JavaScript TUTORIAL

JavaScript Prototype

Summary: in this tutorial, you'll learn about the JavaScript prototype and how it works under the hood.

Introduction to JavaScript prototype

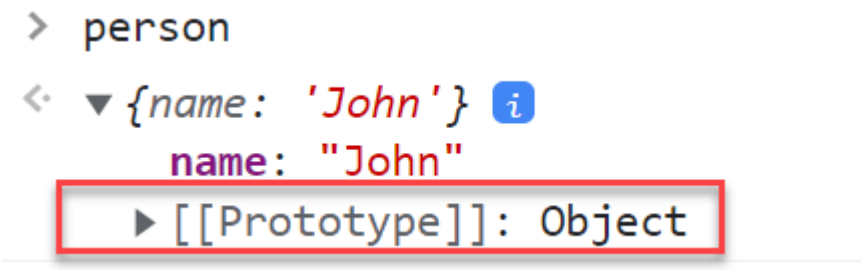
In JavaScript, objects can inherit features from one another via **prototypes**. Every object has its own property called a `prototype` .

Because the `prototype` itself is also another object, the `prototype` has its own `prototype` . This creates a something called **prototype chain**. The prototype chain ends when a prototype has `null` for its own prototype.

Suppose you have an object `person` with a property called `name` :

```
let person = {'name' : 'John'}
```

When examining the `person` object in the console, you'll find that the `person` object has a property called `prototype` denoted by the `[[Prototype]]` :



The prototype itself is an object with its own properties:

When you access a property of an object, if the object has that property, it'll return the property value. The following example accesses the `name` property of the `person` object:

```
> person.name
< 'John'
```

It returns the value of the `name` property as expected.

However, if you access a property that doesn't exist in an object, the JavaScript engine will search in the prototype of the object.

If the JavaScript engine cannot find the property in the object's prototype, it'll search in the prototype's prototype until it finds the property or reaches the end of the prototype chain.

For example, you can call the `toString()` method of the `person` object like this:

```
> person.toString()
< '[object Object]'
```

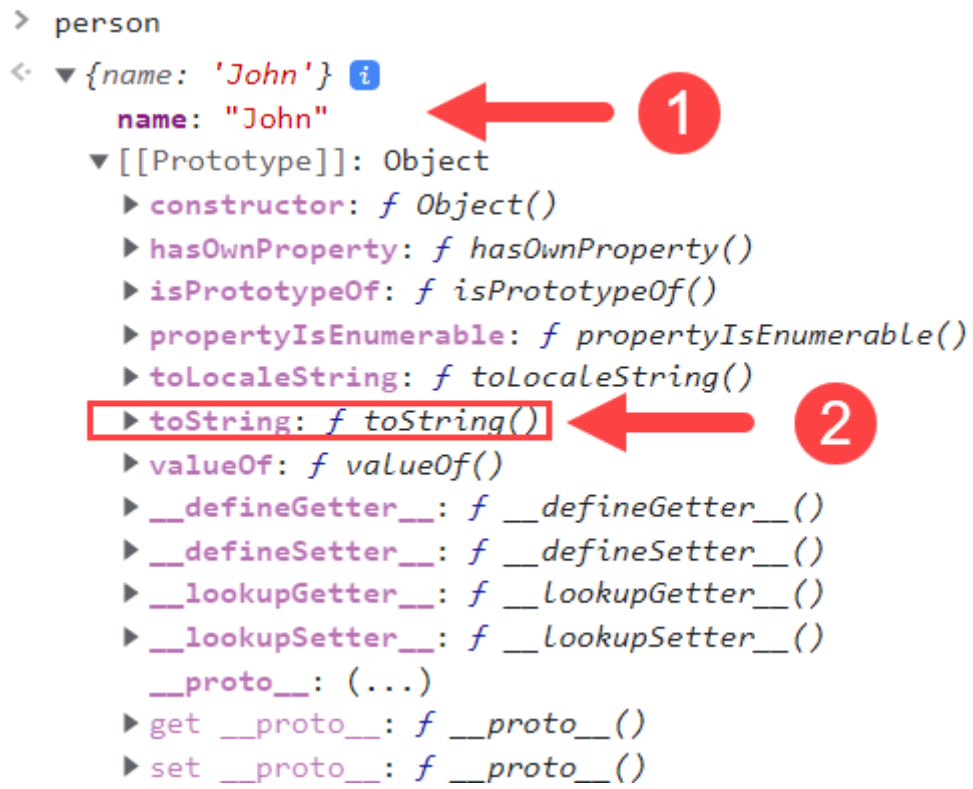
The `toString()` method returns the string representation of the `person` object. By default, it's `[object Object]` which is not obvious.

> Note that when a function is a value of an object's property, it's called a **method**. Therefore, a method is a property with value as a function.

In this example, when we call the `toString()` method on the `person` object, the JavaScript engine finds it in the `person` object.

