# 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

# Main Point Preview: Prototypal inheritance

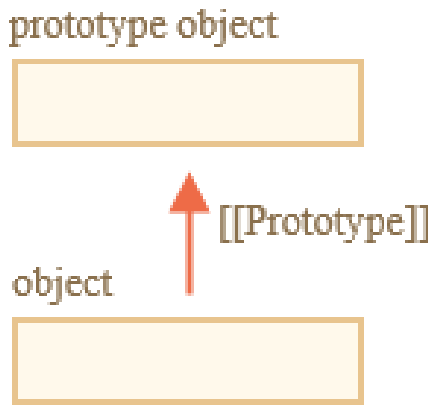
Prototypal inheritance allows objects to inherit properties from a ‘prototype’ parent object. The main purpose of inheritance is to promote code reuse and avoid duplication. Science of Consciousness: Reuse of code for common tasks is efficient and avoids errors that can arise from inconsistent updates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

# 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]]



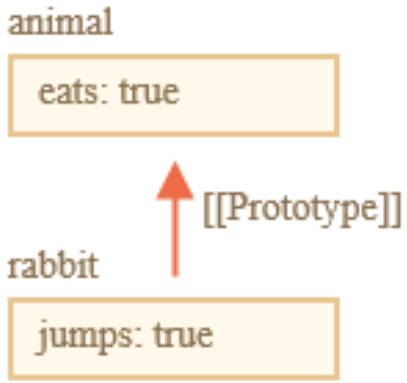

- every object has special hidden property [[Prototype]]
  - either null or references another object.
  - object is called “a prototype”:

- 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.

# Inherit properties

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

- console.log tries to read property rabbit.eats (**),
  - JavaScript follows the [[Prototype]] reference and finds it in animal

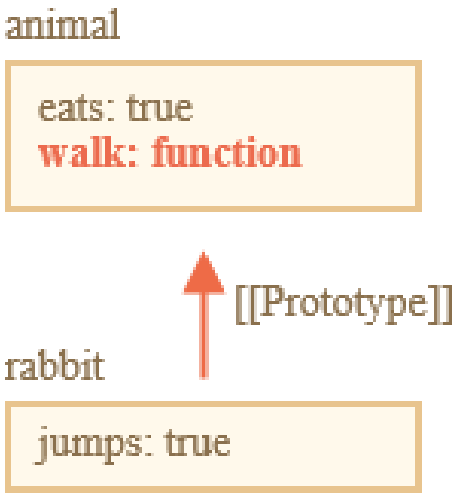



```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal; // (*) __proto__ is a 'sneaky' (deprecated) way to access [[Prototype]]

// 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

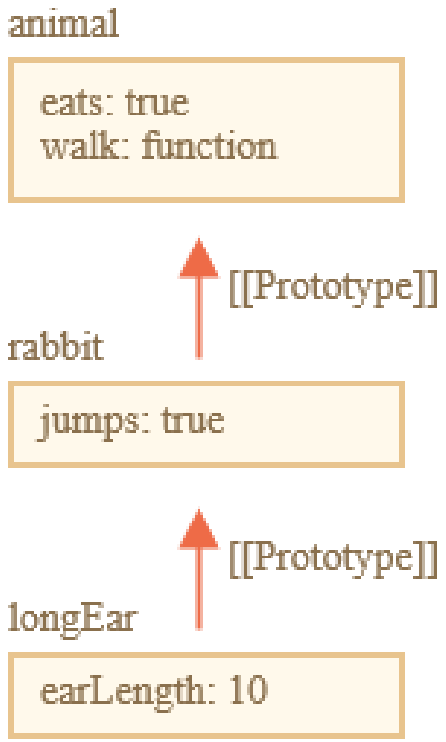
```
let animal = {
  eats: true,
  walk:  function() {
  console.log("Animal walk");
  }
};

