PROTOTYPE INHERITANCE

Archetypal Patterns of Intelligence

Slides based on material from https://javascript.info licensed as CC BY-NC-SA.
As per the CC BY-NC-SA licensing terms, these slides are under the same license.

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

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.

Code duplication issue

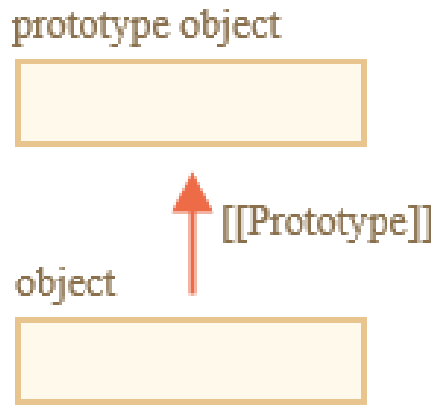
- If make multiple instances of a module, then duplicate all the module code every time create a new module instance
- The makeCounter function is an object factory
  - – allows reuse of the module pattern functionality
- Problem: Why is an object factory inefficient when the methods become nontrivial?

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

- inheritance is one important language feature that avoids duplicating common code
- languages with built-in classes use class based inheritance
- JavaScript has objects, but no classes in the sense of Java
- uses a different type of inheritance – “prototypal” inheritance

[[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.

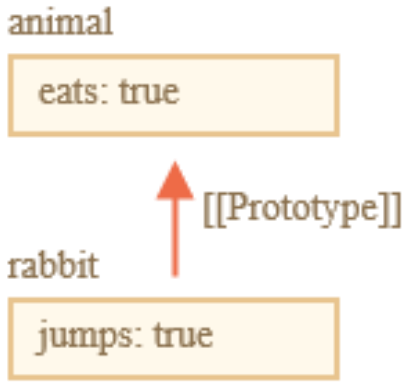
```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal;  // __proto__ is a ‘sneaky’ (deprecated) way to access [[Prototype]]
```

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

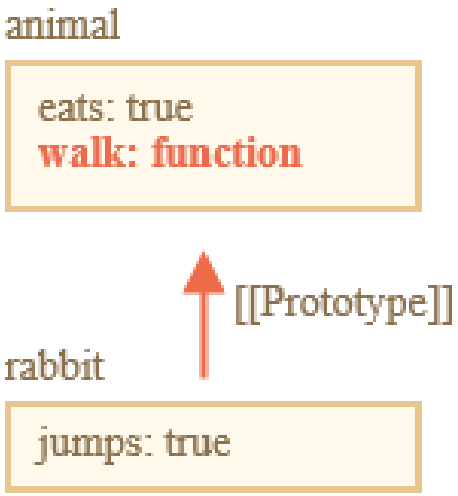
```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal; // (*)

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



Inherit methods

- method in animal, it can be called on rabbit

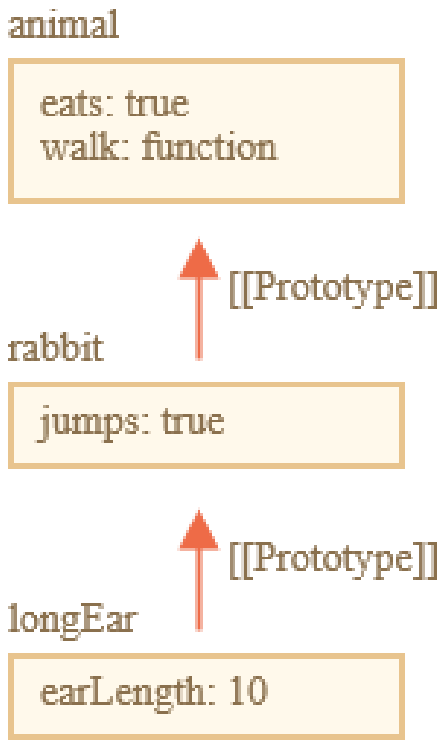
```
let animal = {
  eats: true,
  walk:  function() {
    alert("Animal walk");
  }
};

let rabbit = {
  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() {
    alert("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” 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() {
  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”