Because the `person` object doesn't have the `toString()` method, it'll search for the `toString()` method in the person's prototype object.

Since the person's prototype has the `toString()` method, JavaScript calls the `toString()` of the person's prototype object.



JavaScript prototype illustration

JavaScript has the built-in `Object()` function. The `typeof` operator returns `'function'` if you pass the `Object` function to it. For example:

```
typeof(Object)
```

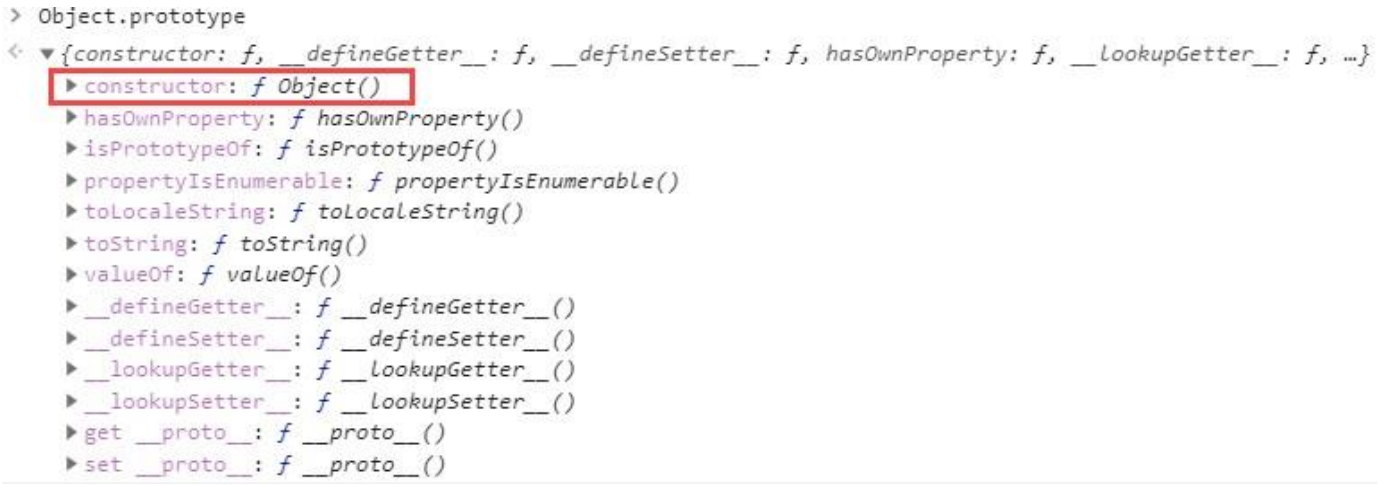
Output:

```
'function'
```

> Please note that `Object()` is a function, not an object. It's confusing if this is the first time you've learned about the JavaScript prototype.

Also, JavaScript provides an anonymous object that can be referenced via the `prototype` property of the `Object()` function:

```
console.log(Object.prototype);
```



The `Object.prototype` object has some useful properties and methods such as `toString()` and `valueOf()` .

The `Object.prototype` also has an important property called `constructor` that references the `Object()` function.

The following statement confirms that the `Object.prototype.constructor` property references the `Object` function:

```
console.log(Object.prototype.constructor === Object); // true
```

Suppose a circle represents a function and a square represents an object. The following picture illustrates the relationship between the `Object()` function and the `Object.prototype` object:

First, define a constructor function called `Person` as follows:

```
function Person(name) {
    this.name = name;
}
```

In this example, the `Person()` function accepts a `name` argument and assigns it to the `name` property of the `this` object.

Behind the scenes, JavaScript creates a new function `Person()` and an anonymous object:

Like the `Object()` function, the `Person()` function has a property called `prototype` that references an anonymous object. The anonymous object has the `constructor` property that references the `Person()` function.

