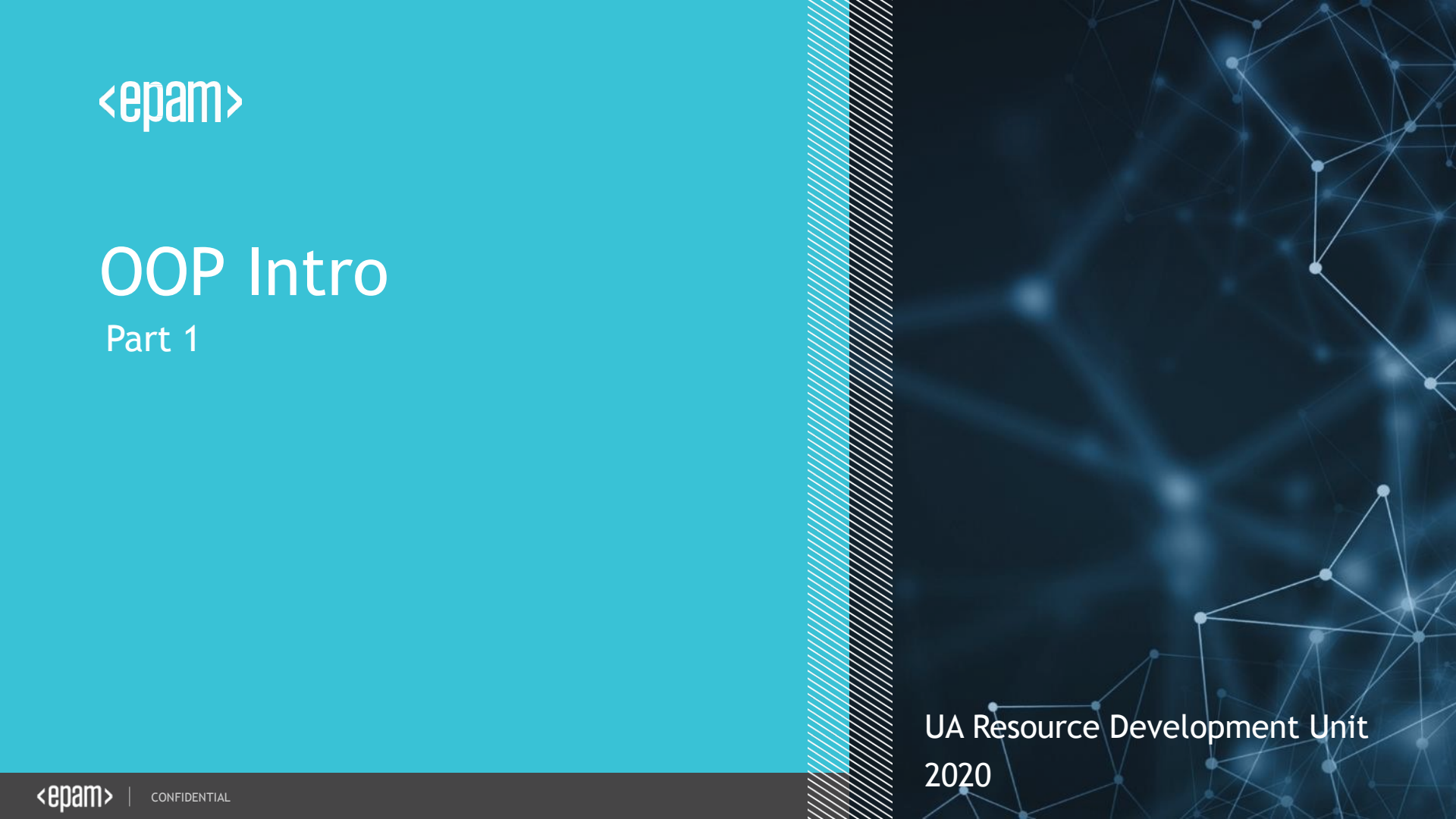
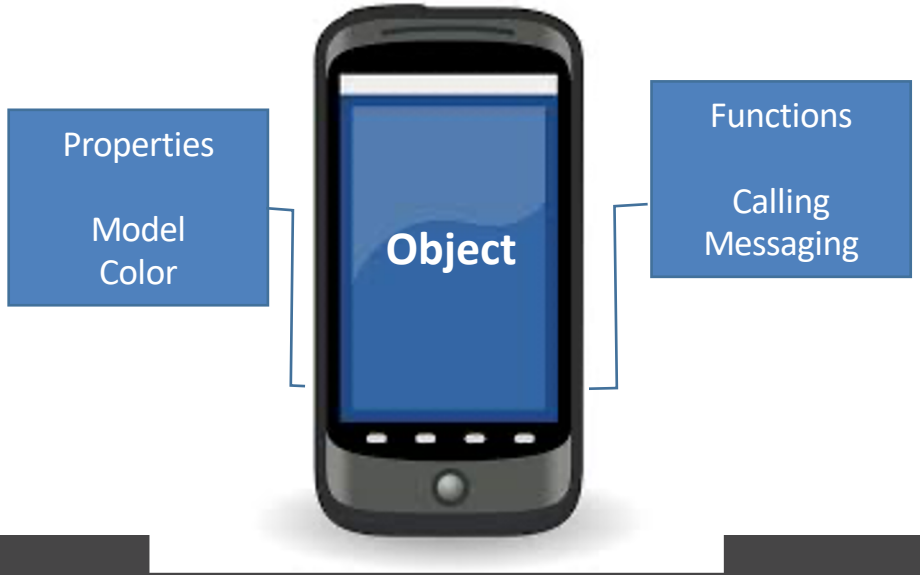

