Object-Oriented Programming in JavaScript

by Andrei Akula

List of contents

Object creation patterns:
Factory
Constructor
Prototype

Object creation

```
var person = new Object();
person.name = “Nicholas”;
person.age = 29;
person.job = “Software Engineer”;
person.sayName = function(){
     alert(this.name);
};
```

```
var person = {
     name: “Nicholas”,
     age: 29,
     job: “Software Engineer”,
     sayName: function(){
          alert(this.name);
     }
};
```

What about creating N objects person?

Object creation.
The Factory Pattern

```
function createPerson(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
    return o;
}

var person1 = createPerson(“Nicholas”, 29, “Software Engineer”);
var person2 = createPerson(“Greg”, 27, “Doctor”);

var personN = createPerson(. . .);
```

issue of object identification

Object creation.
The Constructor Pattern

```
function Person(name, age, job){
     this.name = name;
     this.age = age;
     this.job = job;
     this.sayName = function(){
          alert(this.name);
     };
}
```

- There is no object being created explicitly.
- The properties and method are assigned directly onto the `this` object.
- There is no `return` statement.

```
var person1 = new Person(“Nicholas”, 29, “Software Engineer”);
var person2 = new Person(“Greg”, 27, “Doctor”);
var personN = new Person(. . .);
```

Object creation.
The Constructor Pattern

`new` operator

1. Create a new object. Set prototype link to `Person.prototype`
2. Call constructor function **`Person`** with the specified arguments and **`this`** bound to the newly created object.
3. Return the new object.

```
var person1 = new Person(“Nicholas”, 29, “Software Engineer”);
var person2 = new Person(“Greg”, 27, “Doctor”);
```

```
> person1.constructor == Person
  true
> person2.constructor == Person
  true
```

```
> person1 instanceof Person
  true
> person2 instanceof Person
  true
```

Defining your own constructors ensures that instances can be identified as a particular type later on

Object creation.
Problems with Constructors

Methods are created once for each instance.

```
function Person(name, age, job){
     this.name = name;
     this.age = age;
     this.job = job;
     this.sayName = new Function("alert(this.name)");  //logical equivalent
}

>  person1.sayName == person2.sayName
   false
```

Object creation.
The Prototype Pattern

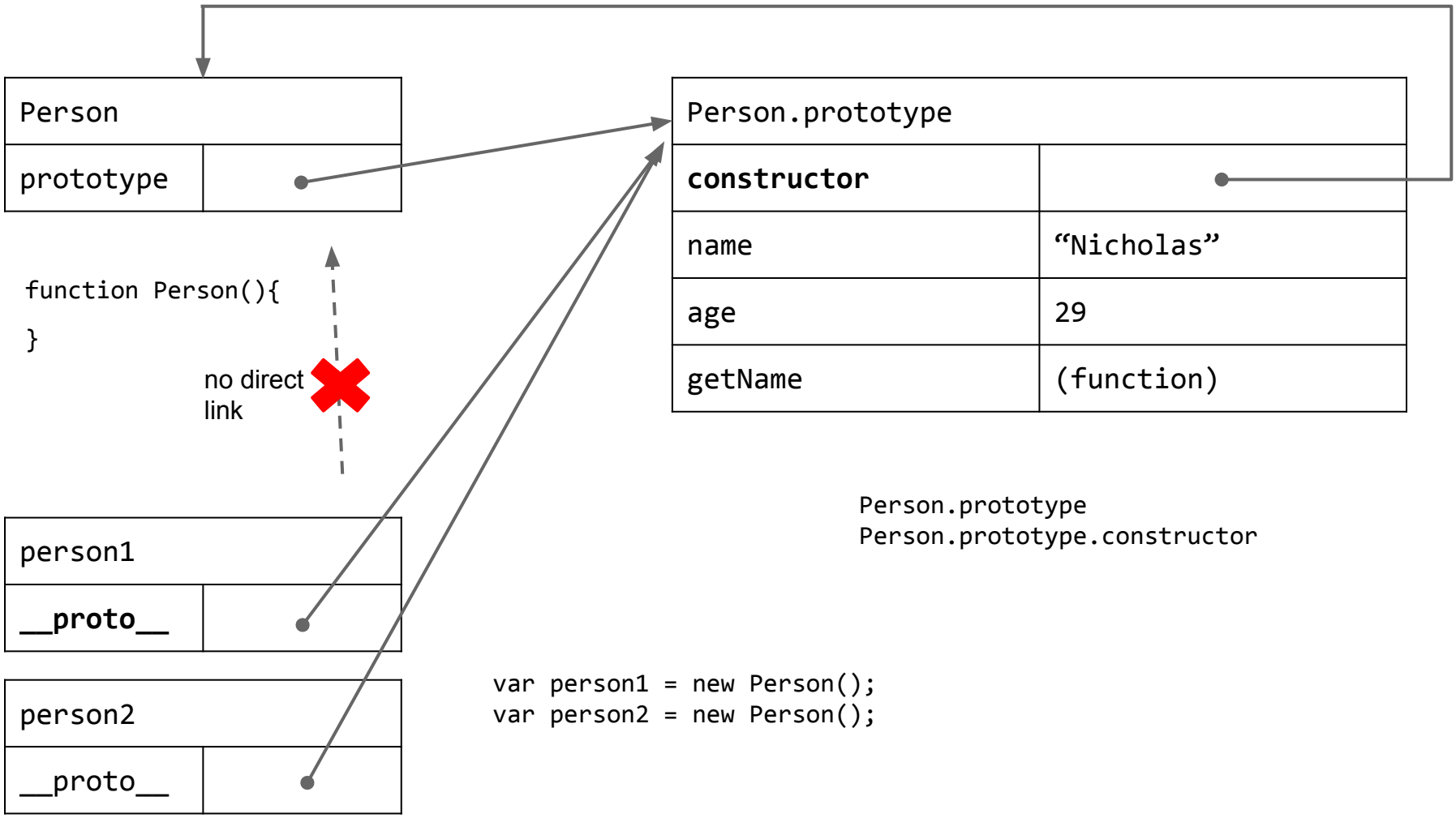
All of prototype properties and methods are shared among object instances

```
function Person(){
}
Person.prototype.name = “Nicholas”;
Person.prototype.age = 29;
Person.prototype.getName = function(){
      return this.name;
};

var person1 = new Person();     var person2 = new Person();
```

```
> person1.getName() === person2.getName()
  true
```

```
> person1.getName == person2.getName
  true
```