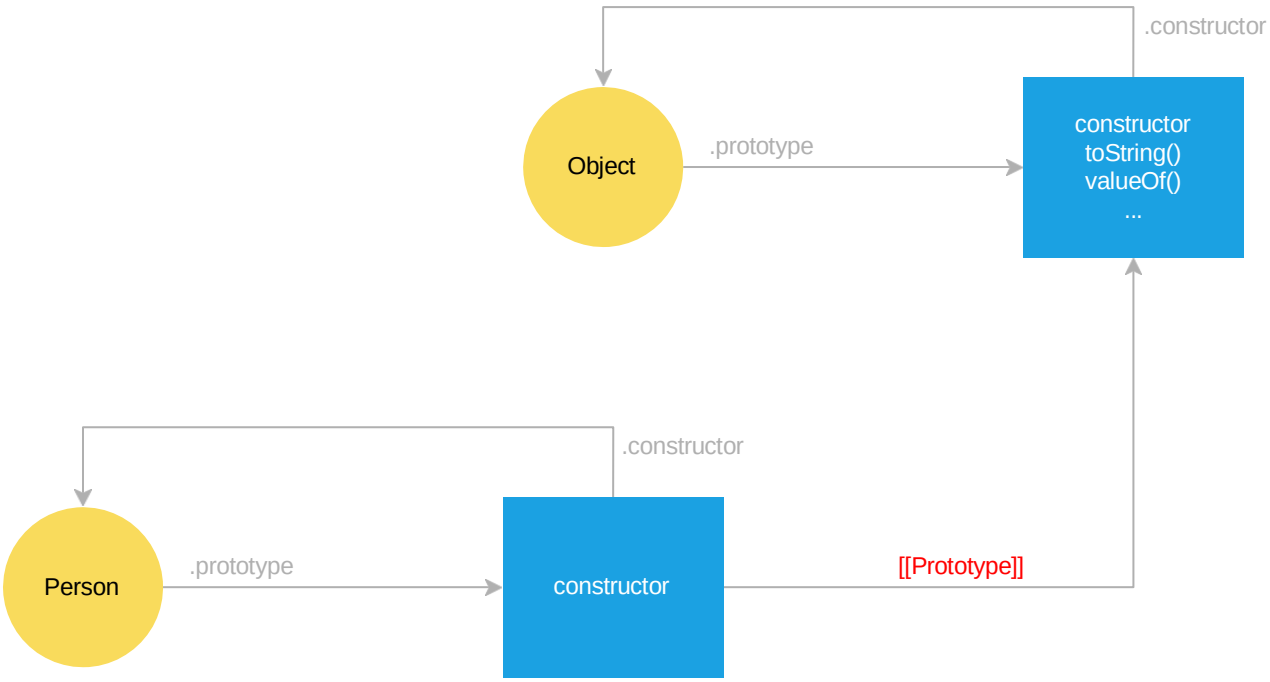
The following shows the `Person()` function and the anonymous object referenced by the `Person.prototype` :

```
console.log(Person);
console.log(Person.prototype);
```

```
> Person
ƒ Person(name) {
    this.name = name;
}
> Person.prototype
▼{constructor: ƒ}
  ▼constructor: ƒ Person(name)
      arguments: null
      caller: null
      length: 1
      name: "Person"
    ▶prototype: {constructor: ƒ}
    ▶__proto__: ƒ ()
      [[FunctionLocation]]: VM582:1
    ▶[[Scopes]]: Scopes[2]
```

In addition, JavaScript links the `Person.prototype` object to the `Object.prototype` object via the `[[Prototype]]`, which is known as a *prototype linkage*.

The prototype linkage is denoted by `[[Prototype]]` in the following figure:

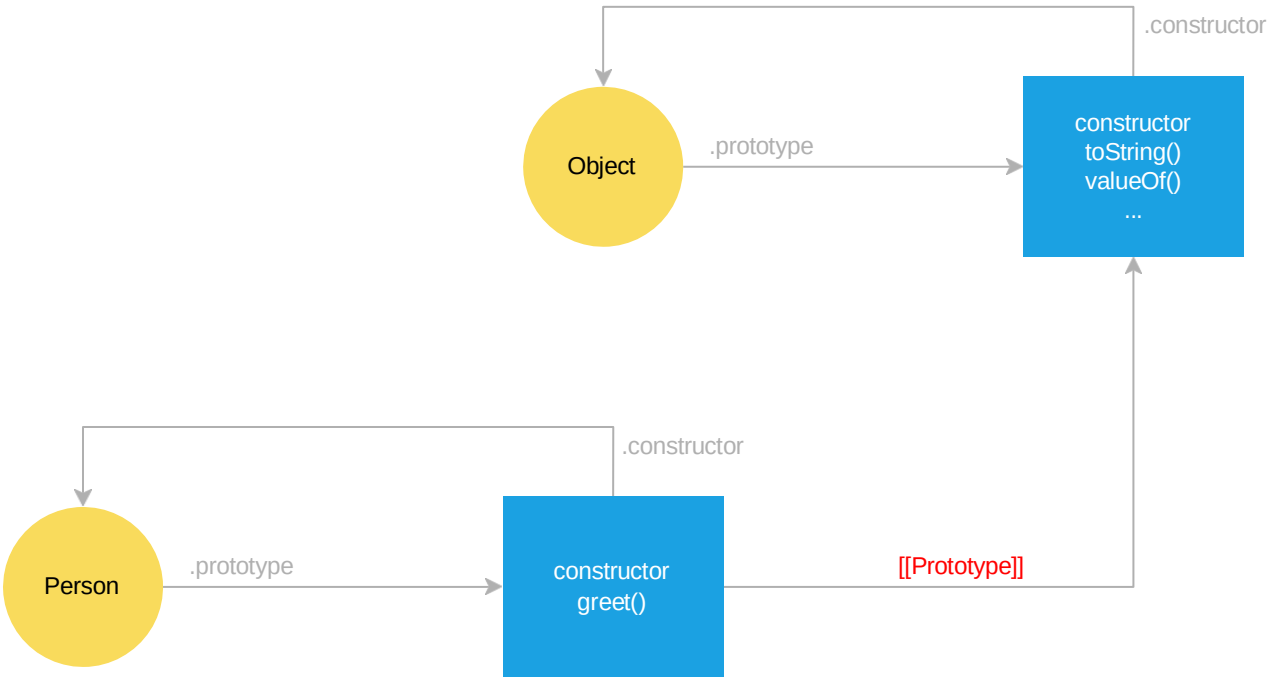


Defining methods in the JavaScript prototype object

The following defines a new method called `greet()` in the `Person.prototype` object:

```
Person.prototype.greet = function() {
    return "Hi, I'm " + this.name + "!";
}
```