epam
OOP Intro
Part 1
UA Resource Development Unit
2020
epam
CONFIDENTIAL

AGENDA

1. **Object**
2. Prototype
3. **Constructor**
4. Literals Objects
5. Classes

Object

An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method.

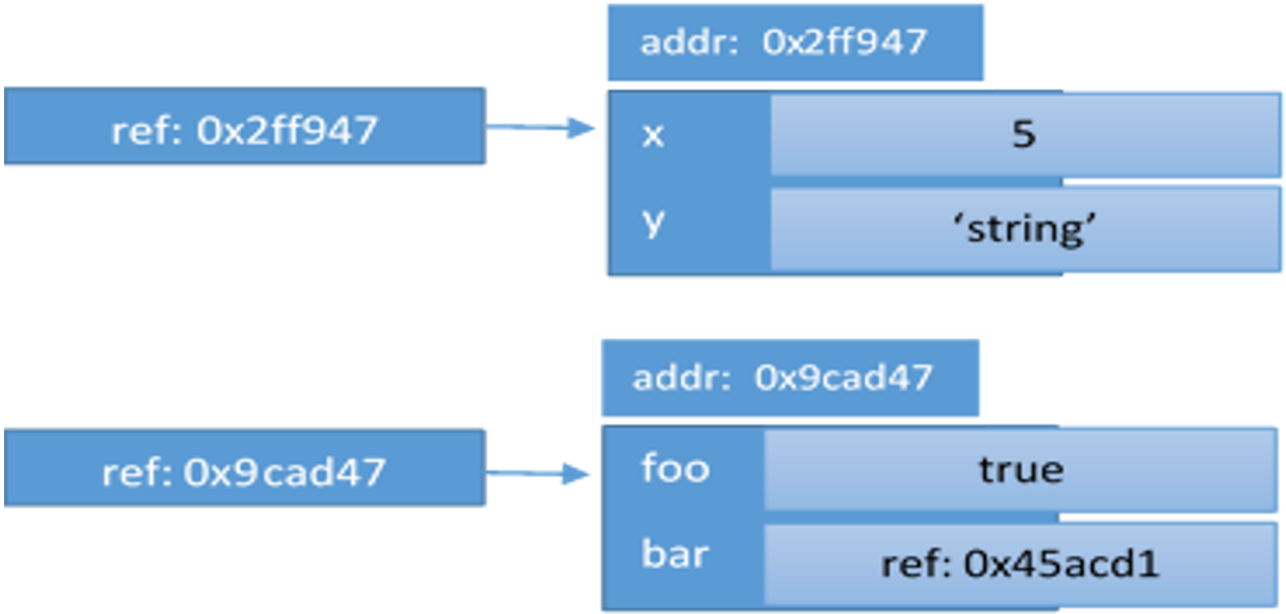


Definition

An object is a
collection of ***properties***
A property is a
named container for a ***value***
with some additional attributes

The ***name of a property*** is called ***a key***;
thus, ***an object*** can be considered as
a collection of key-value pairs.

Object


addr: 0x2ff947
ref: 0x2ff947
x
5
y
'string'
addr: 0x9cad47
ref: 0x9cad47
foo
true
bar
ref: 0x45acd1

Object properties

properties

name and value

Attributes of the property

writable, enumerable(for/in), configurable

Attributes of the object

Attributes of the object

prototype

class

extensible

Way to Create JavaScript Objects

1- Literals Objects.
2- Constructor Functions.

Literals Objects

This is the basic and first type of JavaScript objects. This format goes like JSON. Here you will create the object using two braces {} and their method and properties all are public. Literal object is a single object for all cases, even if you store it in different variables but all of them point to the same object.

```
var my_var = {
  init: function (post) {
    this.post = post;
   }
};
var my_var2 = my_var;
```

Using Constructor Functions:

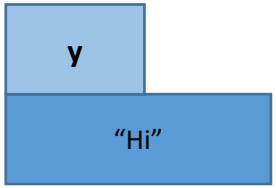
We will use functions to define a prototype and it can be self-executable also can use local values, but in a changeable context. It declared and implemented within other functions, and passed as arguments to some other functions as well.