How Prototypes Work



How Prototypes Work

```
function Person() {
}
Person.prototype = {
     constructor: Person,
     name: “Nicholas”,
     age: 29,
     getName:  function() { return this.name; }
};
```

No `this` assignments

```
> var person = new Person();
> person.getName() // what returns and why ?
> person.name = “instanceName”;
> person.getName() // ?
```

```
> delete person.name
> person.getName() // what returns and why ?
> delete person.name
> person.getName() // ?
```

Prototype Chain

```
var a = { name: “a” };
var b = { name: “b” };
var c = { name: “c” };
```

```
b.__proto__ = a;
c.__proto__ = b;
```

```
console.dir(c)
```

```
▼ Object i
    name: "c"
  ▼ __proto__: Object
      name: "b"
    ▼ __proto__: Object
        name: "a"
      ▼ __proto__: Object
        ▶ __defineGetter__: function __defineGetter__() { [native code] }
        ▶ __defineSetter__: function __defineSetter__() { [native code] }
        ▶ __lookupGetter__: function __lookupGetter__() { [native code] }
        ▶ __lookupSetter__: function __lookupSetter__() { [native code] }
        ▶ constructor: function Object() { [native code] }
        ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
        ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
        ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }
        ▶ toLocaleString: function toLocaleString() { [native code] }
        ▶ toString: function toString() { [native code] }
        ▶ valueOf: function valueOf() { [native code] }
        ▶ get __proto__: function __proto__() { [native code] }
        ▶ set __proto__: function __proto__() { [native code] }
```

```
c.toString()
"[object Object]"

c.toString = function(){
  return this.name;
}

c.toString()
"c"
```

Own Properties and prototype Properties

```
function Person(){
      this.name = “Nicholas”;
}
Person.prototype.getName = function(){
      return this.name;
};

var person = new Person();
person.age = 29;
```

```
person.hasOwnProperty(“age”)      // true
person.hasOwnProperty(“name”)     // ?
person.hasOwnProperty(“getName”)  // ?

console.log(“age” in person)     // true
console.log(“name” in person)    // ?
console.log(“getName” in person) // ?

for(var n in person) {
      console.log(n);       // ???
}
```

```
for(var n in person) {
      if (person.hasOwnProperty(n))
            console.log(n);
      }
}
```