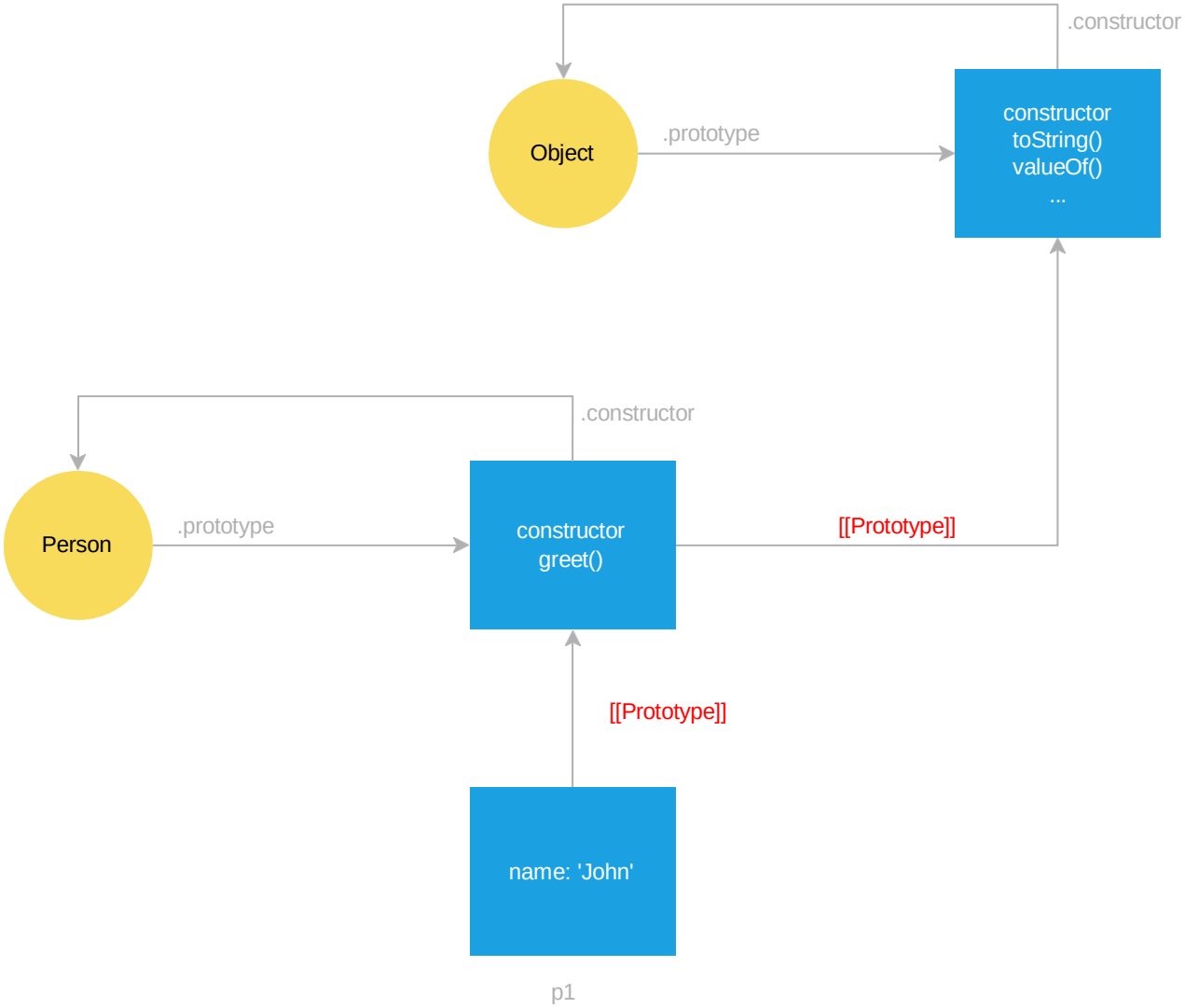
In this case, the JavaScript engine adds the `greet()` method to the `Person.prototype` object:



The following creates a new instance of the `Person` :

```
let p1 = new Person('John');
```

Internally, the JavaScript engine creates a new object named `p1` and links the `p1` object to the `Person.prototype` object via the prototype linkage:



The link between `p1` , `Person.prototype` , and `Object.protoype` is called a *prototype chain*.