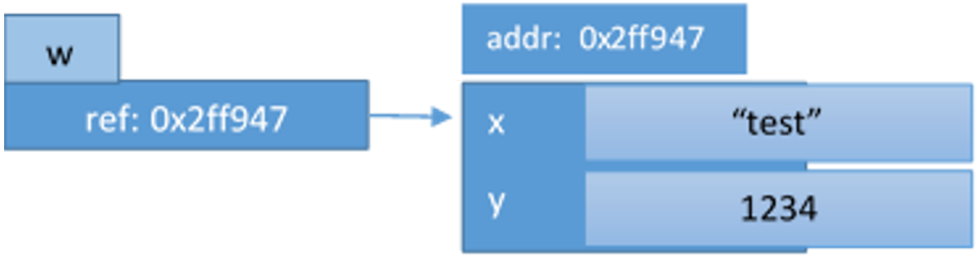
```
function Tag(name) {
  this.name = name;
}
var tag = new Tag('Post');
```

Create object

```
let date = new Date();
let arr = new Array();
let obj = new Object();
let obj = {};
let obj = Object.create({x:1, y:2});
let obj = Object.create(null);
let obj = Object.create(Object.prototype);
```

A “*variable*” vs a “*property*” in an object






```
// Value containers
var y = “Hi";
var w = {
  x: "test",
  y: 1234
};
```

```
// To get the values
y; // “Hi!"
w; // (the object ref)
w.x; // "test"
w['x']; // "test"
w.y; // 1234
w["y"]; // 1234
```

Object Initialiser (Object Literal)

```
var w = {
  x: "test",
  y: 1234,
  z: {},
  w: {},
  "": "hi"
};
```

```
var w = new Object();
w.x = "test";
w.y = 1234;
w.z = new Object();
w.w = new Object();
w[""] = "hi";
```

The code on the left-hand side has exactly the same result as the one on the right-hand side

property

```
var x=1, y=1;
var obj = {
  x: x,
  y: y
};
```

ES6

```
let x=1, y=1;
let obj = {x, y};
```

```
var person = {};
person.age = 25;
var key = 'age';

alert( person[key] ); // output: 25
```

Computed property names

```
var obj = {};
obj["F"+"N"]="Max";
console.log(obj["F"+"N"]); // "Max"
```

ES6

```
var obj1= {["First"+"Name"]:"Olga"};
console.log(obj1["First"+"Name"]); // "Olga"
```

Object

```
var obj = {name:"Dima"};

var key = "email";
obj[key] = "dmitry@m.com";

obj.getName = function() {return this.name;}
obj.getName(); // "Dima"

obj.42= "forty-two"; // SyntaxError
obj["42"] = "forty-two"; // Correct way
```

this

```
var x = 10;
console.log(
  x,       // 10
  this.x,  // 10
  window.x // 10
);
```

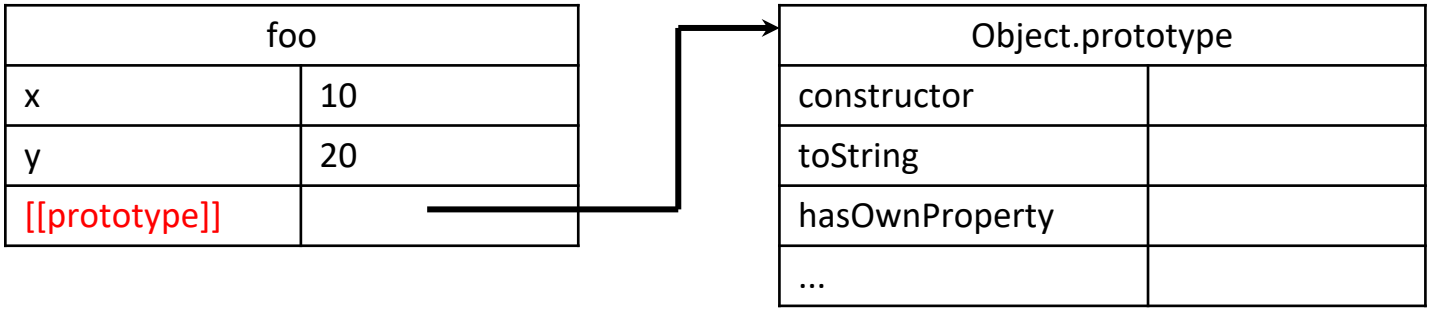
```
var user = {
  name : 'John',
  lastName : 'Smith',
  getFullName: function() {
    console.log(this.name + ' ' + this.lastName);
    console.log(user.name + ' ' + user.lastName);
  }
}

user.getFullName()  // "John Smith"
                    // "John Smith"
```

example

```
var calculator = {
 operand1: 1,
 operand2: 1,
 add: function() {
  this.result = this.operand1 + this.operand2;
}};
calculator.add();
console.log(calculator.result);  // 2
```