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

// 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() {
  console.log("Animal walk");
  }
};
let rabbit = {
  jumps: true,
  __proto__: animal
};
let longEar = {
  earLength: 10,
  __proto__: rabbit
};
```



# 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 = {
  __proto__: animal
};

rabbit.walk = function() {
  console.log("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

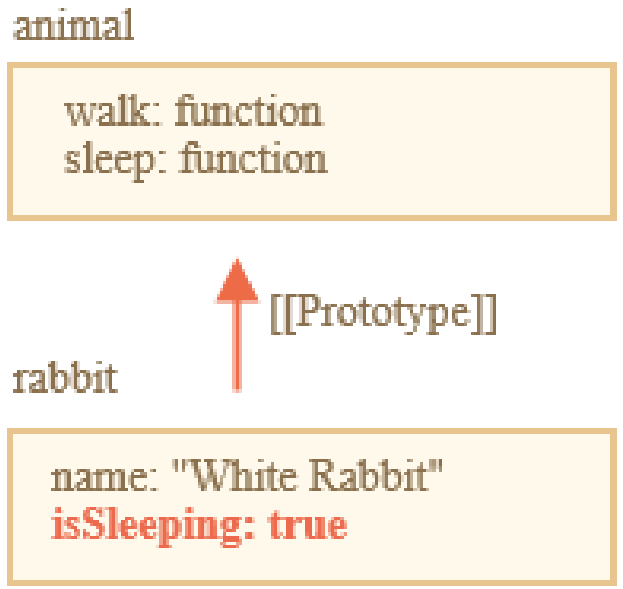
- a super-important thing,
  - may have a big object with many methods and inherit from it.
  - descendent objects can run its methods, and they will modify their own state

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

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

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



# Exercise

1. Use __proto__ so any property lookup will follow the
path: pockets → bed → table → head.

pockets.pen should be 3

bed.glasses should be 1

2. Draw the object diagram with objects and labeled arrows for the [[Prototype]] links

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

let table = {
    pen: 3
};

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

let pockets = {
    money: 2000
};

console.log("expect 3: ", pockets.pen);
console.log("expect 1: ", bed.glasses);
```

# For…in loop

- for..in loops over inherited properties too.

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

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

// Object.keys only return own keys
console.log(Object.keys(rabbit)); // jumps

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

# Main Point: Prototypal inheritance

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.

# Main Point Preview:  Setting prototypes with constructors

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]].
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’.

# Constructor functions, operator “new” (review)

- 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: (review)

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.

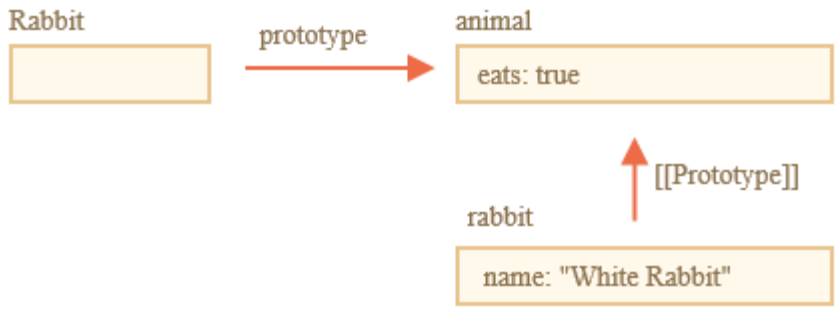
➢In other words, new User(...) does something like:

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

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

  // return this;  (implicitly)}
```

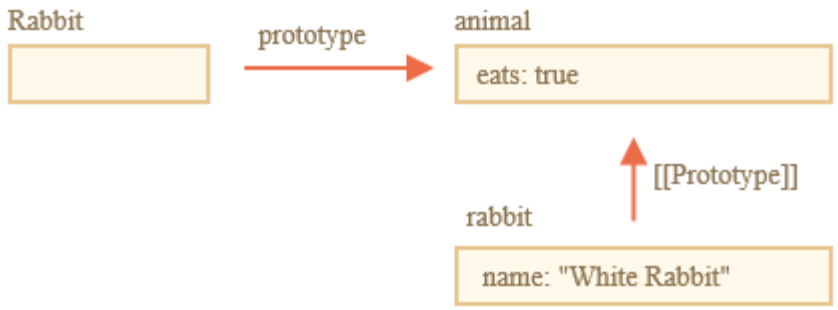
# F.prototype -- Set [[Prototype]] using constructor function

- If MyConstructor.prototype is an object,
  - new operator sets it to [[Prototype]] for the new object.
- MyConstructor.prototype is a regular property named "prototype"
  - This is not the ‘special hidden’ [[Prototype]] property
  - regular property with this name
- When ‘new’ is called sets [[Prototype]]  to MyConstructor.prototype