The following calls the `greet()` method on the `p1` object:

```
let greeting = p1.greet();
console.log(greeting);
```

Because `p1` doesn't have the `greet()` method, JavaScript follows the prototype linkage and finds it on the `Person.prototype` object.

Since JavaScript can find the `greet()` method on the `Person.prototype` object, it executes the `greet()` method and returns the result:

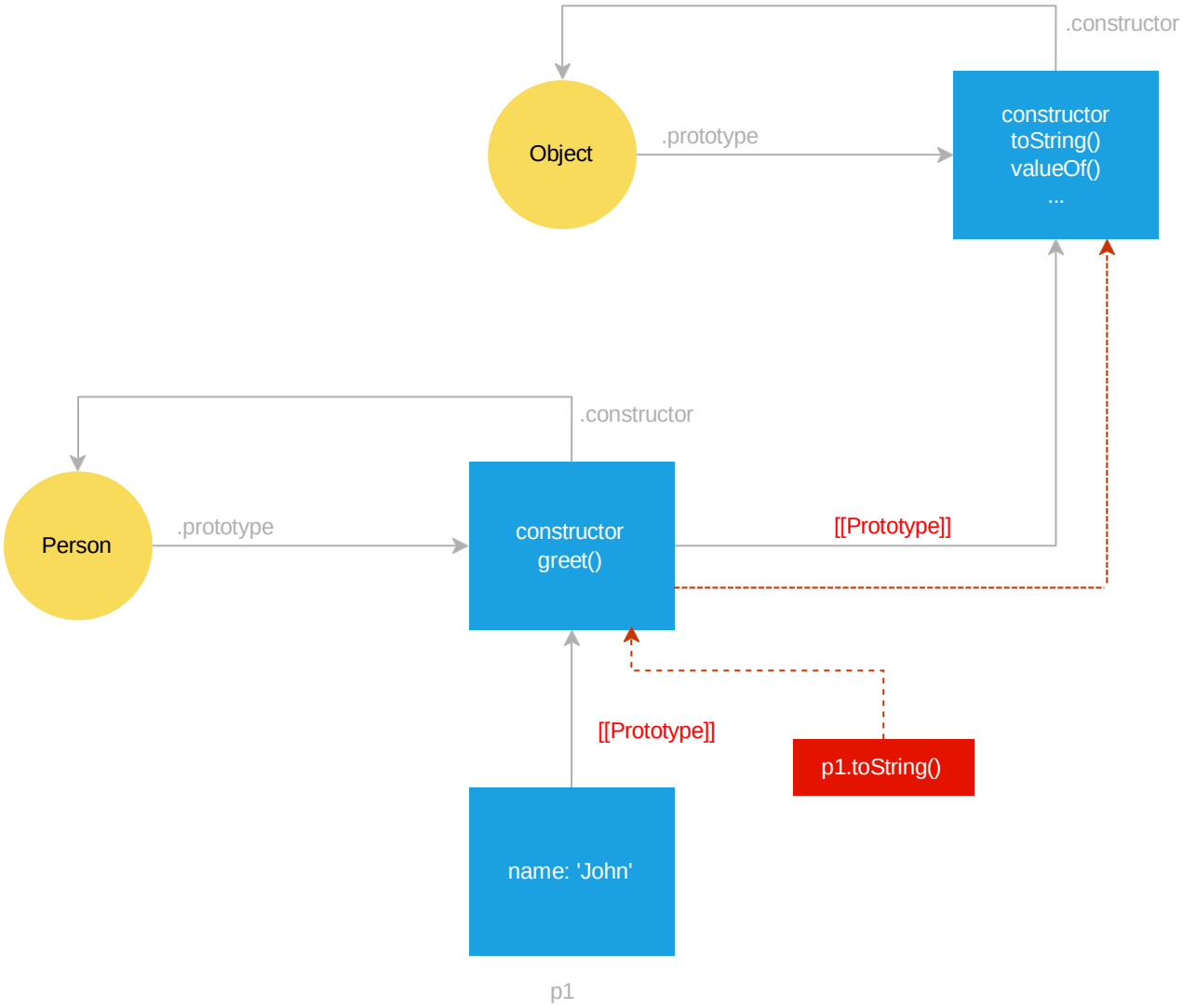
The following calls the `toString()` method on the `p1` object:

```
let s = p1.toString();
console.log(s);
```

In this case, the JavaScript engine follows the prototype chain to look up the `toString()` method in the `Person.prototype` .

Because the `Person.prototype` doesn't have the `toString()` method, the JavaScript engine goes up to the prototype chain and searches for the `toString()` method in the `Object.prototype` object.

Since JavaScript can find the `toString()` method in the `Object.prototype` , it executes the `toString()` method.