[[prototype]] or __proto__

```
var foo = {
    x: 10,
    y: 20
};
```



__proto__

ES5

```
var x={x:12};
var y = Object.create(x,{y:{value:13}});
console.log(y.x);  // 12
console.log(y.y);  // 13
```

ES6

```
let a = {a: 12, __proto__:{b:13}}
console.log(a.a);  // 12
console.log(a.b);  // 13
```

[[prototype]] or __proto__

```
Object.prototype.x = 10;

var a = {};

alert(a["x"]); // 10
alert(a.toString);
alert(a.constructor);
```

```
var x={x:12};
var y ={b:15};
y.__proto__=x;
```

```
var emptyHash = Object.create(null);
console.log(emptyHash.toString());
// TypeError: emptyHash.toString is not a function
```

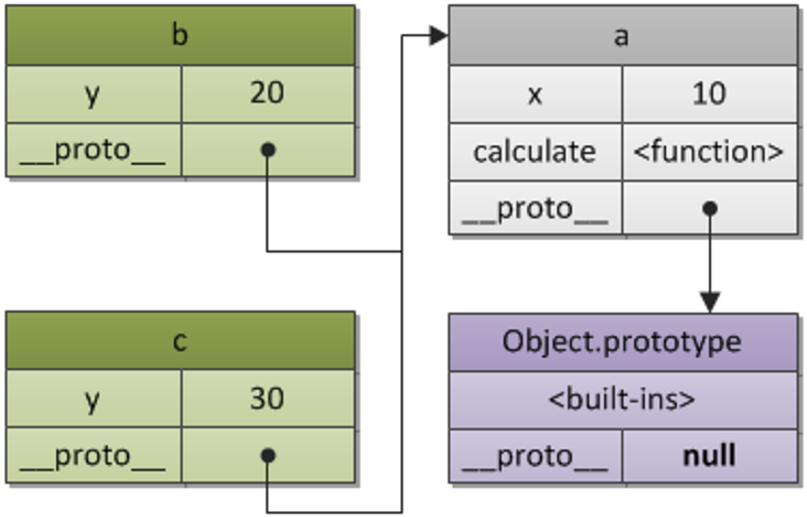
Object.setPrototypeOf()

```
let a= {a:12};
let b = {b:15};
Object.setPrototypeOf(b,a);
```

The **Object.setPrototypeOf()** method sets the prototype (i.e., the internal [[Prototype]] property) of a specified object to another object or null.

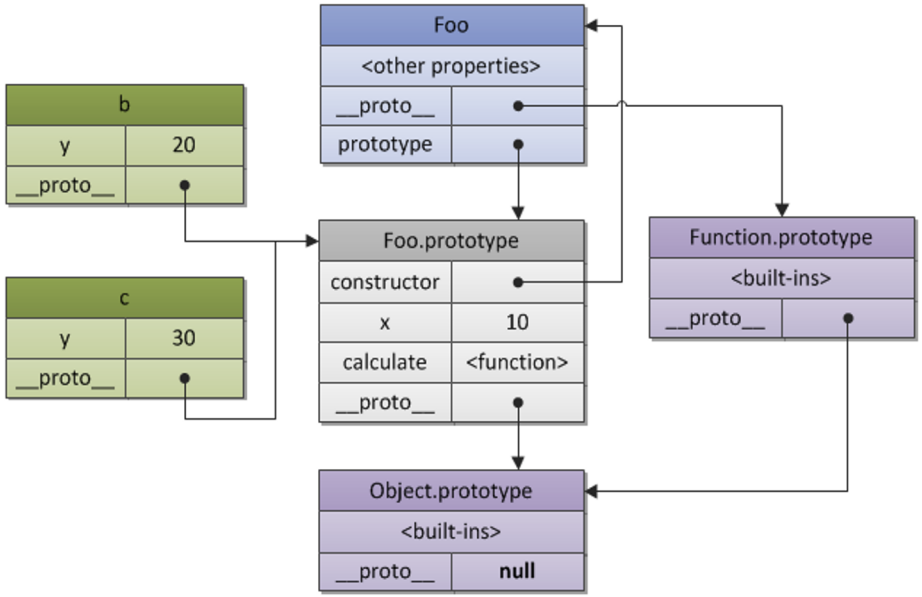
prototype chain

```
var a = {
  x: 10,
  calculate: function (z) {
    return this.x + this.y + z;
  }
};

var b = {
  y: 20,
  __proto__: a
};

var c = {
  y: 30,
  __proto__: a
};

b.calculate(30); // 60
c.calculate(40); // 80
```