⟺

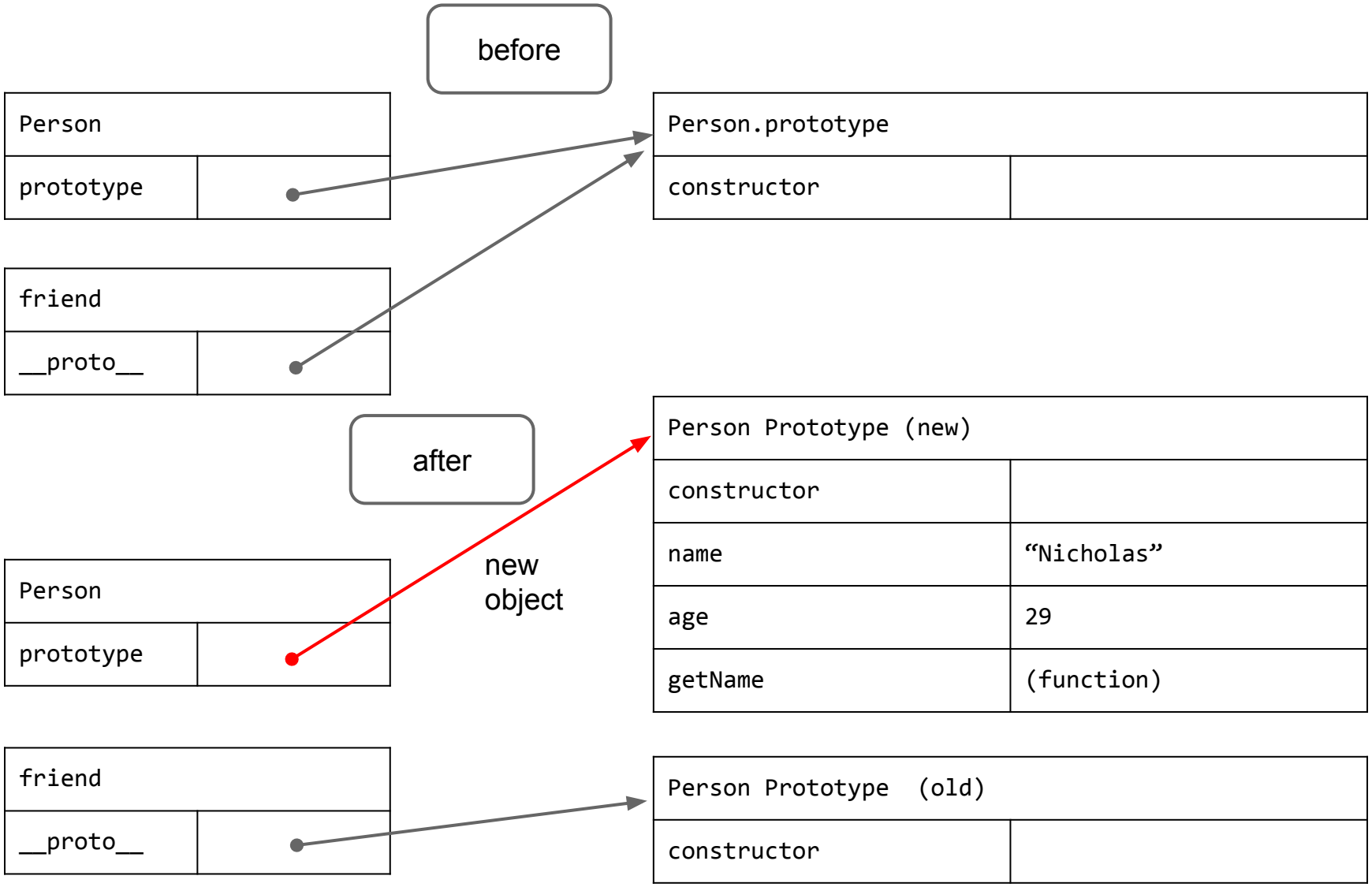
```
var props = Object.keys(person);
for(var i=0; i<props.length; i++) {
      console.log(props[i]);
}
```

Dynamic Nature of Prototypes

```
var friend = new Person();
Person.prototype.sayHi = function(){
     console.log(“hi”);
};
friend.sayHi();
// ”hi” - works!
```

```
function Person(){
}
var friend = new Person();

Person.prototype = {
     constructor: Person,
     name : “Nicholas”,
     age : 29,
     getName : function () {
          return this.name;
     }
};

friend.getName(); //  Error
```

Dynamic Nature of Prototypes



How get Prototype of the object?

```
Object.getPrototypeOf(obj)
```

```
function Person(){
}
var person = new Person;

Object.getPrototypeOf(person) === Person.prototype  // true

Object.getPrototypeOf(person) === person.__proto__  // ?
```

Problems with Prototypes

a reference value

```
function Person(){
}
Person.prototype = {
      constructor: Person,
      name : “Nicholas”,
      age : 29,
      friends : [“Shelby”, “Court”],
      getName : function () {
            return this.name;
      }
};

var person1 = new Person();
var person2 = new Person();

person1.friends.push(“Van”);     // what about friends in person2 ?

console.log(person1.friends);    // [”Shelby, Court, Van”]
console.log(person2.friends);    // [”Shelby, Court, Van”]

(person1.friends === person2.friends);  //true
```

Combination Constructor/Prototype Pattern

```
function Person(name, age){
     this.name = name;
     this.age = age;
     this.friends = [“Shelby”, “Court”];
}

Person.prototype.getName = function () {
     return this.name;
};


var person1 = new Person(“Andrew”, 29);
var person2 = new Person(“David”, 25);

var personN = new Person(“Lloyd”, 18);
```

- Own instances of name, age and friends
- Share single function `getName`

Inheritance in EcmaScript 5

Inheritance

```
function SuperType(name){
     this.name = name;
     this.colors = [“red”, “blue”, “green”];
}
SuperType.prototype.sayName = function(){
     return “Name is ” + this.name;
};

function SubType(age){
     this.age = age;
}
     // inherit Prototype
SubType.prototype = new SuperType();
SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){
     return “Age is ” + this.age;
};

var instance = new SubType(29);
```