```
let animal = {
  eats: true
};
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype = animal;
let rabbit = new Rabbit("White Rabbit"); //rabbit.__proto__ == animal
 console.log( rabbit.eats ); // true
```



# Exercise




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

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

Rabbit.prototype = animal;

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

console.log( rabbit.eats ); // true

//rewrite this to have the same inheritance hierarchy using __proto__ instead of the constructor .  I.e., delete the Rabbit prototype and use only object literals and __proto__
```

# Exercise

```
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'));

function getFullName() { return this.firstname + ' ' + this.lastname;}
function getAge() {return new Date().getFullYear() - this.birthDate.getFullYear();}



//complete the code so that the above functions reside in a single object and are inherited by all User
//objects that are created using User as a constructor function.
console.log(user1.getFullName()); //John Smith
console.log(user1.getAge()); //21
```

# Exercise

draw the object diagram for the User constructor exercise

# Homework

- Changing “prototype”
  - exercise involving the F.prototype property
  - Drawing the object diagram will help clarify the answer

# Native prototypes

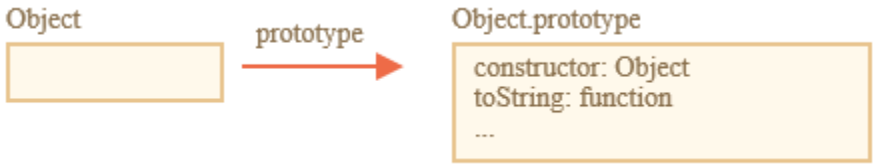
- "prototype" property is widely used by core of JavaScript
  - All built-in constructor functions use
  - for adding new capabilities to built-in objects.

