

Prototypes and Inheritance

Prototypes, Prototype Chain, Class Inheritance



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>



sli.do

#js-advanced

1. Internal Object Properties

- Enumerable, Configurable, Writable, Value

2. Inheritance

- Types of Inheritance

3. The Prototype

- Constructor Functions
- Prototype Chaining

4. Class Inheritance (ES6)





Internal Object Properties

- Every object field has **four** properties:
 - **Enumerable** - can access to all of them using a **for...in** loop
 - Enumerable property are returned using **Object.keys** method
 - **Configurable** - can **modify** the **behavior** of the property
 - You **can delete** only **configurable** properties
 - **Writable** - can **modify** their **values** and update a property just assigning a new value to it
 - **Value**

Object's Non-enumerable Properties

- They won't be in for...in iterations
- They won't appear using Object.keys function
- They are not serialized when using JSON.stringify

```
let ob = {a:1, b:2};  
ob.c = 3;  
Object.defineProperty(ob, 'd', { value: 4, enumerable: false });  
ob.d; // => 4  
for( let key in ob ) console.log( ob[key] ); //1 2 3  
Object.keys( ob ); // => ["a", "b", "c"]  
ob; // => {a: 1, b: 2, c: 3, d: 4}  
ob.d; // => 4
```

Object's Non-writable Properties

- Once its value is defined, it is **not possible to change** it using assignments

```
let ob = { a: 1 };  
Object.defineProperty(ob, 'B', { value: 2, writable: false });  
ob.B; // => 2  
ob.B = 10;  
ob.B; // => 2
```

- If the non-writable property **contains** an **object**, the **reference** to the object is what is **not writable**, but the **object** itself **can be modified**

Object's Non-configurable Properties

- Once you have defined the property as **non-configurable**, there is only **one behavior** you **can change**
 - If the property is **writable**, you can convert it to non-writable
 - Any other try of definition update will **fail** throwing a `TypeError`

```
let ob = {};  
Object.defineProperty(ob, 'a', { configurable: false, writable: true });  
Object.defineProperty(ob, 'a', { enumerable: true }); // throws a TypeError  
Object.defineProperty(ob, 'a', { value: 12 }); // 12  
Object.defineProperty(ob, 'a', { writable: false }); // This is allowed!!  
Object.defineProperty(ob, 'a', { value: 12 }); // // throws a TypeError  
Object.defineProperty(ob, 'a', { writable: true }); // throws a TypeError  
delete ob.a; // => false
```


Object Freeze and Seal

```
let cat = { name: 'Tom', age: 5 };  
Object.freeze(cat);  
cat.age = 10;           // Error in strict mode  
cat.gender = 'male';    // Error in strict mode  
console.log(cat);       // { name: 'Tom', age: 5 }
```

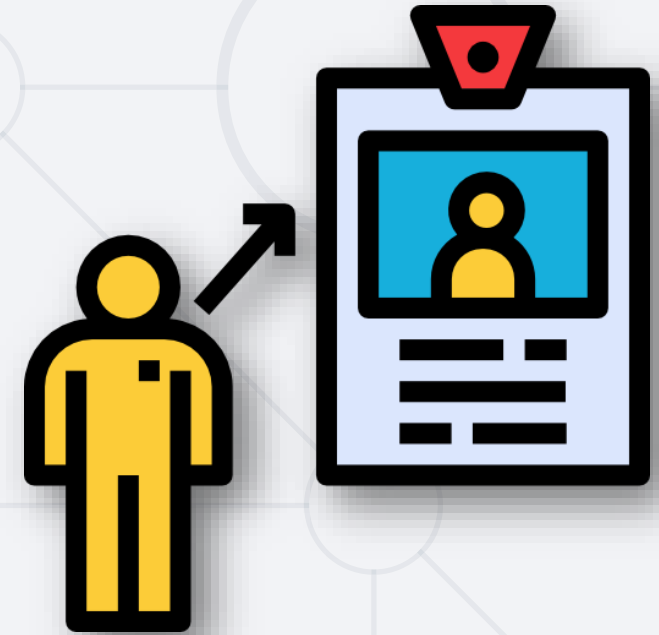
```
cat = { name: 'Tom', age: 5 };  
Object.seal(cat);  
cat.age = 10;           // OK  
delete cat.age;         // Error in strict mode  
console.log(cat);       // { name: 'Tom', age: 10 }
```

