = new Number(12);
const b = new String("Hello");
const c = new Date(2016, 03, 01);
```

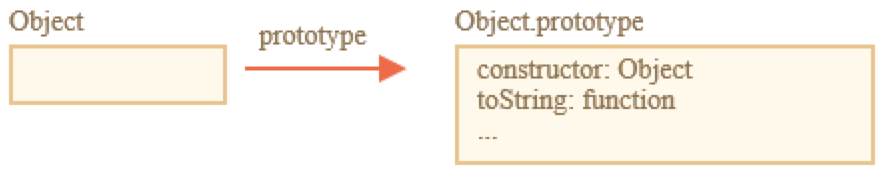
  - for adding new capabilities to built-in objects.
    - Define your own filter, map, etc functions in Array

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

- Where’s code that generates the "[object Object]"?
  - a built-in toString method, but where is it?

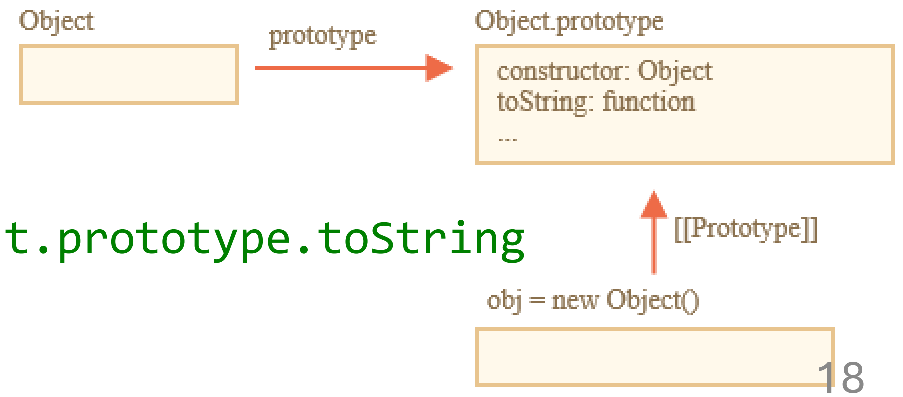
Object.prototype

- `obj = {}` is the same as `obj = new Object()`
  - Object is a built-in object constructor function,
  - prototype is huge object with toString and other methods.



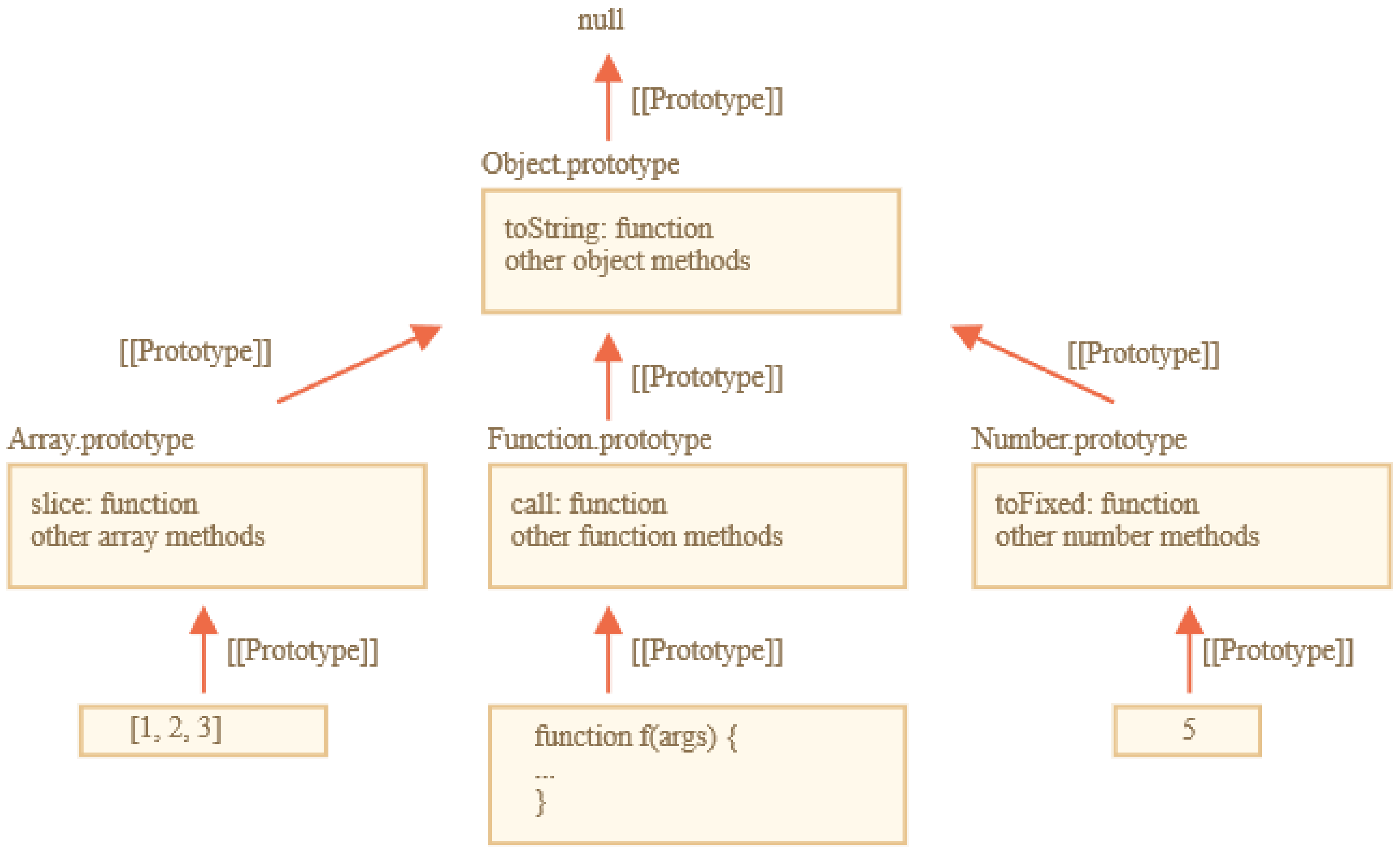
- When `new Object()` is called (or create object literal `{...}`)
  - `[[Prototype]]` of it is set to `Object.prototype`

  `obj.toString()` is inherited from `Object.prototype`.

