LESSON 8
PROTOTYPE INHERITANCE

Archetypal Patterns of Intelligence

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.

Main Points

1. Prototypal inheritance and [[Prototype]]
2. Setting prototypes with constructors and Object.create

Main Point Preview: Prototypal inheritance and Object.create

Prototypal inheritance allows object to inherit properties from a ‘prototype’ parent object. The main purpose of inheritance is to promote code reuse and avoid duplication. Science of Consciousness: Reuse of code for common tasks is efficient and avoids errors that can arise from inconsistent updates of duplicated code. Natural law takes the path of least action. Do less and accomplish more.

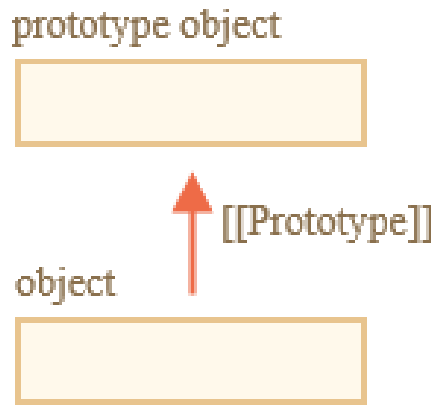
Programmers cannot directly access the special [[Prototype]] property. All functions have a regular ‘prototype’ property. When they are called as constructors with ‘new’ that property will be set as the value of [[Prototype]]. [[Prototype]] can also be set with the __proto__ property, but that is now deprecated in favor of Object.create. Science of Consciousness: JavaScript’s prototype is like “archetype”, which is an original object that is a basis for other objects. Deeper levels of thought are connected to archetypal patterns of intelligence or ‘laws of nature’.

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
};
rabbit.__proto__ = animal; // __proto__ is a ‘sneaky’ (deprecated) way to access
 [[Prototype]]
```

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__`

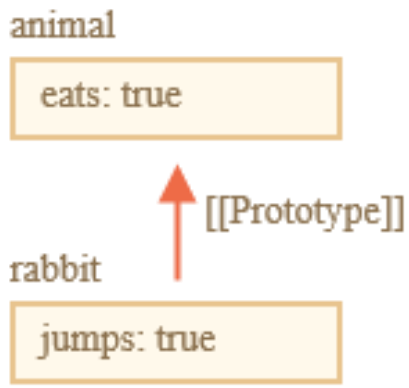
```
let animal = {
    eats: true
};
// create a new object with animal as a prototype
let rabbit = Object.create(animal);
alert(rabbit.eats); // true
```

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`

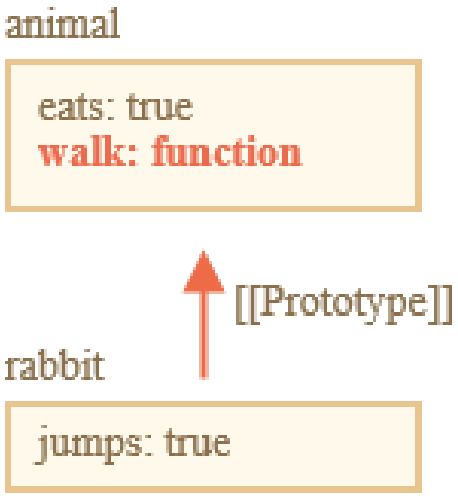
```
let  animal  = { eats: true };