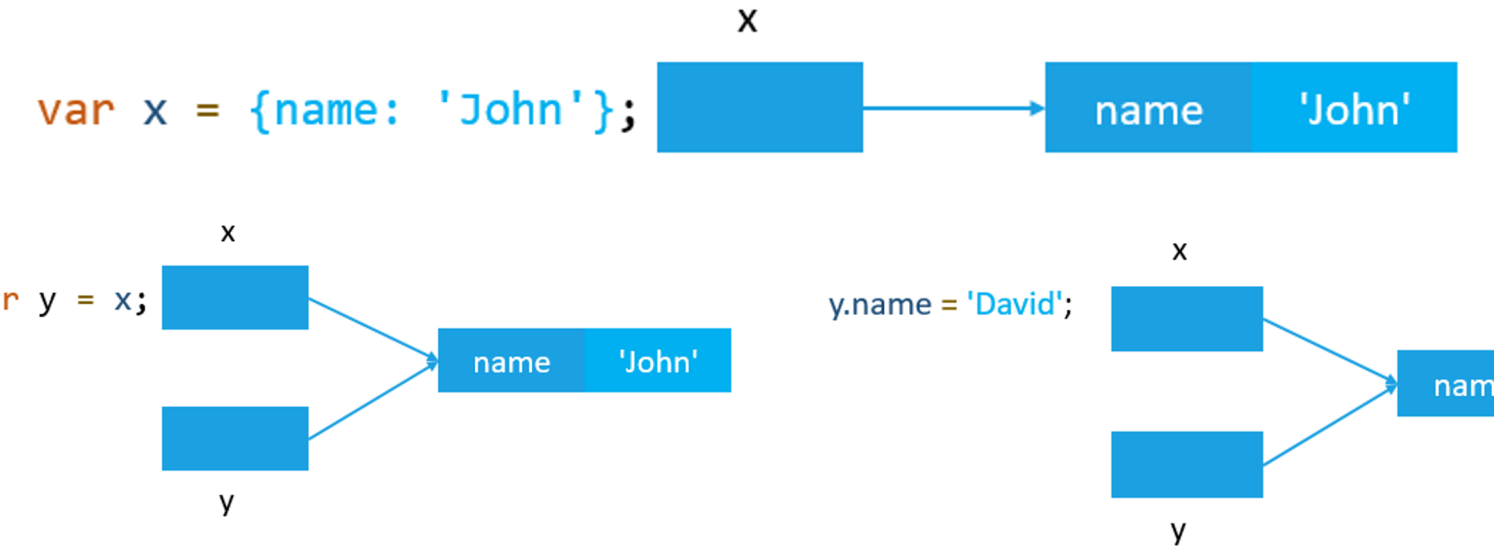
Constructor

```
function Foo(y) {
  this.y = y;
}

Foo.prototype.x = 10;
Foo.prototype.calculate =
  function (z) {
    return this.x + this.y + z;
  };

var b = new Foo(20);
var c = new Foo(30);
b.calculate(30); // 60
c.calculate(40); // 80
```



Copying by reference

One of the fundamental differences of objects vs primitives is that they are stored and copied “by reference”.



Const object

An object declared as `const` *can* be changed.

```
const user = {
  name: "John"
};
user.age = 25;
alert(user.age); // 25
```

```
const car = {color: 'red'};
const new_car = car;
delete new_car.color;
alert(car.color); // undefined
```

Object.assign()

```
var o1 = { a: 1 };
var o2 = { b: 2 };
var o3 = { c: 3 };
var obj = Object.assign(o1, o2, o3);
console.log(obj);                    //  {a: 1, b: 2, c: 3}
console.log(o1);                     //  {a: 1, b: 2, c: 3}
```

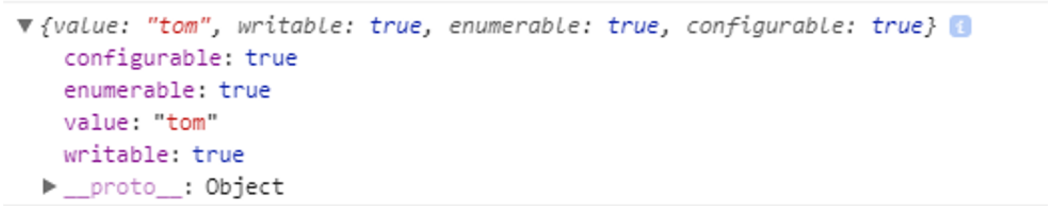
method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Object Property Descriptors

The Object.getOwnPropertyDescriptor() method returns a property descriptor for an own property (that is, one directly present on an object and not in the object's prototype chain) of a given object.

```
let cat={
  name: 'tom',
  age:5
};
```

```
let a = Object.getOwnPropertyDescriptor(cat, 'name');
console.log(a);
```



The “for...in” loop

To walk over all keys of an object, there exists a special form of the loop: `for..in`.