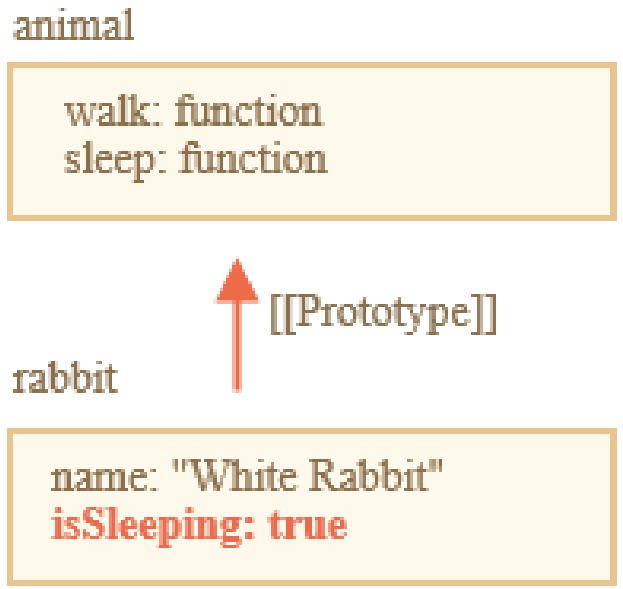
- what’s the value of this inside an inherited method
  - answer: this is not affected by prototypes at all.
  - No matter where the method is found:
    - in an object or its prototype
    - this is always the object before the dot (“object before dot rule”)

- 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)
```



For…in loop

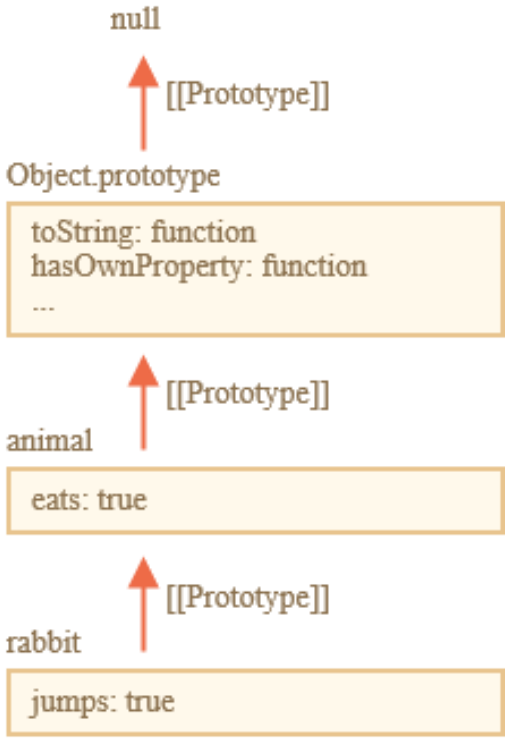
- for..in loops over inherited properties too.

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

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

// Object.keys only return own keys
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 = {
  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
  }}
```



Exercises

- Working with prototype
- Searching algorithm
- Where it writes
- Why two hamsters are full