- Return an object with **firstName**, **lastName** and **fullName**
 - If **firstName** or **lastName** are **changed**, then **fullName** should **also** be changed
 - If **fullName** is changed, then **firstName** and **lastName** should also be changed

```
let person = createPerson("Albert", "Simpson");  
console.log(person.fullName); //Albert Simpson  
person.firstName = "Simon";  
console.log(person.fullName); //Simon Simpson
```

Solution: Person

```
function createPerson(firstName, lastName) {  
  const result = {  
    firstName,  
    lastName  
  };  
  
  Object.defineProperty(result, "fullName", {  
    get() { // calculate and return value },  
    set(value) { // set value + validation }  
  });  
  return result;  
}
```





Inheritance

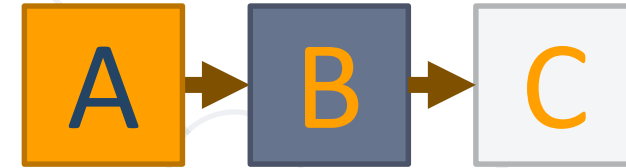
Types of Inheritance



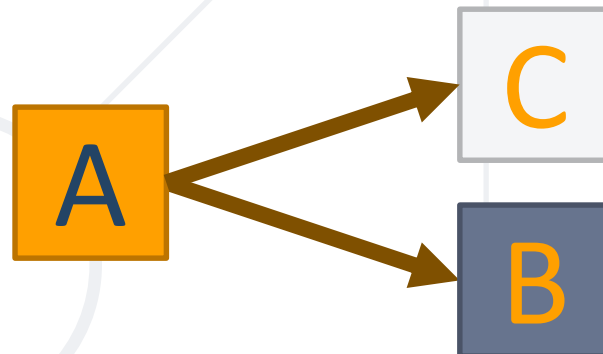
Single Inheritance



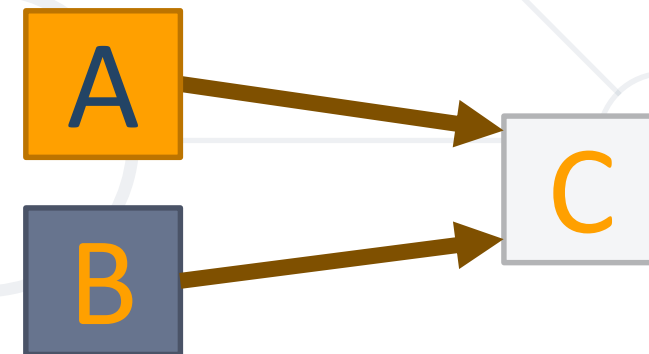
Multilevel Inheritance



Hierarchical Inheritance



Multiple Inheritance*




** Not supported in JS with **classes**, but works with **composition***



The Prototype

What is a Prototype?

- 
- Every object in JS has a **prototype** (template)
 - Internally called **__proto__** in browsers and NodeJS
 - Properties lookup follows the **prototype chain**
 - Obtained with **`Object.getPrototypeOf(obj)`**
 - **Reference** to another objects
 - Objects are **not** separate and disconnected, but **linked**


*Note: **__proto__** is for debugging and should **never** be used in production code!*

- Objects **inherit properties** and **methods** from a **prototype**
- The **prototype property** allows you to add **new properties** to object **constructors**

```
function Person(first, last, age) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
}  
Person.prototype.nationality = "Bulgarian";
```


Simulated Class Functionality

- Before ES6, **classes** were composed **manually**



```
function Rectangle(width, height) {  
  this.width = width;  
  this.height = height;  
}  
  
Rectangle.prototype.area = function () {  
  return this.width * this.height;  
}  
  
let rect = new Rectangle(3, 5);
```

Comparison with the New Syntax



```
class Rectangle {  
    constructor(width, height) {  
        this.width = width;  
        this.height = height;  
    }
```



```
    area() {
```

```
        return this.width * this.height;
```

```
    }
```

```
}
```



```
function Rectangle(width, height) {  
    this.width = width;  
    this.height = height;  
}
```

```
Rectangle.prototype.area = function () {  
    return this.width * this.height;  
}
```

Object Creation

- **Literal** creation
- **Constructor** creation (function constructors)
 - Have an **implicit reference** (prototype) to the value of their constructor's "prototype" property
 - Gets an internal **__proto__ link** to the object



■ Literals

```
let bar = {  
  me: "I am b1",  
  speak: function() {  
    console.log("Hello, " +  
      this.me + ".");  
  }  
};
```