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`

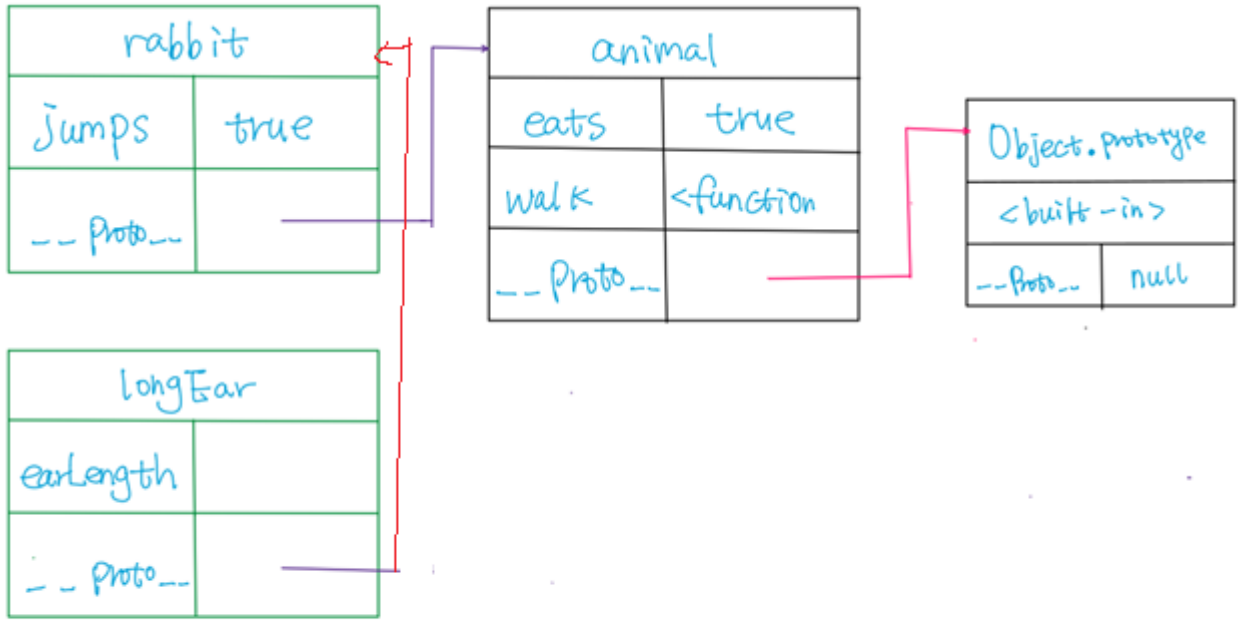
```
let  animal  =  {
    eats:  true,
    walk:  function()  {
        alert("Animal walk");
    }
};
let  rabbit  =  Object.create(animal);
rabbit.jumps  =  true;

// 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" 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!
```

The value of "`this`"

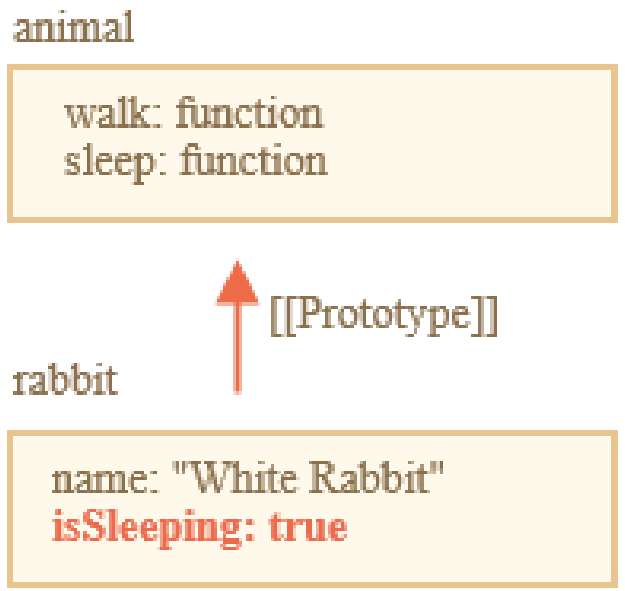
- what's the value of `this` inside an inherited method
  - answer: `this` is not affected by prototypes at all.
  - No matter where the method is found:
    - in an object or its prototype
    - `this` is always the object before the dot

- a super-important thing,
  - may have a big object with many methods and inherit from it.
  - descendent objects can run its methods, and they will modify their own state