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: **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)}
```

Constructor vs object literal

- Result of new User("Jack") is same as

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

- if we want to create other users, can call new User("Ann"), new User("Alice") etc
  - shorter than using literals every time
  - easy to read
  - For CS303 favor object literals
  - Constructors will become important with inheritance and classes

- Exercises
  - Two functions – one object
  - Create new Calculator
  - Create new Accumulator

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.

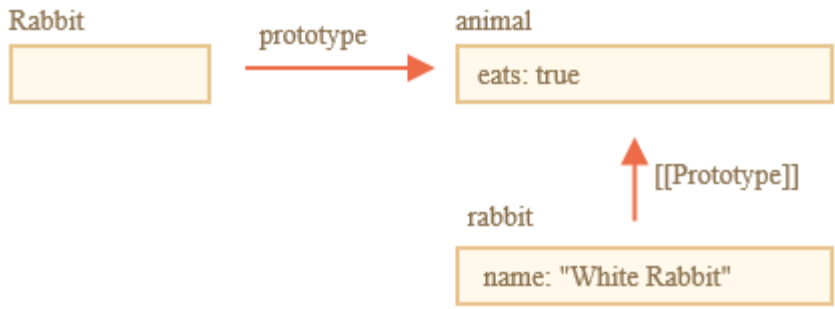
Main Point Preview: Setting prototypes with constructors and Object.create

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’.

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

- recall, new objects can be created with constructor function, like new F().
- If F.prototype is an object,
  - new operator uses it to set [[Prototype]] for the new object.
- F.prototype is a regular property named "prototype" on F.
  - This is not the ‘special hidden’ [[Prototype]] property
  - regular property with this name
- When ‘new’ is called takes value of F.prototype and sets as value of [[Prototype]] property

```
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
```



Create objects via Function Constructors

```
function Person() {
 console.log(this);
 this.university = 'MUM';
 this.year = '2016';
}

const faculty1 = new Person();

Person.prototype.greet = function() {
 return 'Hi ' + this.university;
}

const greeting = faculty1.greet();
console.log(greeting); // "Hi MUM"
```

- Can create thousands of objects from original function constructor with less memory space.
- Can extend the functionality of all objects at runtime by adding methods and properties to the **prototype property.**
  - ***distinct from __proto__ which is used by the JSengine***

Create course objects via function constructors

```
// By convention we use capital first letter for function constructor
function Course(coursename) {
 this.coursename = coursename;
 console.log('Function Constructor Invoked!');
  //implicit return of 'this' when called via 'new'
}

//add function register to prototype of all course objects (created from //Course
  constructor)
Course.prototype.register = function() {
 return 'Register ' + this.coursename;
}

const wap = new Course('WAP'); // Function Constructor Invoked!
console.log(wap); // Course {coursename: "WAP"}
console.log(wap.__proto__); // Course.prototype
console.log(wap instanceof Course); // true
console.log(Course.prototype.register); // function(){ ... }
console.log(wap.register()); // Register WAP
```

Default F.prototype constructor property

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

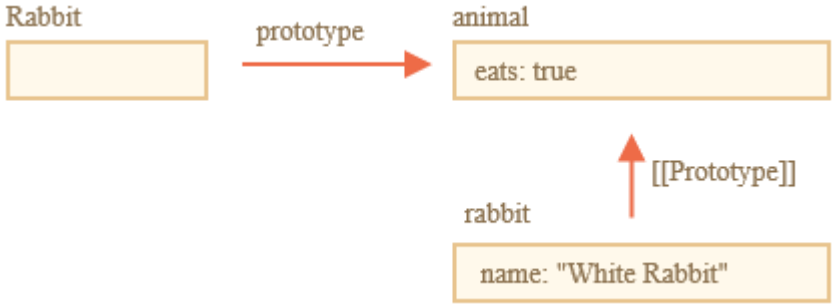
```
function Rabbit() {}
/* default prototype */
Rabbit.prototype = { constructor: Rabbit };
```

- handy if don’t know constructor was for an object
  - (e.g. it comes from a 3rd party library),
  - 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
```