■ Constructed

```
function Bar(name) {  
  this.me = "I am " + name;  
  this.speak = function() {  
    console.log("Hello, " +  
      this.me + ".");  
  };  
}; let b1 = new Bar("b1");
```



- The **Object.create()** method creates a **new object**, using an existing object as **prototype**

```
const dog = {  
  name: 'Sparky',  
  printInfo: function() {console.log(`My name is ${this.name}`)}}  
};
```

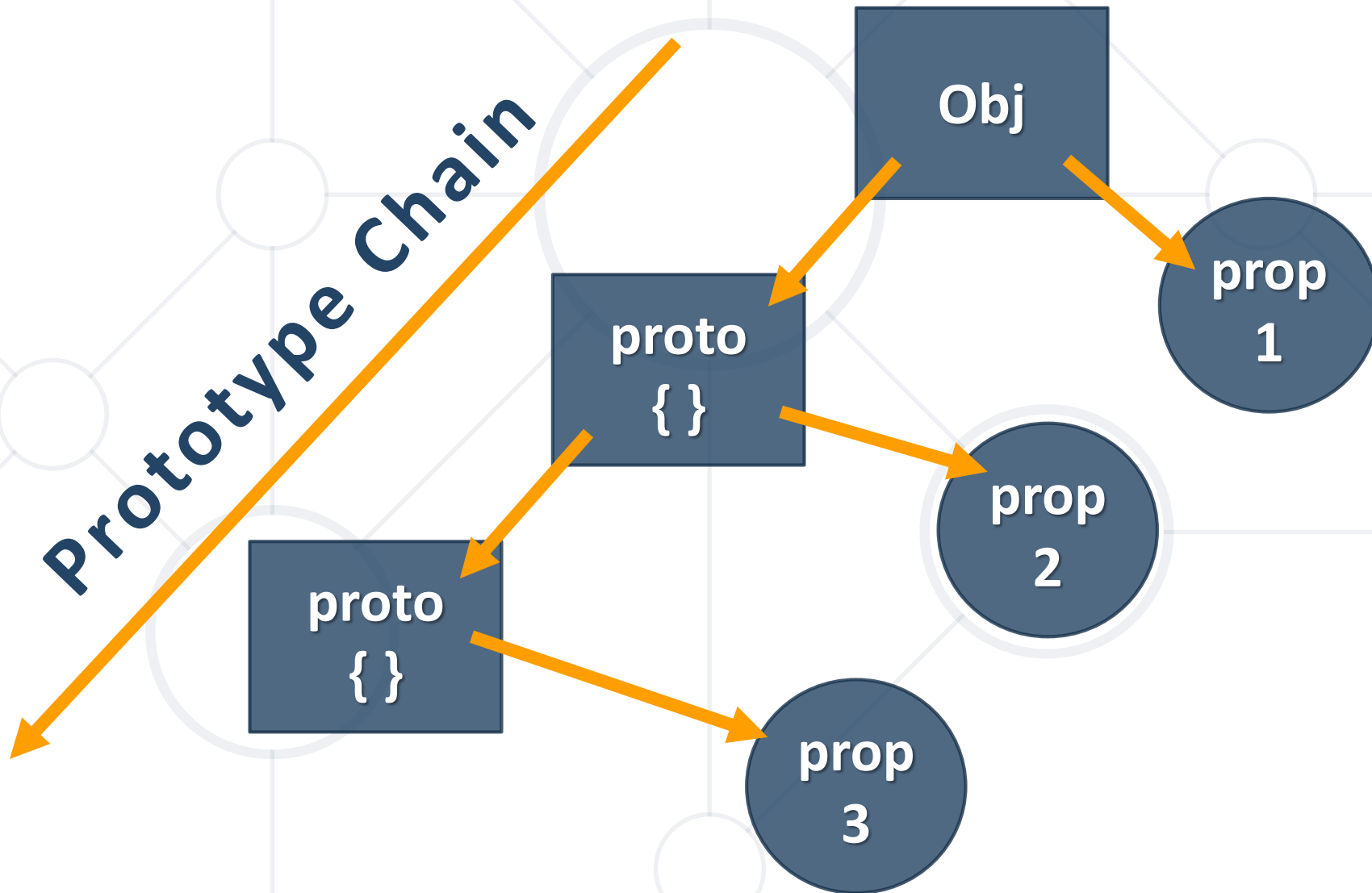
```
const myDog = Object.create(dog);  
myDog.name = 'Max'; // inherited properties can be overwritten  
myDog.breed = 'shepherd'; // breed is a property of myDog  
myDog.printInfo(); // My name is Max
```