- methods are often shared, but the object state generally is not

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)
```



For...in loop

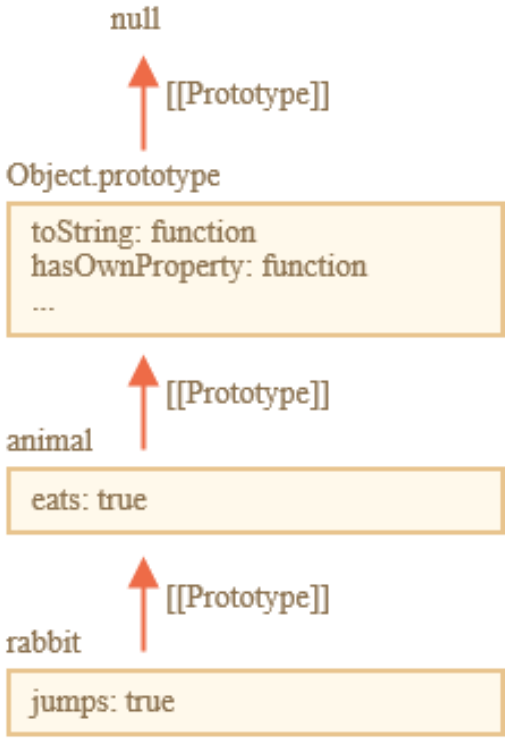
- `for..in` loops over inherited properties too.

```
let animal = {
    eats: true
};
let rabbit = Object.create(animal);
rabbit.jumps = true;

// Object.keys only return 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
```

built-in method `obj.hasOwnProperty(key)`

- it returns true if `obj` has its own property named key
  - can filter out inherited properties

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

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

for (let prop in rabbit) {
    let isOwn = rabbit.hasOwnProperty(prop);
    if (isOwn) {
        alert(`Our: ${prop}`);  // Our: jumps
    } else {
        alert(`Inherited: ${prop}`);  // Inherited: eats
    }
}
```



In-class Exercises

1. Use Object.create() to assign prototypes in a way that any property lookup will follow the path: pockets → bed → table → head. For instance, pockets.pen should be 3 (found in table), and bed.glasses should be 1 (found in head).
2. Answer the question: is it faster to get glasses as pockets.glasses or head.glasses ? Benchmark if needed.

```
let head = {
    glasses: 1
};

let table = {
    pen: 3
};

let bed = {
    sheet: 1,
    pillow: 2
};

let pockets = {
    money: 2000
};
```

Main Point: Prototypal inheritance and Object.create

Prototypal inheritance allows object to inherit properties from a ‘prototype’ parent object. The main purpose of inheritance is to promote code reuse and avoid duplication. Science of Consciousness: Reuse of code for common tasks is efficient and avoids errors that can arise from inconsistent updates of duplicated code. Natural law takes the path of least action. Do less and accomplish more.

Programmers cannot directly access the special [[Prototype]] property. All functions have a regular ‘prototype’ property. When they are called as constructors with ‘new’ that property will be set as the value of [[Prototype]]. [[Prototype]] can also be set with the __proto__ property, but that is now deprecated in favor of Object.create. Science of Consciousness: JavaScript’s prototype is like “archetype”, which is an original object that is a basis for other objects. Deeper levels of thought are connected to archetypal patterns of intelligence or ‘laws of nature’.

Main Point Preview: **Constructor, operator "new"**

Constructor functions are helpful when we need to create many similar objects. They are also used in establishing prototype relations and underly JavaScript classes.

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.