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for(let key in user) {

  alert( key );

  alert( user[key] );
}
```

```
// keys
 // name, age, isAdmin
 // values for the keys
 // John, 30, true
```

summary

Objects are associative arrays with several special features.

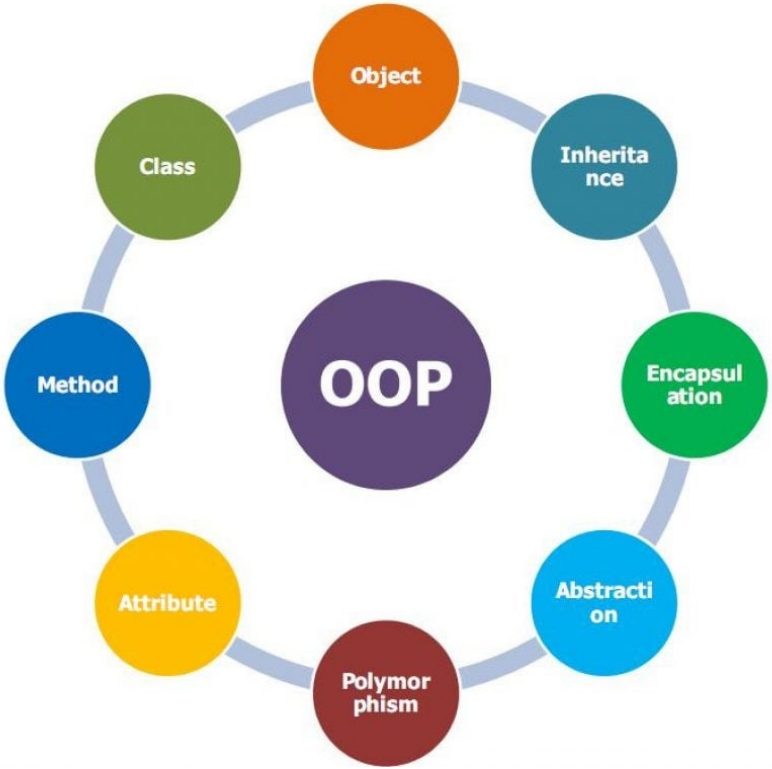
Property keys must be strings or symbols (usually strings).
Values can be of any type.

The dot notation: obj.property.
Square brackets notation obj["property"].

To delete a property: delete obj.prop.
To check if a property with the given key exists: "key" in obj.
To iterate over an object: for(let key in obj) loop.
Objects are assigned and copied by reference.

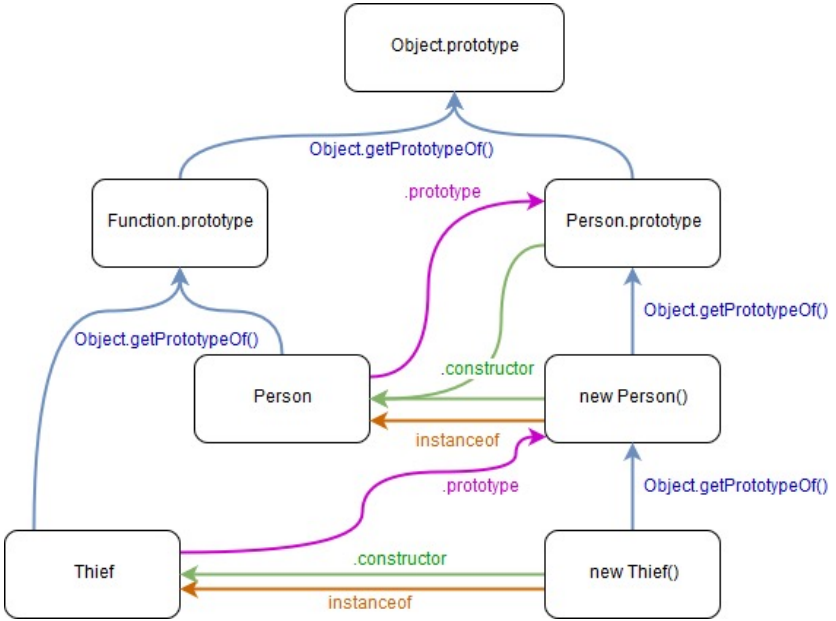
Object-oriented programming

- Encapsulation
- Composition,
  Inheritance, and delegation
- Polymorphism
- Abstraction

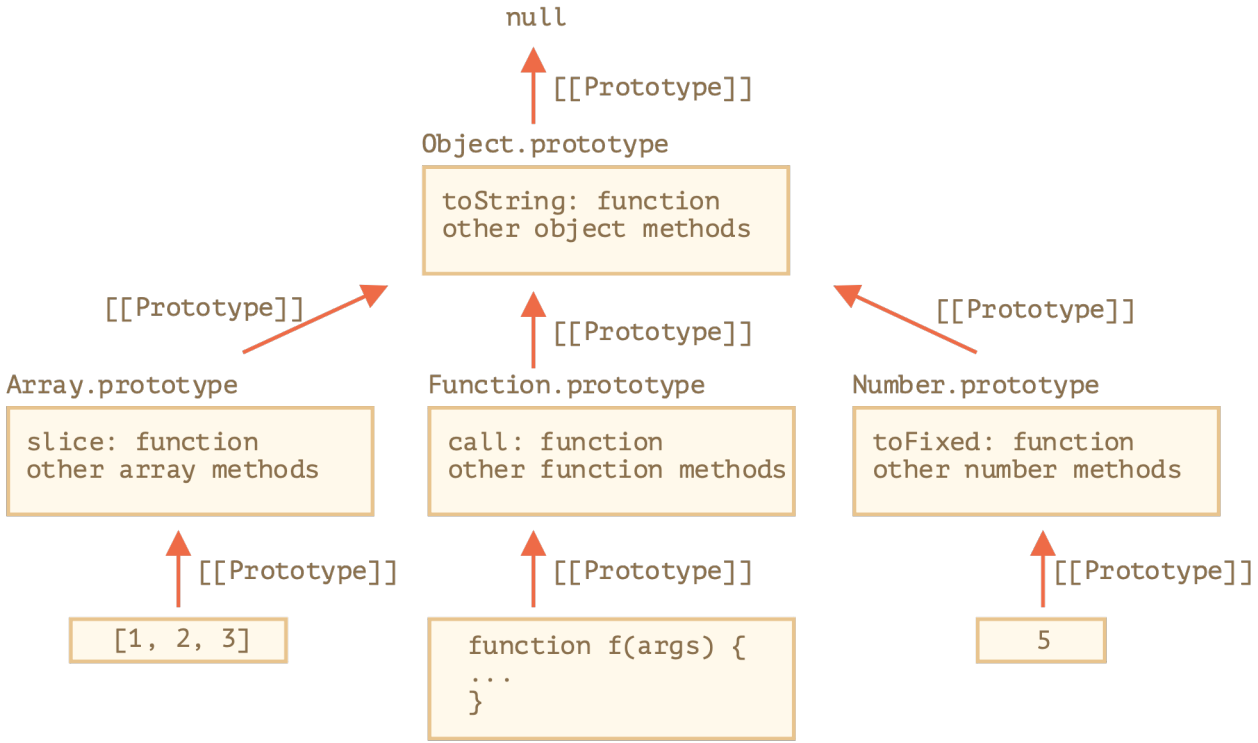


Object-oriented programming. Prototypes