Preserve constructor link in prototype

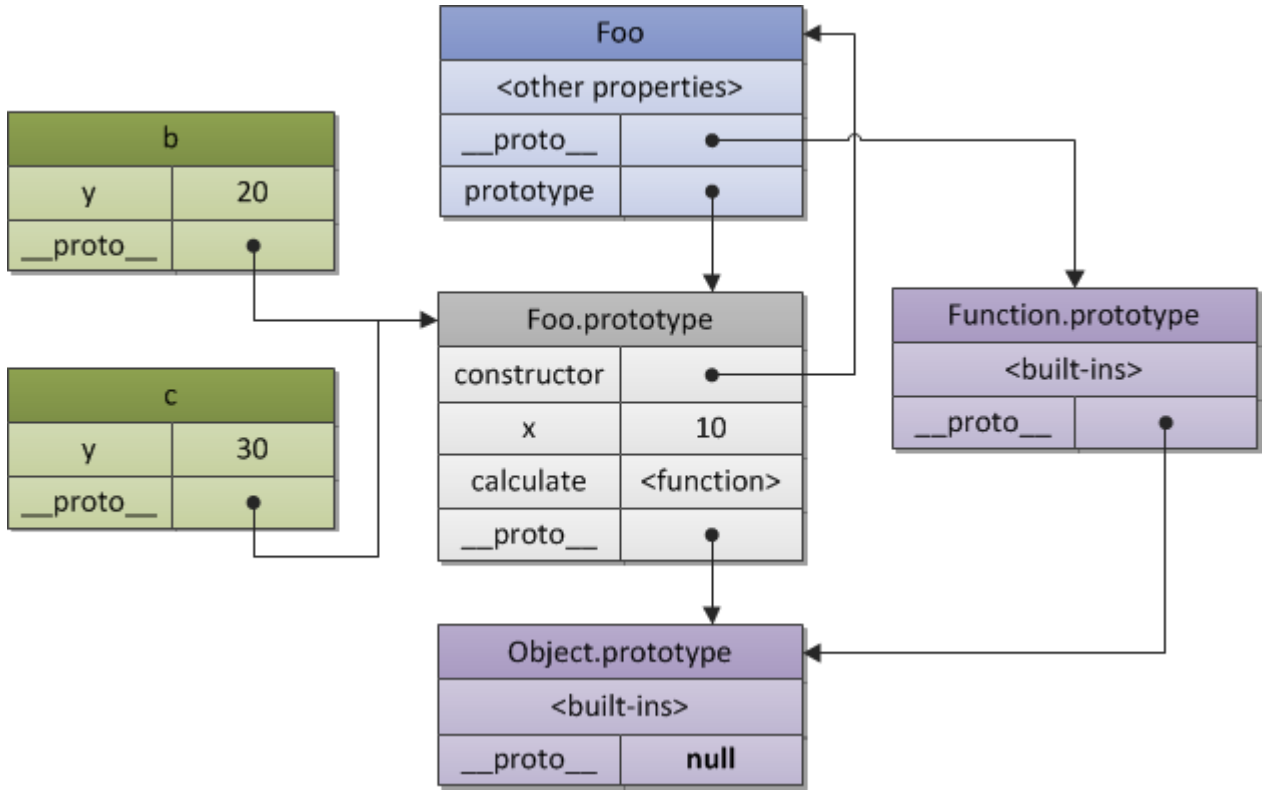
- to retain constructor link when set prototype object via new need to
  - add/remove properties to default ‘prototype’ property
  - Or, recreate the constructor manually

```
function Rabbit() {}
// Do not overwrite Rabbit.prototype totally,  just add to it
Rabbit.prototype.jumps = true


Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};
// now constructor is also correct, because we added new properties to existing one
```

- Exercises: Changing “prototype”, Create an object with the same constructor

Constructor function prototype diagram

```
// a constructor function
function Foo(x) {
  this.y = x;
}


/*
__proto__ property of new objects point to prototype object of the constructor function so we may use it to define

shared/inherited properties or methods  "x“ and "calculate“
*/
Foo.prototype.x = 10;
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};


// now create our "b" and "c" objects using "pattern" Foo
var b = new  Foo(20);
var c = new  Foo(30);
);
```