If you call a method that doesn't exist on the `Person.prototype` and `Object.prototype` object, the JavaScript engine will follow the prototype chain and throw an error if it cannot find the method. For example:

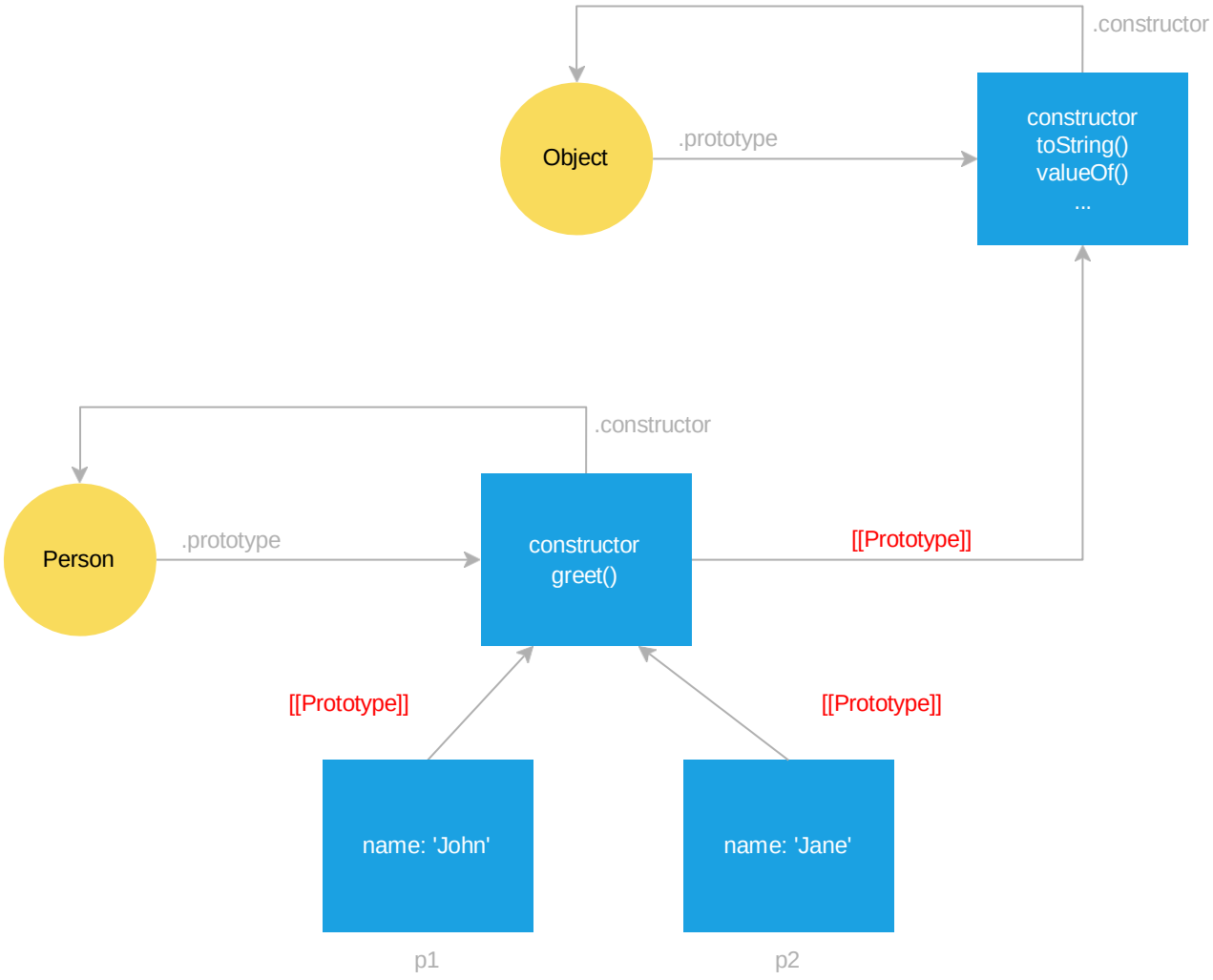
```
p1.fly();
```

Because the `fly()` method doesn't exist on any object in the prototype chain, the JavaScript engine issues the following error:

```
TypeError: p1.fly is not a function
```

The following creates another instance of the `Person` whose name property is `'Jane'` :

```
let p2 = new Person('Jane');
```



The `p2` object has the same properties and methods as the `p1` object.