__proto__ vs Prototype Property

- **__proto__**
 - Property of an objects that **points** at the prototype that has been **set**
 - Using **__proto__** directly is deprecated!
- **prototype**
 - Property of **a function** set if your object is created by a **constructor function**
 - Objects do not have **prototype** property



Prototype Chain



Prototype Chain - Simple Example

```
function Foo(y) {  
    this.y = y;  
}  
  
Foo.prototype.x = 10;  
Foo.prototype.calculate = function (z) {  
    return this.x + this.y + z;  
};  
  
let b = new Foo(20);  
console.log(Foo.prototype); // { x: 10, calculate: [Function] }  
console.log(b.calculate(30)); // 60
```


Prototype Inheritance

```
function Foo(who) {  
    this.me = who;  
}  
Foo.prototype.identify = function () { return "I am " + this.me; }  
function Bar(who) { Foo.call(this, who); }  
  
Bar.prototype = Object.create(Foo.prototype);  
Bar.prototype.speak = function () {  
    console.log("Hello, " + this.identify() + ".");  
}  
let b1 = new Bar("b1");  
let b2 = new Bar("b2");  
b1.speak(); b2.speak();
```

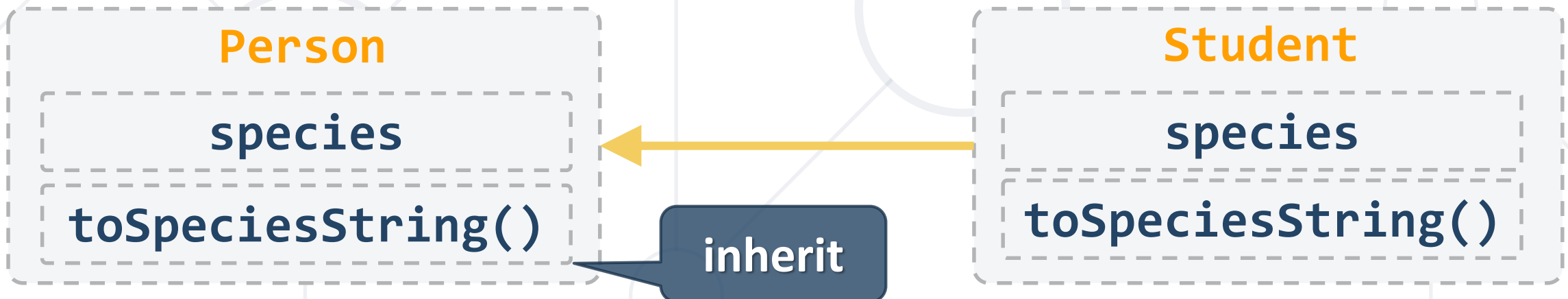
Problem: Extending Prototype

- Extend a passed class's **prototype** with a property **species** and method **toSpeciesString()**:
 - **Person.prototype.species** - holds a string value *"Human"*
 - **Person.prototype.toSpeciesString()** - returns
 - *"I am a {species}. {class.toString()}"*

```
new Person("Maria", "maria@gmail.com").toSpeciesString()  
// "I am a Human. Person (name: Maria, email: maria@gmail.com)"
```

Solution: Extending Prototype

```
function extendPrototype(Class) {  
    Class.prototype.species = "Human";  
    Class.prototype.toSpeciesString = function () {  
        return `I am a ${this.species}. ${this.toString()}`;  
    }  
}  
  
extendPrototype(Person);
```





Class Inheritance (ES6)

Traditional Classes

- Classes are a **design pattern**
- Classes mean - creating **copies**
 - When **instantiated** – a **copy** from class to instance
 - When **inherited** – a **copy** from parent to child
- Class inheritance is a powerful tool, but has many **drawbacks** and **limitation**
 - **Composition** should be **preferred** whenever possible!