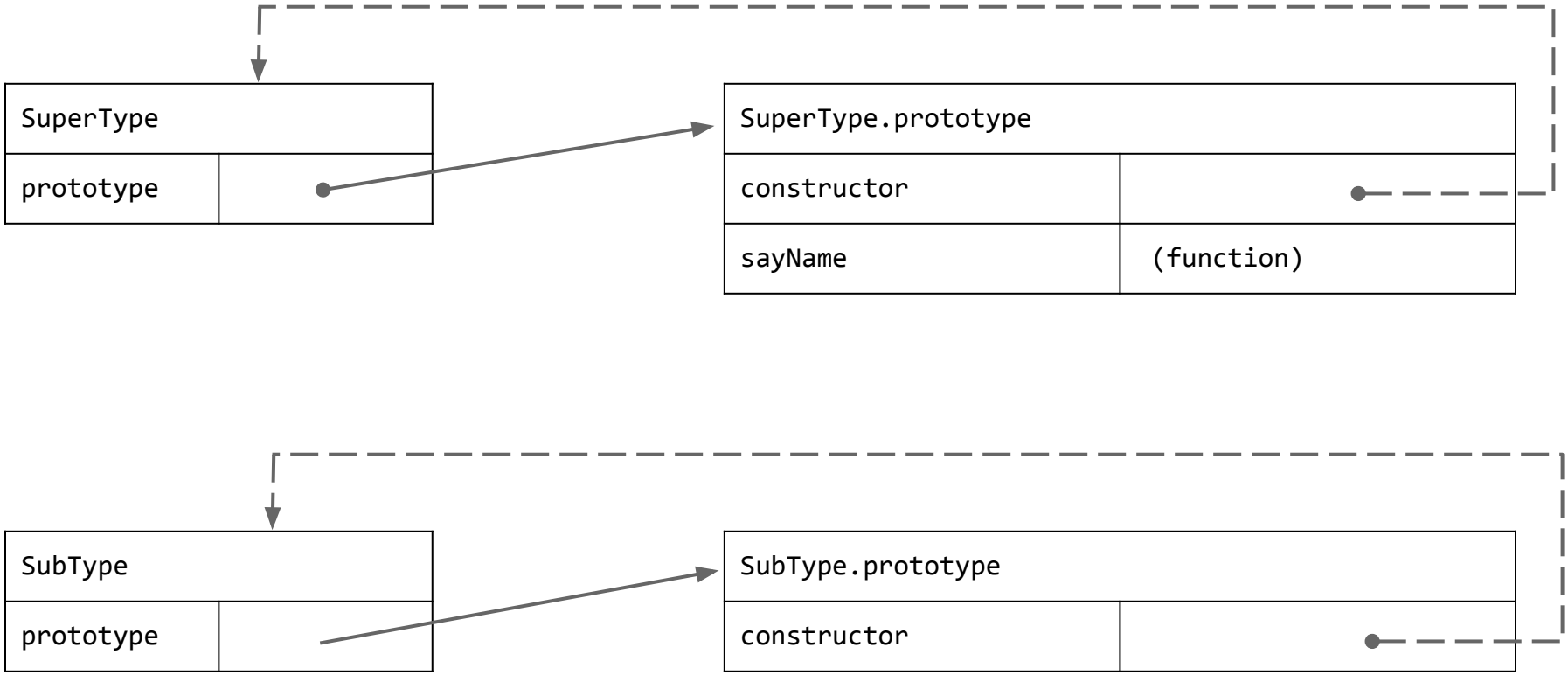
1

2

3

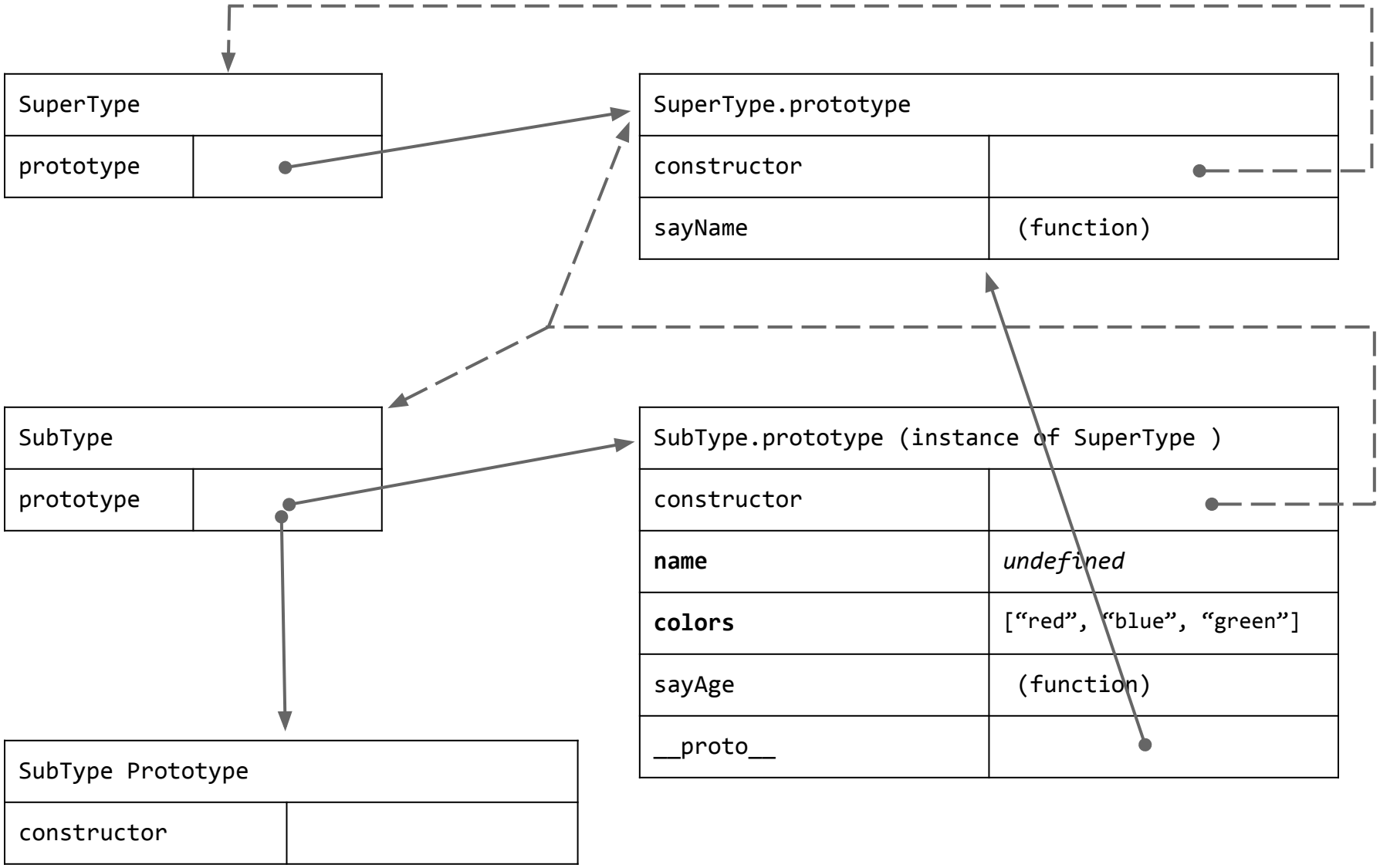
Inheritance

1



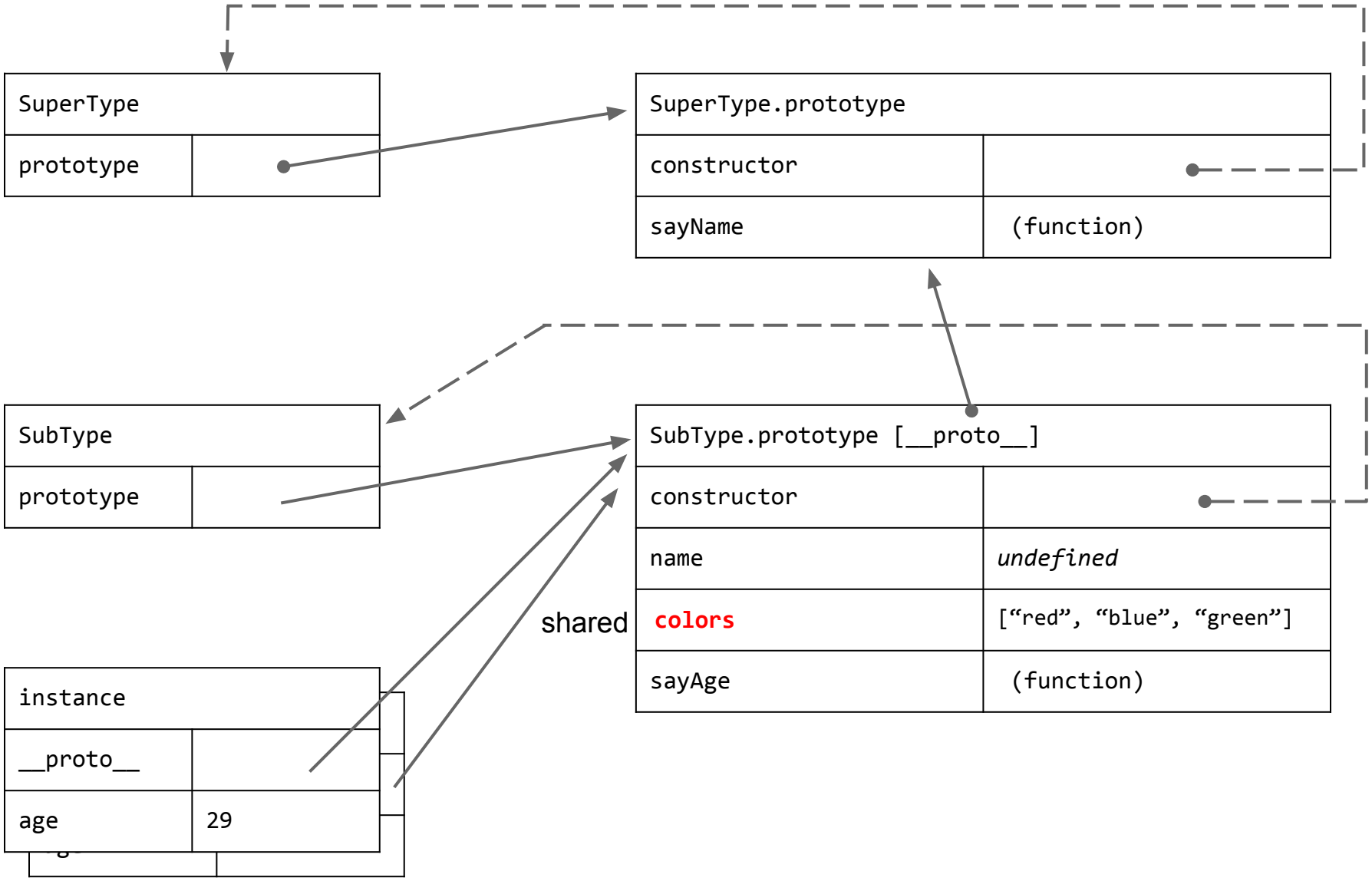
Inheritance

2



Inheritance

3

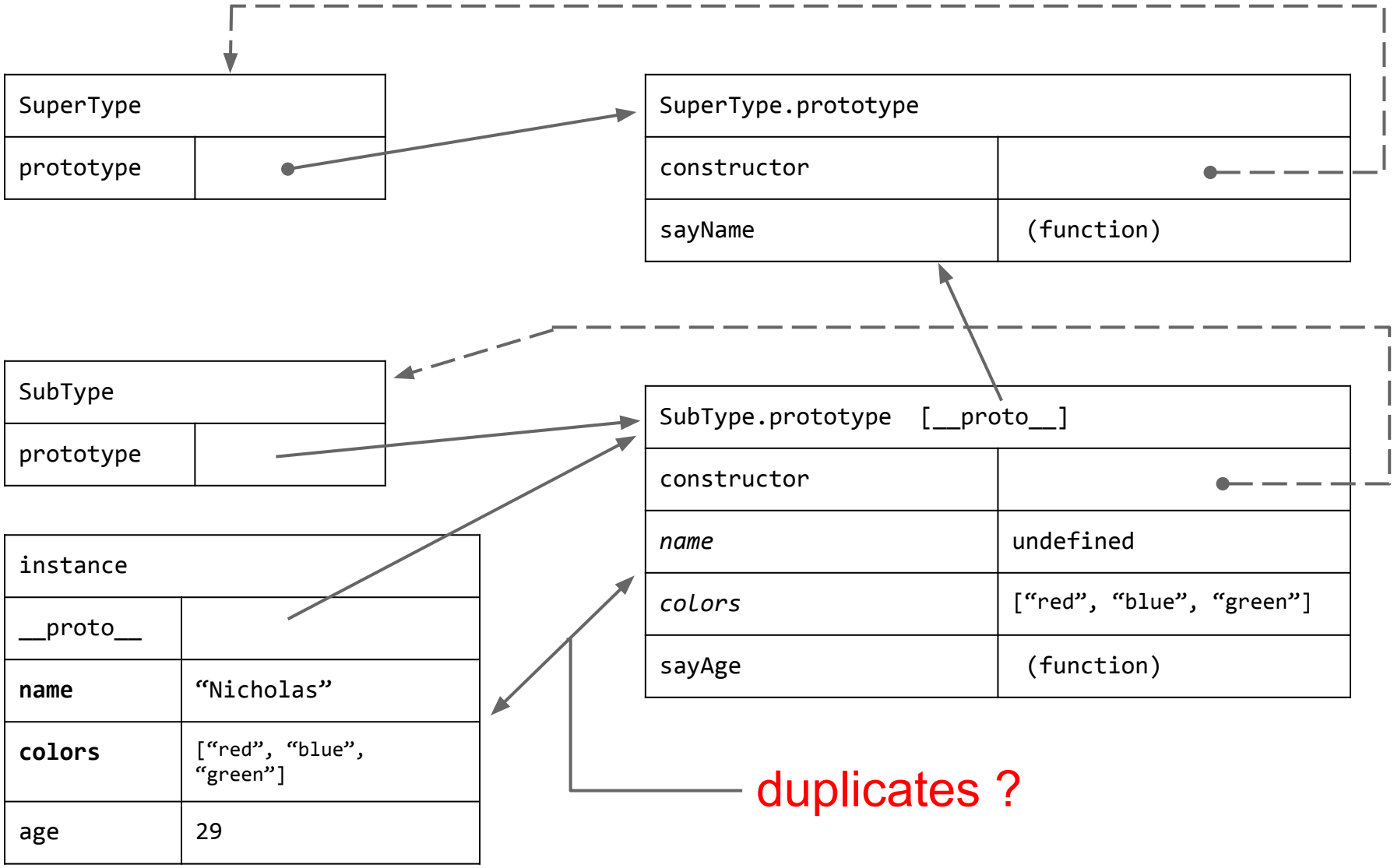


Inheritance. Borrowing a Constructor

```
function SuperType(name){
     this.name = name;
     this.colors = [“red”, “blue”, “green”];
}
SuperType.prototype.sayName = function(){
     return “Name is ” + this.name;
};

function SubType(name, age){
     SuperType.call(this, name);
     this.age = age;
}

SubType.prototype = new SuperType();           // inherit Prototype
SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){
     return “Age is ” + this.age;
};

var instance = new SubType(“Nicholas”, 29);
```