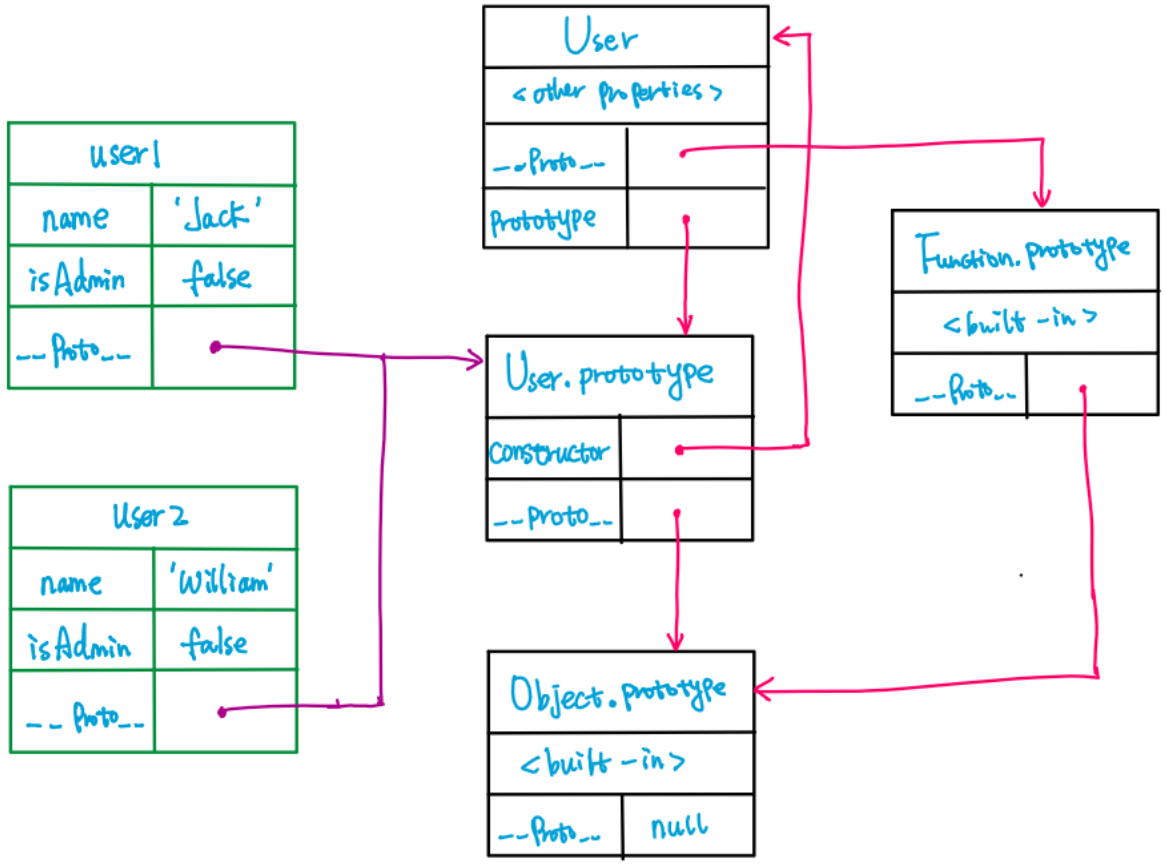
➢In other words, `new User(...)` does something like:

```
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");
```



Default `F.prototype` constructor property

- Every function has "`prototype`" property by default
  - object with property '`constructor`' that points back to function

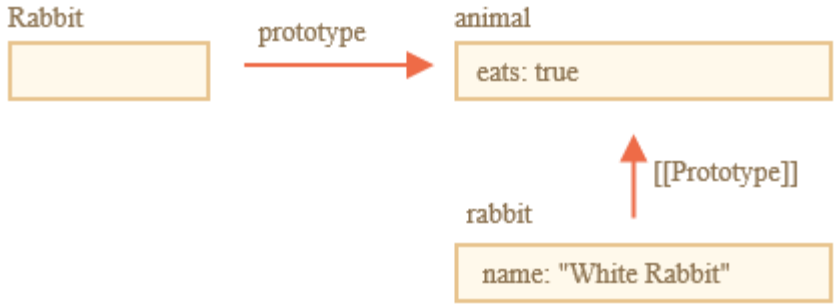
```
function Rabbit() { }
/* default prototype */
Rabbit.prototype = { constructor: Rabbit };
```

- handy if don't know constructor was for an object
  - need to create another one of the same kind.