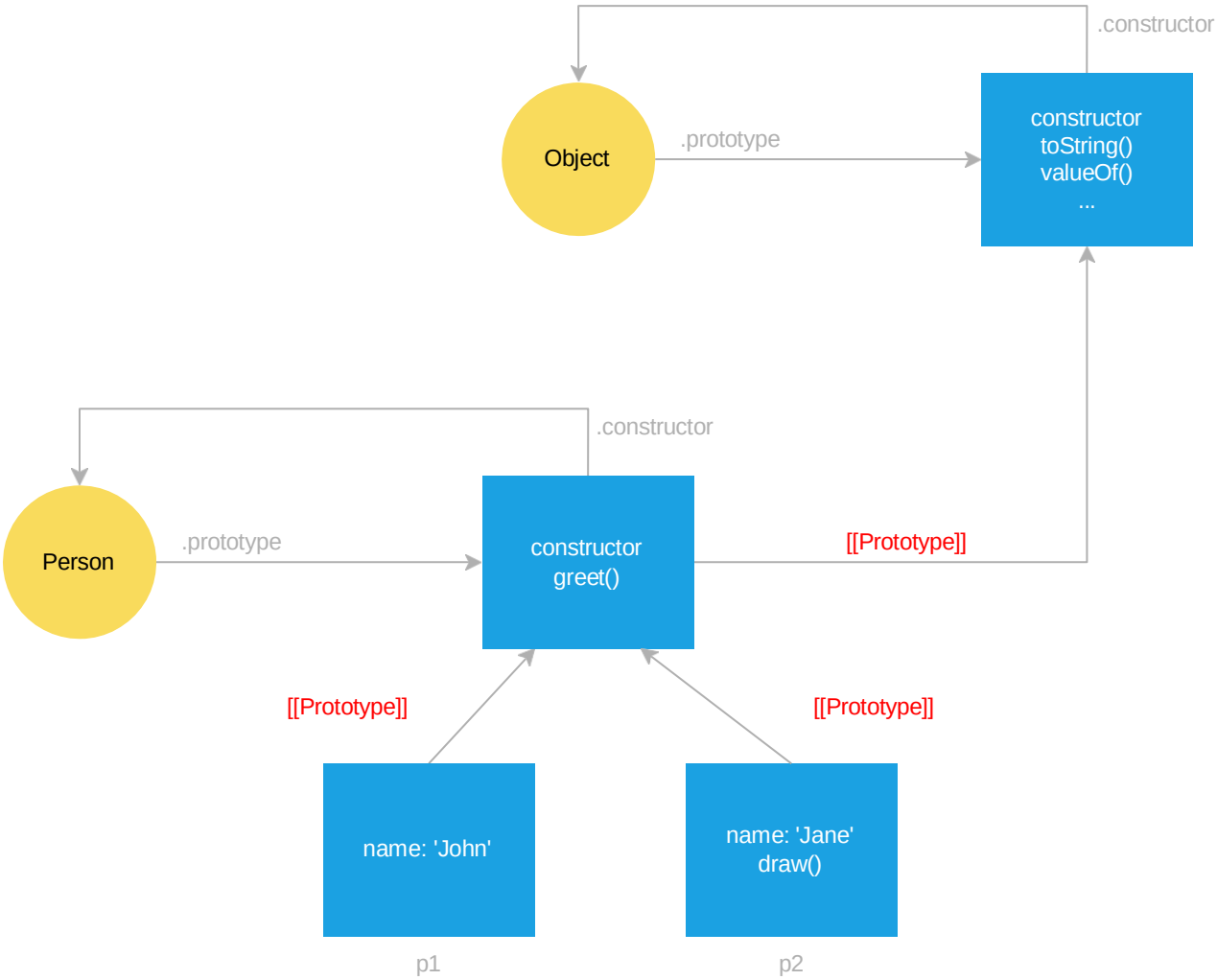
In conclusion, when you define a method on the `prototype` object, this method is shared by all instances.

Defining methods in an individual object

The following defines the `draw()` method on the `p2` object.

```
p2.draw = function () {
    return "I can draw.";
};
```

The JavaScript engine adds the `draw()` method to the `p2` object, not the `Person.prototype` object:



It means that you can call the `draw()` method on the `p2` object:

```
p2.draw();
```

But you cannot call the `draw()` method on the `p1` object:

```
p1.draw()
```

Error:

```
TypeError: p1.draw is not a function
```

When you define a method in an object, the method is only available to that object. It cannot be shared with other objects by default.

Getting prototype linkage

The `__proto__` is pronounced as dunder proto. The `__proto__` is an accessor property of the `Object.prototype` object. It exposes the internal prototype linkage (`[[Prototype]]`) of an object through which it is accessed.

The `__proto__` has been standardized in ES6 to ensure compatibility with web browsers. However, it may be deprecated in favor of `Object.getPrototypeOf()` in the future. Therefore, you should never use the `__proto__` in your production code.

The `p1.__proto__` exposes the `[[Prototype]]` that references the `Person.prototype` object.

Similarly, `p2.__proto__` also references the same object as `p1.__proto__`:

```
console.log(p1.__proto__ === Person.prototype); // true
console.log(p1.__proto__ === p2.__proto__); // true
```

As mentioned earlier, you should use the `Object.getPrototypeOf()` method instead of the `__proto__`. The `Object.getPrototypeOf()` method returns the prototype of a specified object.

```
console.log(p1.__proto__ === Object.getPrototypeOf(p1)); // true
```

Another popular way to get the prototype linkage is when the `Object.getPrototypeOf()` method is not available via the `constructor` property as follows:

```
p1.constructor.prototype
```

The `p1.constructor` returns `Person`, therefore, `p1.constructor.prototype` returns the prototype object.