Inheritance. Borrowing a Constructor

3



Inheritance. Borrowing a Constructor

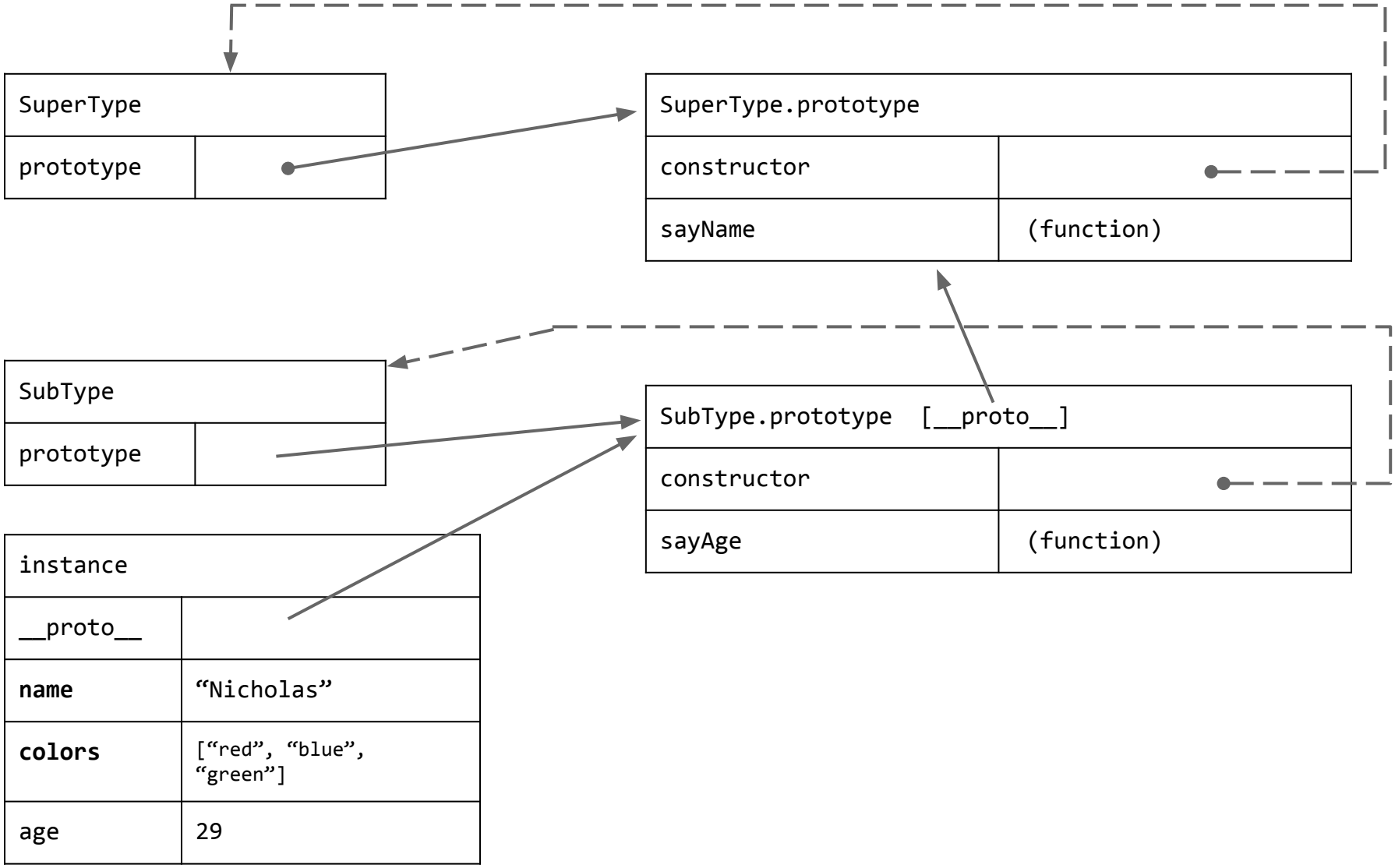
```
function SuperType(name){
     this.name = name;
     this.colors = [“red”, “blue”, “green”];
}
SuperType.prototype.sayName = function(){
     return “Name is ” + this.name;
};

function SubType(name, age){
     SuperType.call(this, name);          //second call to SuperType()
     this.age = age;
}

SubType.prototype = new SuperType();      //first call to SuperType()
SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){
     return “Age is ” + this.age;
};

var instance = new SubType(“Nicholas”, 29);
```

Inheritance. Single call to SuperType

```
function SuperType(name){
      this.name = name;
      this.colors = [“red”, “blue”, “green”];
}
SuperType.prototype.sayName = function(){
      return “Name is ” + this.name;
};

function SubType(name, age) {
      SuperType.call(this, name);
      this.age = age;
}

inheritPrototype(SubType, SuperType);

SubType.prototype = Object.create(SuperType.prototype);
SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){
      return “Age is ” + this.age;
};

var instance = new SubType(“Nicholas”, 29);
```

```
function inheritPrototype(subType, superType) {
      function TempCtor() {}
      TempCtor.prototype = superType.prototype;

      var prototype = new TempCtor();

      subType.prototype = prototype;
      subType.prototype.constructor = subType;
}
```