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

- Can lose constructor link if set prototype property

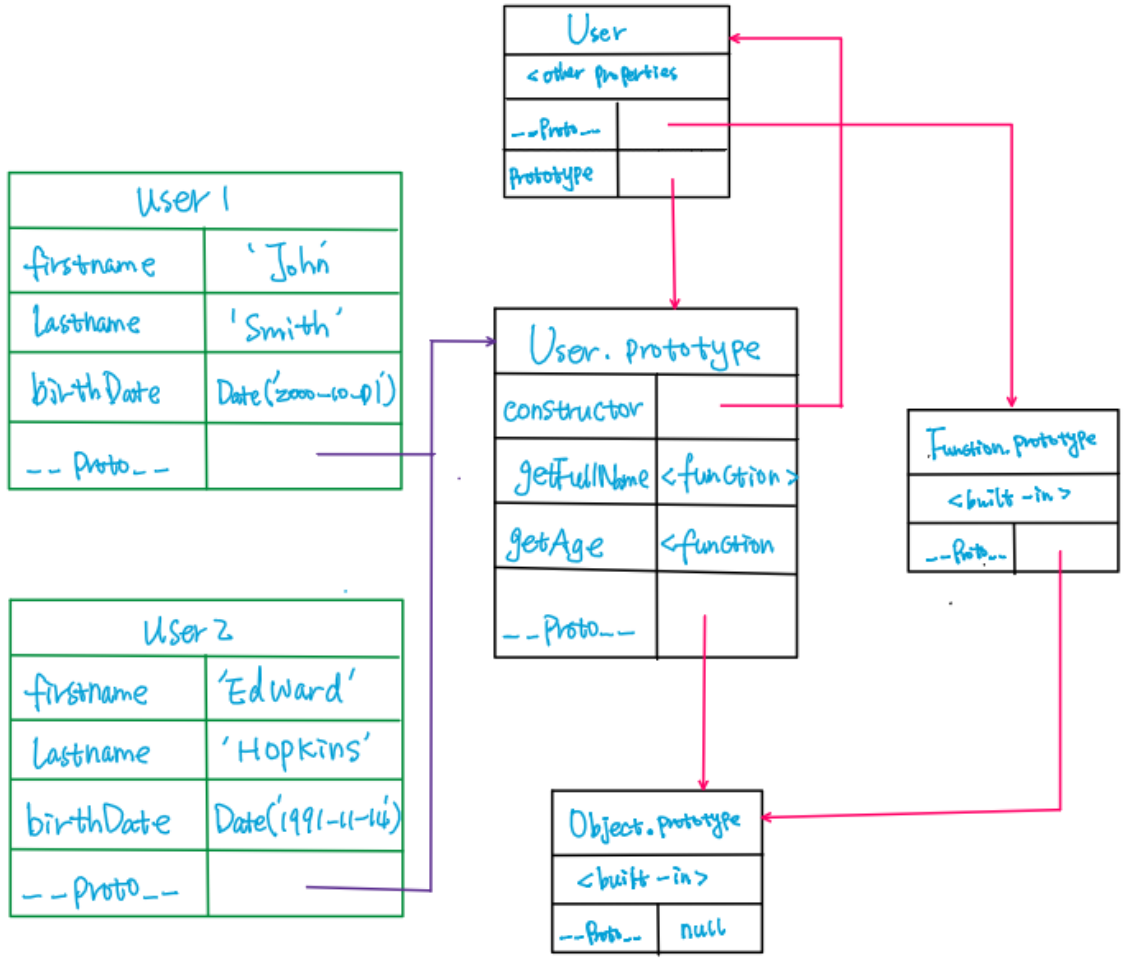
```
function Rabbit() { }
Rabbit.prototype = {
    jumps: true
};
let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```



Extend functionality using `F.prototype` property

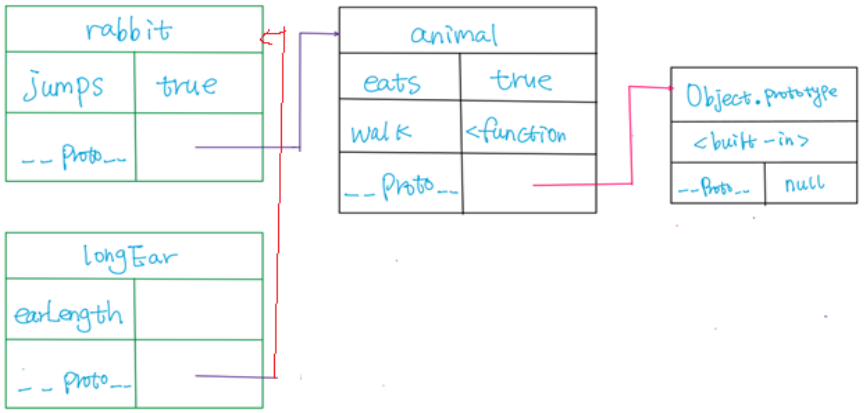
- add/remove properties to default '`prototype`' property