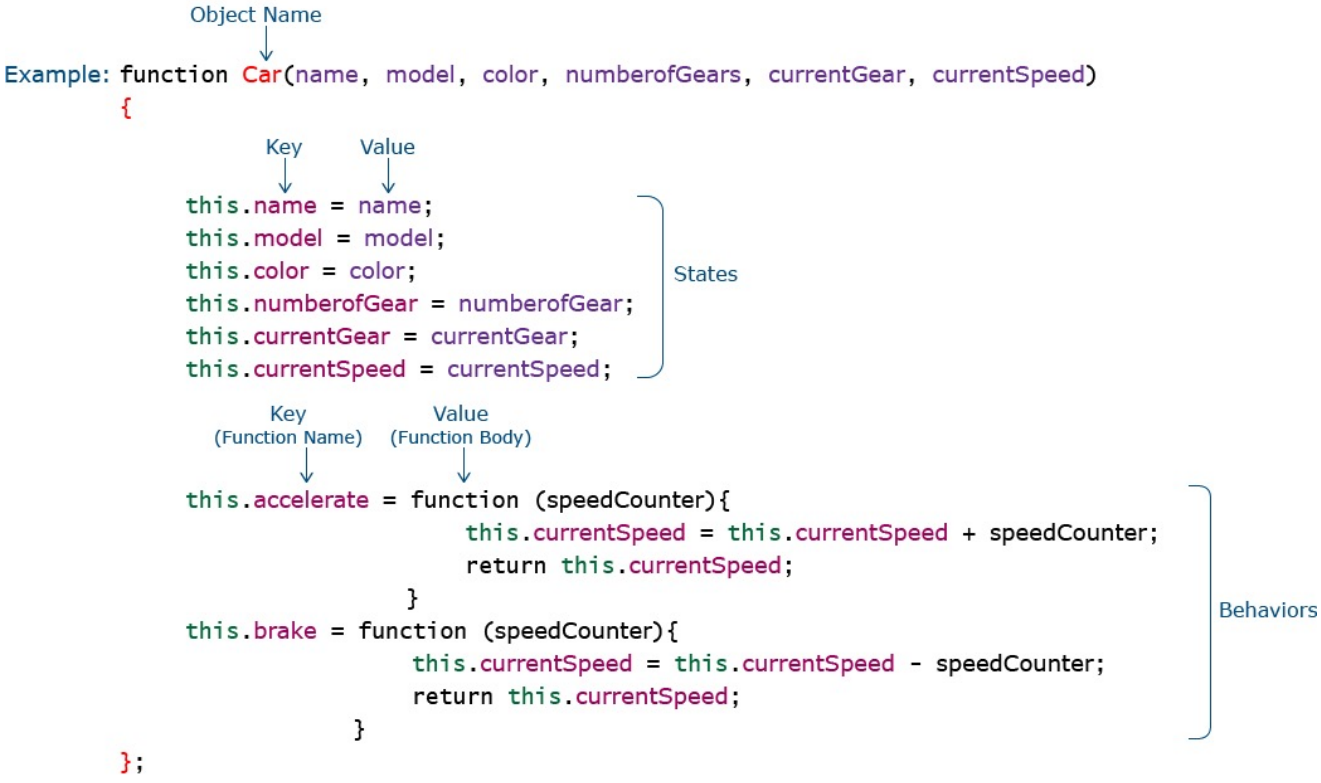
Prototype-based programming is a style of object-oriented programming in which behavior reuse (known as inheritance) is performed via a process of reusing existing objects that serve as prototypes. This model can also be known as *prototypal*, *prototype-oriented, classless*, or *instance-based* programming.
Prototype-based programming uses generalized objects, which can then be cloned and extended.



Prototypes


null
[[Prototype]]
Object.prototype
toString: function
other object methods
[[Prototype]]
[[Prototype]]
[[Prototype]]
Array.prototype
slice: function
other array methods
Function.prototype
call: function
other function methods
Number.prototype
toFixed: function
other number methods
[[Prototype]]
[[Prototype]]
[[Prototype]]
[1, 2, 3]
function f(args) {
...
}
5

Prototypes. Function constructor.



Prototypes. Function constructor.