//Ref: http://dmitrysoshnikov.com/ecmascript/javascript-the-core/

Exercises

- Changing “prototype”
  - exercise involving the F.prototype property

- Create an object with the same constructor
  - exercise involving preserving the .constructor link in the [[Prototype]] object

Native prototypes

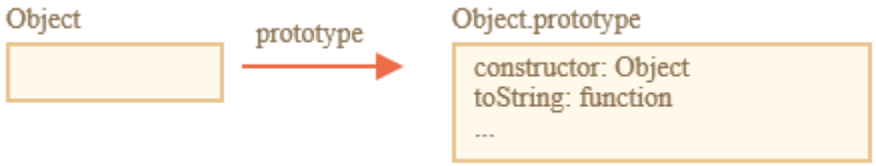
- "prototype" property is widely used by core of JavaScript
  - All built-in constructor functions use
  - for adding new capabilities to built-in objects.

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

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

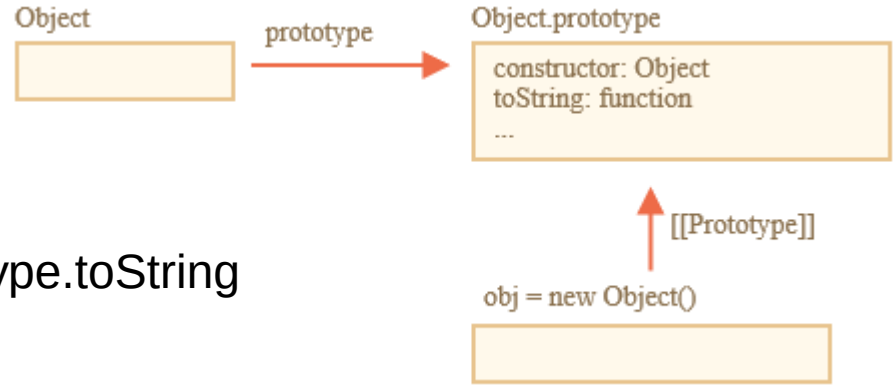
Object.prototype

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

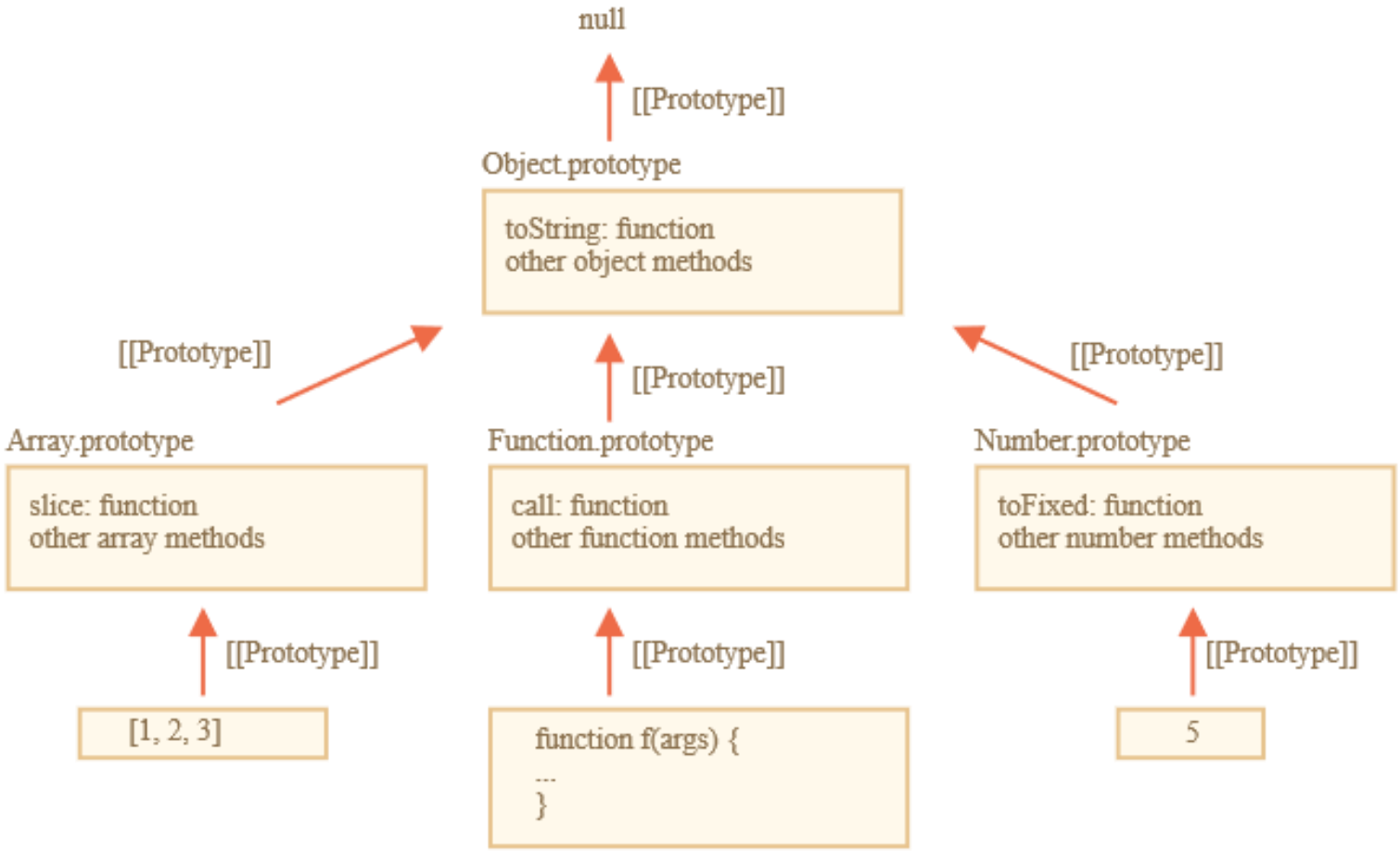


- When new Object() is called (or create object literal {...})
  - [[Prototype]] of it is set to Object.prototype
  - obj.toString() is inherited from Object.prototype.

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



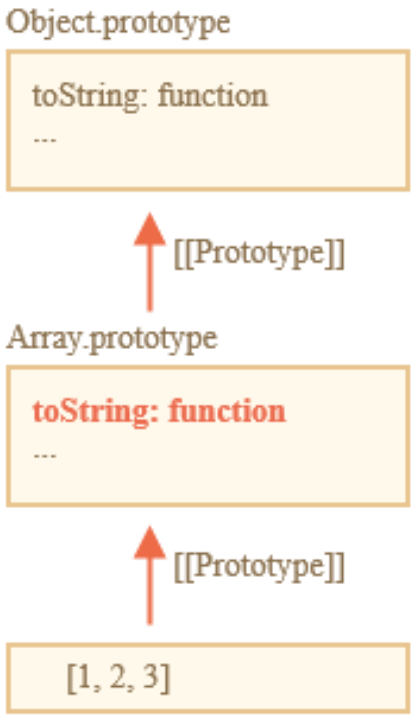
Other built-in prototypes



Built-in Function Constructors

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

/* Number.prototype, String.prototype, Date.prototype
 are objects with helper methods
 available because objects were created using new()
 keyword */

a.toString(); // "12"
b.italics(); // "<i>Hello</i>"
c.getMonth(); // 3
```

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

Exercises

- Add method f.defer(ms) to functions
- Add the decorating “defer()” to functions

Remember the rules for ‘this’ and remember that functions are objects.

Object.create versus __proto__

- __proto__ is considered outdated and “sort of” deprecated

- Object.create(proto) sets [[Prototype]] 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
```

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

Exercises

- The difference between calls

Main Point: Setting prototypes with constructors and Object.create

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’.

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.