```
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
```

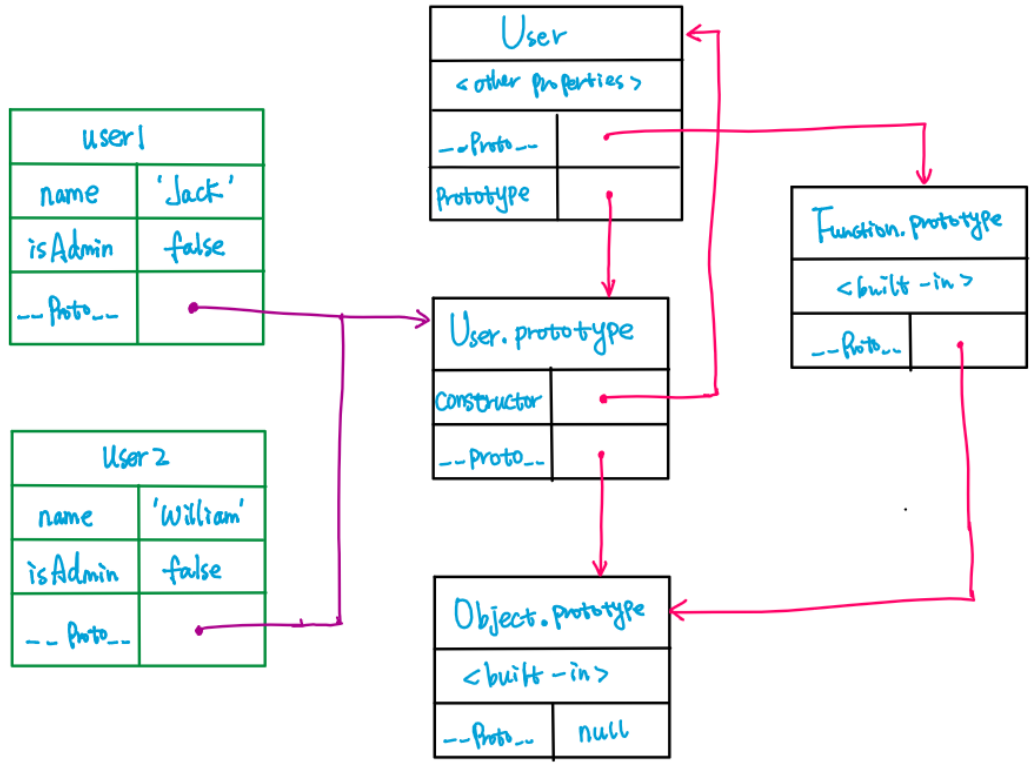


Constructor Function vs object literal

```
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();
```




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



Constructor Function vs object literal

In previous slide:

- `rabbit` and `longEar` `[[Prototype]]/__proto__` properties point to
  - `animal` for `Object.create`
- `user1` and `user2` `[[Prototype]]/__proto__` properties point to
  - `User`.prototype for constructor function, using `new` keyword

- Extensions are made by adding new properties to
  - the prototype object with Object.create
    - added in object `animal`
  - FunctionConstructor.prototype with constructor function
    - Added in `User`.prototype

- new ContructorFunction() is Object.create(ContructorFunction.prototype) plus run constructor function

In-class Exercises

- Create a constructor function `Calculator` which accepts two `Number` type parameters.
- The `Calculator` constructor function has the following methods:
  - `add`: sum the passed-in two arguments
  - `substract`: substract the passed-in two arguments
  - `multiply`: multiply the passed-in two arguments
  - `divide`: divide the passed-in two arguments
- Create an object using `Calculator`, then make call to those methods.

Main Point: **Constructor, operator "new"**

Constructor functions are helpful when we need to create many similar objects. They are also used in establishing prototype relations and underly JavaScript classes.

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);
```