Inheritance. Single call to SuperType



Classes and Inheritance in EcmaScript 6

Class

EcmaScript 6

```
class Person {
     constructor(name, age) {
          this.name = name;
          this.age = age;
     }

     getName() {
          return this.name;
     }
}


var person = new Person(“Nicholas”, 29);
```

EcmaScript 5

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


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

Inheritance

EcmaScript 6

```
class SuperType {
      constructor(name) {
            this.name = name;
      }

      sayName() {
            return “Name is ” + this.name;
      }
}

class SubType extends SuperType {
      constructor(name, age) {
            super(name);
            this.age = age;
      }

      sayAge() {
            return “Age is ” + this.age;
      }
}

var instance = new SubType(“Nicholas”, 29);
```

EcmaScript 5

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

SuperType.prototype.sayName = function(){
      return “Name is ” + this.name;
};



function SubType(name, age) {
      SuperType.call(this, name);
      this.age = age;
}

SubType.prototype =
            Object.create(SuperType.prototype);
SubType.prototype.constructor = SubType;


SubType.prototype.sayAge = function(){
      return “Age is ” + this.age;
};

var instance = new SubType(“Nicholas”, 29);
```

Polymorphism

Different classes might define the same method or property.

Polymorphism

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

SuperType.prototype.sayName = function(){
      return “Name is ” + this.name;
};



function SubType(name, age) {
      SuperType.call(this, name);
      this.age = age;
}

SubType.prototype =
            Object.create(SuperType.
      prototype);
SubType.prototype.constructor = SubType;


SubType.prototype.sayAge = function(){
      return “Age is ” + this.age;
};

SubType.prototype.sayName = function(){
      return this.name + “ is name” ;
};
```

```
function sayName(type) {
     if (type instanceof SuperType) {
           return type.sayName();
     }
     return “”;
}

var instance1 = new SuperType(“Super”);
var instance2 = new SubType(“Sub”, 33);

sayName(instance1);  //   “Name is Super”
sayName(instance2);  //   “Sub is name”
```

Encapsulation
Privileged methods

Private data and methods in Constructors

```
function MyType() {
    //private variable
    var privateVariable = 10;

    //private function
    function privateFunction() {
        return false;
    }

    //privileged method
    this.publicMethod = function () {
        privateVariable++;
        return privateFunction();
    };
}
```

Private data and methods in Prototypes

```
var MyType = (function() {
    //private variable
    var privateVariable = 10;
    //private function
    function privateFunction() {
        return false;
    }
    // constructor function
    var T = function() {
    };
    // prototype methods
    T.prototype.publicMethod = function () {
        privateVariable++;
        return privateFunction();
    };
    return T;
})();
```

```
var instance = new MyType();
```

Object property attributes

What we can do with properties?

Get list of properties

```
for(var n in person) {
     console.log(n);
}
```

```
name
age
job
sayName
```

Change a property

```
person.age = 31;

>console.log(person.age);
 31
```

Can we hide a property from enumeration?

Can we define a property to be read only?

Delete a property

```
delete person.age;

>console.log(person.age);
  undefined
```

Property attributes

- `[[Enumerable]]` — Indicates if the property will be returned in a for-in loop.
- `[[Writable]]` — Indicates if the property's value can be changed.
- `[[Value]]` — Contains the actual data value for the property.
- `[[Configurable]]` — Indicates if the property may be redefined by removing the property via delete, changing the property's attributes, or changing the property into an accessor property.

To change any of the default property attributes, you must use the ECMAScript 5

```
Object.defineProperty(obj, prop, descriptor)
```

Enumerable attribute

```
var person = {
     job: “Software Engineer”
};


Object.defineProperty(person, “name”, {
     enumerable: false,
     value: “HiddenMan”
});
```

```
> console.log(person.name);
  HiddenMan

> Object.keys(person)
  [“job”]
```

```
> Object.getOwnPropertyNames(person)
  ["job", "name"]
```

Writable attribute

```
var person = {
};


Object.defineProperty(person, “name”, {
     writable: false,
     value: “Pedro”
});
```

```
> console.log(person.name);
  Pedro
> person.name = “Alexandros”;
> console.log(person.name);
  Pedro
```

```
> delete person.name
< false
> console.log(person.name);
  Pedro
```

Configurable attribute

```
var person = {
};
```

```
Object.defineProperty(person, “name”, {
    configurable: false,
    value: “Pedro”
});
```

```
> console.log(person.name);
  Pedro
> person.name = “Alexandros”;
> console.log(person.name);
  Pedro
```

```
> delete person.name
< false
> console.log(person.name);
  Pedro
```

```
Object.defineProperty(person, “name”, {
    configurable: true,
    value: “Pedro”
});
```

```
Uncaught TypeError: Cannot redefine
property: name
```

Accessor Properties

```
var book = {
      _year: 2004,
      edition: 1
};

Object.defineProperty(book, “year”, {
      get: function(){
            return this._year;
      },
      set: function(newValue){
            if (newValue > 2004) {
                  this._year = newValue;
                  this.edition += newValue - 2004;
            }
      }
});
```

```
> book.year = 2005;
> console.log(book.edition);
  2
```