Class Inheritance

- Classes can **inherit** (extend) other classes
 - Child class inherits data + methods from its parent
- **Child class** can:
 - Add **properties** (data)
 - Add / replace **methods**
 - Add / replace **access** properties
- Use the keyword **extends**

Base (parent,
super) class

Person

name

email



Teacher

subject

Child (derived) class



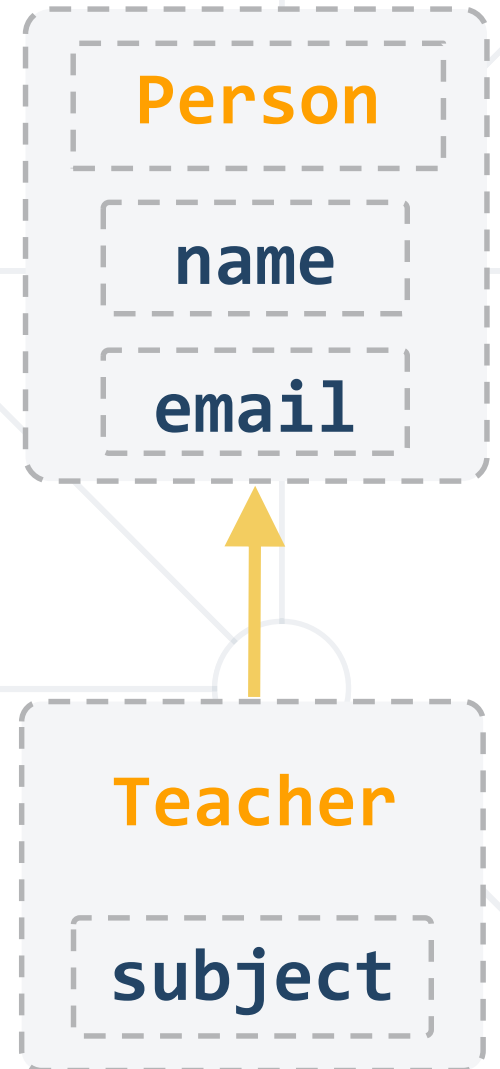
Class Inheritance - Example

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

class **Teacher**
inherits **Person**

```
class Teacher extends Person {  
    constructor(name, email, subject) {  
        super(name, email);  
        this.subject = subject;  
    }  
}
```

Invoke the parent
constructor



Class Inheritance - Example

```
let p = new Person("Maria", "maria@gmail.com");  
console.log("Person: " + p.name + ' (' + p.email + ')');  
// Person: Maria (maria@gmail.com)
```

```
let t = new Teacher("Ivan", "iv@yahoo.com", "PHP");  
console.log("Teacher: " + t.name +  
    ' (' + t.email + '), teaches ' + t.subject);  
// Teacher: Ivan (iv@yahoo.com), teaches PHP
```


Classes in JavaScript

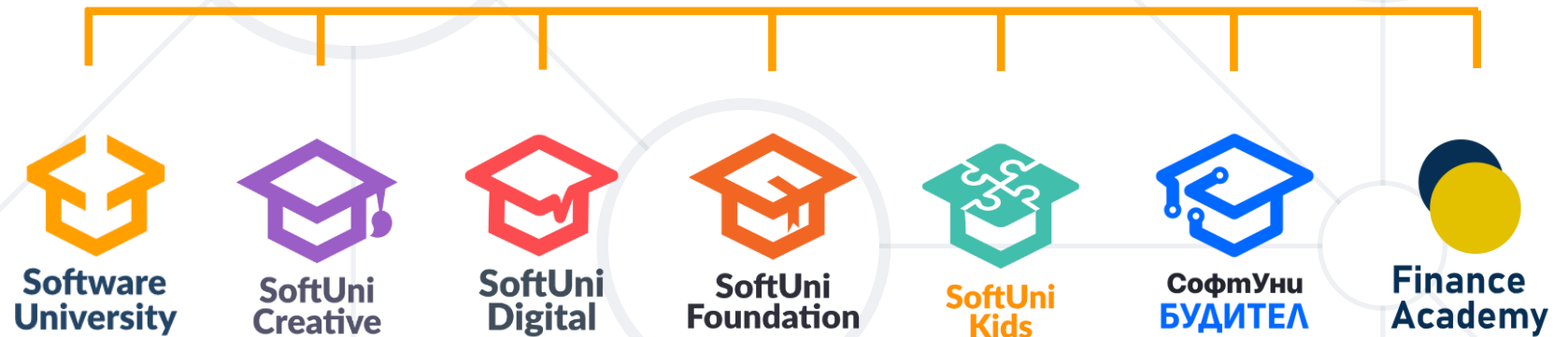


- **Prototypal inheritance** instead of classical inheritance
- **Does not automatically** create copies
- Common keys and values are shared by **reference**
- **Delegates not blueprints!**

- Inheritance allows **extending** existing classes
 - Child class inherits **data + methods** from its parent
- Objects in JS have **prototypes**
 - Objects look for **properties** in their prototypes
 - Prototypes form a **hierarchical chain**



Questions?



SoftUni Diamond Partners



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg, about.softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity