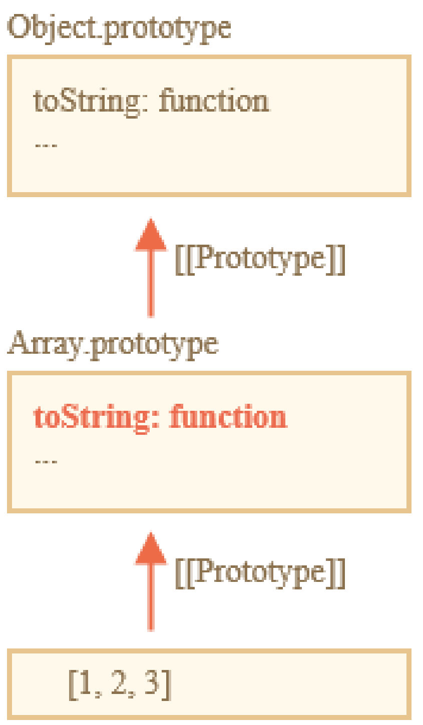


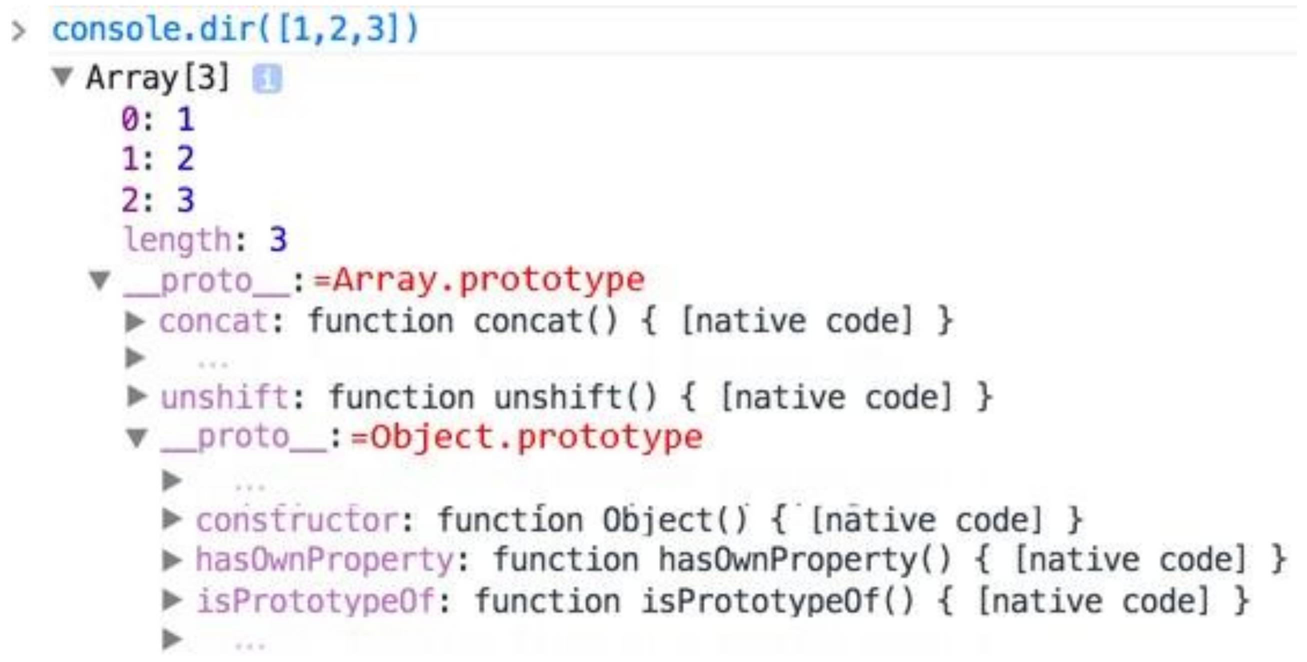
```
let obj = {};
alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString === Object.prototype.toString
```

Other built-in prototypes



JS object hierarchy





```
> 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] }
      ▶   ...
```

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Archetypal Patterns of Intelligence

1. JavaScript objects often share common methods through prototype chains.

2. Modern JavaScript sets up prototype chains using the prototype property of constructor functions and the Object.create method.

---

3. **Transcendental consciousness**. Is the experience of pure consciousness, the level of awareness that is the basis of all existence and all patterns of intelligence.
4. **Impulses within the transcendental field:** Thoughts arising from this level have direct access to the deepest patterns of intelligence of nature.
5. **Wholeness moving within itself:** In unity consciousness all levels of existence are perceived as expressions of these archetypal patterns of intelligence.