Shadowing

See the following method call:

```
console.log(p1.greet());
```

The `p1` object doesn't have the `greet()` method defined, therefore JavaScript goes up to the prototype chain to find it. In this case, it can find the method in the `Person.prototype` object.

Let's add a new method to the object `p1` with the same name as the method in the `Person.prototype` object:

```
p1.greet = function() {
    console.log('Hello');
}
```

And call the `greet()` method:

```
console.log(p1.greet());
```

Because the `p1` object has the `greet()` method, JavaScript just executes it immediately without looking it up in the prototype chain.

This is an example of shadowing. The `greet()` method of the `p1` object shadows the `greet()` method of the `prototype` object which the `p1` object references.

Summary

- The `Object()` function has a property called `prototype` that references a `Object.prototype` object.

- The `Object.prototype` object has all properties and methods which are available in all objects such as `toString()` and `valueOf()` .
- The `Object.prototype` object has the `constructor` property that references the `Object` function.
- Every function has a `prototype` object. This prototype object references the `Object.prototype` object via `[[prototype]]` linkage or `__proto__` property.
- The prototype chain allows one object to use the methods and properties of its `prototype` objects via the `[[prototype]]` linkages.
- The `Object.getPrototypeOf()` method returns the prototype object of a given object. Do use the `Object.getPrototypeOf()` method instead of `__proto__` .

```
function Person(name) {
    this.name = name;
    this.dummyFunct = function() {
    }
}

let p1 = new Person('John');
Person.prototype.greet = function() {
    return "Hi, I'm " + this.name + "!";
} // both p1 and p2 will get this variable on prototype object
console.log(p1.greet())
let p2 = new Person('John');
Person.prototype.testvariable = 2; //both p1 and p2 will get this variable
p1.testvariable = 5; // creates a new property on p1 object
console.log("a")
```

```
Person {name: 'John', testvariable: 5, dummyFunct: ƒ}
  dummyFunct: ƒ ()
  name: "John"
  testvariable: 5
  [[Prototype]]: Object
    greet: ƒ ()
    testvariable: 2
    constructor: ƒ Person(name)
    [[Prototype]]: Object
      constructor: ƒ Object()
      hasOwnProperty: ƒ hasOwnProperty()
      isPrototypeOf: ƒ isPrototypeOf()
      propertyIsEnumerable: ƒ propertyIsEnumerable()
      toLocaleString: ƒ toLocaleString()
      toString: ƒ toString()
      valueOf: ƒ valueOf()
      __defineGetter__: ƒ __defineGetter__()
      __defineSetter__: ƒ __defineSetter__()
      __lookupGetter__: ƒ __lookupGetter__()
      __lookupSetter__: ƒ __lookupSetter__()
      __proto__: (...)
      get __proto__: ƒ __proto__()
      set __proto__: ƒ __proto__()
```

```
> p2
Person {name: 'John', dummyFunct: ƒ}
  dummyFunct: ƒ ()
  name: "John"
  [[Prototype]]: Object
    greet: ƒ ()
    testvariable: 2
    constructor: ƒ Person(name)
    [[Prototype]]: Object
      constructor: ƒ Object()
      hasOwnProperty: ƒ hasOwnProperty()
      isPrototypeOf: ƒ isPrototypeOf()
      propertyIsEnumerable: ƒ propertyIsEnumerable()
      toLocaleString: ƒ toLocaleString()
      toString: ƒ toString()
      valueOf: ƒ valueOf()
      __defineGetter__: ƒ __defineGetter__()
      __defineSetter__: ƒ __defineSetter__()
      __lookupGetter__: ƒ __lookupGetter__()
      __lookupSetter__: ƒ __lookupSetter__()
      __proto__: (...)
      get __proto__: ƒ __proto__()
      set __proto__: ƒ __proto__()
>
> Person.prototype
{testvariable: 2, greet: ƒ}
  greet: ƒ ()
  testvariable: 2
  constructor: ƒ Person(name)
  [[Prototype]]: Object
    constructor: ƒ Object()
    hasOwnProperty: ƒ hasOwnProperty()
    isPrototypeOf: ƒ isPrototypeOf()
    propertyIsEnumerable: ƒ propertyIsEnumerable()
    toLocaleString: ƒ toLocaleString()
    toString: ƒ toString()
    valueOf: ƒ valueOf()
    __defineGetter__: ƒ __defineGetter__()
    __defineSetter__: ƒ __defineSetter__()
    __lookupGetter__: ƒ __lookupGetter__()
    __lookupSetter__: ƒ __lookupSetter__()
    __proto__: (...)
    get __proto__: ƒ __proto__()
    set __proto__: ƒ __proto__()
>
```

```
> Object.prototype
< {__defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__: ƒ, __lookupSetter__: ƒ, …}
    i
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
      __proto__: (...)
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
>
```