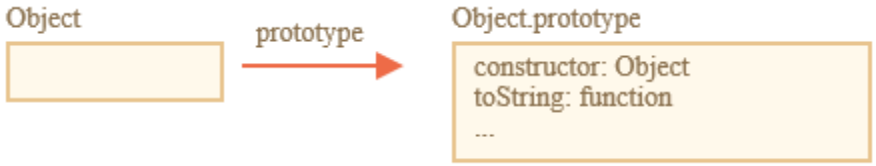
  - for adding new capabilities to built-in objects.
    - Define your own filter, map, etc functions in Array

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

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

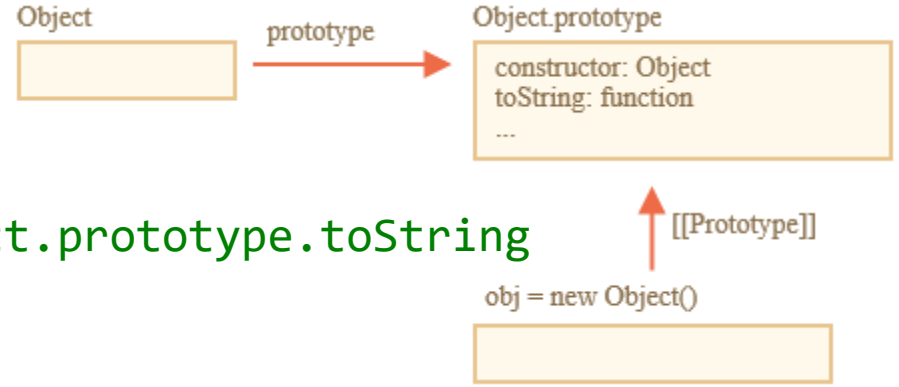
Object.prototype

- `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.



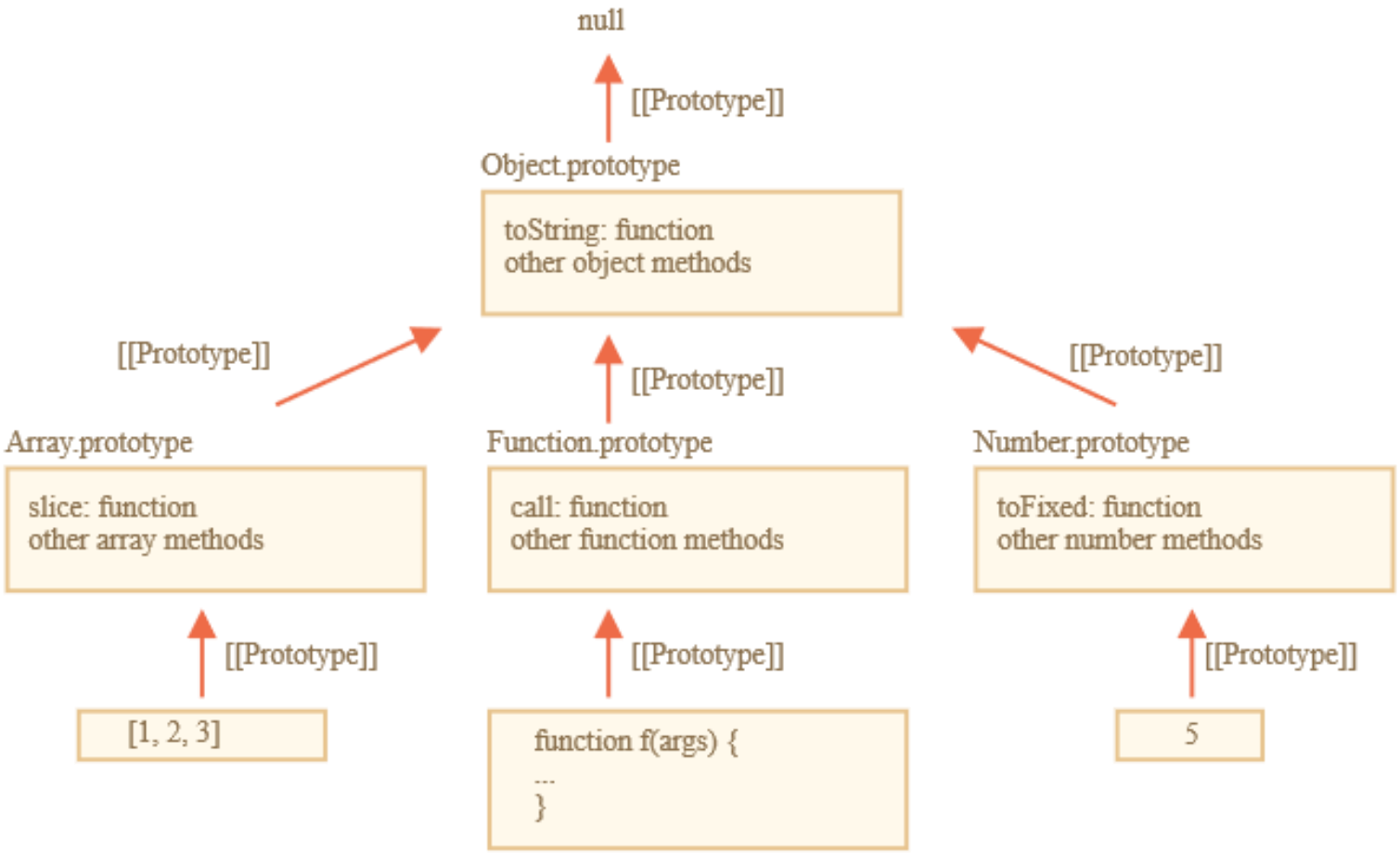
- When `new Object()` is called (or create object literal `{...}`)
  - `[[Prototype]]` of it is set to `Object.prototype`