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

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

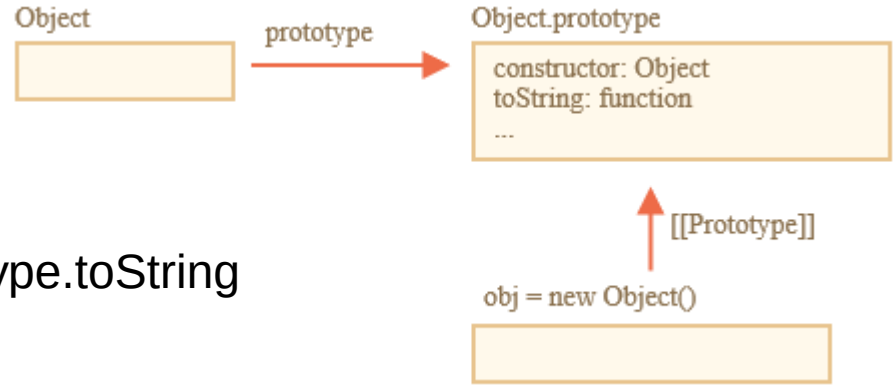
# Object.prototype

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

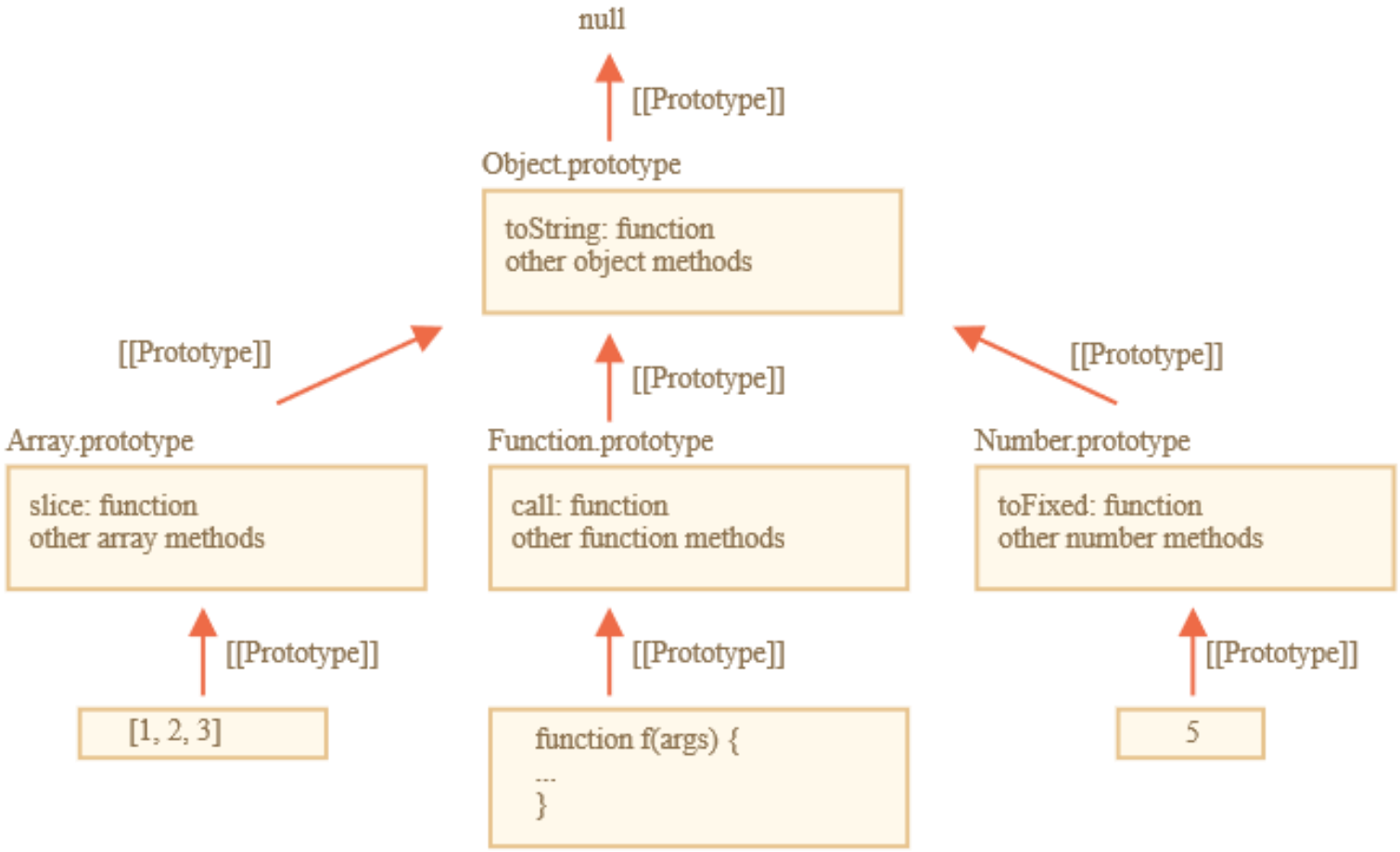



- When new Object() is called (or create object literal {...} )
  - [[Prototype]] of it is set to Object.prototype
  - obj.toString() is inherited from Object.prototype.

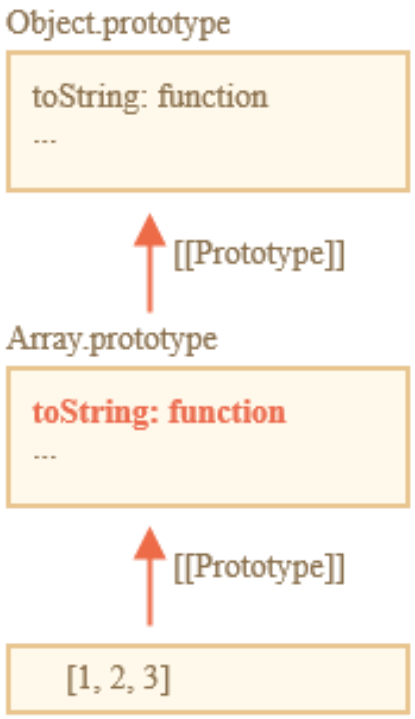
```
let obj = {};
alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString === Object.prototype.toString
```



# 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() {console.log(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

# Main Point: Setting prototypes with constructors

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]]. 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’.

# 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 (or class syntax).

---

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.