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

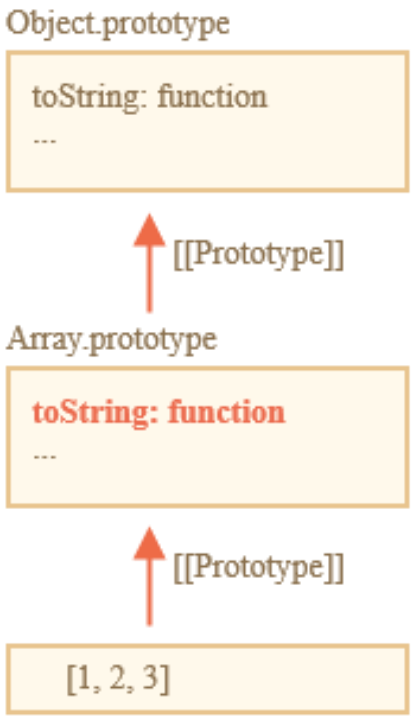



```
let obj = {};
alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString === Object.prototype.toString
```

Other built-in prototypes



JS object hierarchy




```
> console.dir([1,2,3])
▼ Array[3] i
    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] }
      ▶ ...
```

Changing native prototypes

- Native prototypes can be modified.
  - add a method to String.prototype, it becomes available 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.
  - generally a bad idea, easy to get a conflict
  - Native objects and their prototypes are global to all applications
  - If two libraries add a method String.prototype.show, one will overwrite the other

Borrowing from prototypes

- Some methods of native prototypes are often borrowed
  - 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!
```

- works, because join only cares about correct indexes and length property,
  - doesn't check that the object is indeed the array
  - many built-in methods are like that.
- Another possibility is to inherit by setting `obj.__proto__` to `Array.prototype`
  - all Array methods become available in `obj`

History of [[Prototype]], __proto__, prototype

- "`prototype`" property of a constructor function works since ancient times

- 2012: `Object.create` appeared in the standard
  - create objects with the given prototype, but did not allow to get/set it.
  - browsers implemented non-standard `__proto__` accessor
    - allowed to `get/set` a prototype at any time.

- 2015: `Object.setPrototypeOf` and `Object.getPrototypeOf` added to standard
  - same functionality as `__proto__`
  - `__proto__` was de-facto implemented everywhere
    - "kind-of deprecated" and made its way to the Annex B of the standard,
    - optional for non-browser environments

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.