```
> function Vehicle(make, model, color) {
      this.make = make,
          this.model = model,
          this.color = color,
          this.getName = function () {
              return this.make + " " + this.model;
          }
  }
<- undefined
> var car = new Vehicle("Mercedies","CLK200","Black")
<- undefined
> car
<- ▼ Vehicle {make: "Mercedies", model: "CLK200", color: "Black", getName: f}
     color: "Black"
   ▶ getName: f ()
     make: "Mercedies"
     model: "CLK200"
   ▶ __proto__: Object
>
```

ES6 Classes

classes

```
class Name [extends Parent]  {
  constructor
  methods
}
```

classes

One way to define a class is using a **class declaration**.

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

function declarations are **hoisted**
class declarations **are not**

```
var p = new Rectangle(); //
ReferenceError

class Rectangle {}
```

class expressions

A **class expression** is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body.

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

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

constructor

The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class.
A constructor can use the super keyword to call the constructor of a parent class.

```
class Task {
  constructor() {
     console.log("create instance task!");
  }
  static loadAll() {
   console.log("load all tasks...");
   }
}
```

Static methods

The static keyword defines a static method for a class.

Static methods are called without instantiating their class and **cannot** be called through a class instance.

Static methods are often used to create utility functions for an application.

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.hypot(dx, dy);
  }
}
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));

//   7.0710678118654755
```

Sub classing with extends

The extends keyword is used in class declarations or class expressions to create a class as a child of another class

```
class Child extends Parent {
  ...
}
```

The extends keyword can be used to subclass custom classes as well as built-in objects. The .prototype of the extension must be an Object or null.

```
class myDate extends Date
{
  constructor() {
  super();
 }
}
```

```
class nullExtends extends null {
  constructor() {}
 }
Object.getPrototypeOf(nullExtends);
Object.getPrototypeOf(nullExtends.prototype)
new nullExtends();
```

Overriding a method

```
class Child extends Parent {
  stop() {    // ...this will be used for Child.stop()
  }
}
```

Classes provide "super" keyword for that.
super.method(...) to call a parent method.
super(...) to call a parent constructor (inside our constructor only).

Arrow functions have no super

```
class Child extends Parent {
  stop() {
    setTimeout(() => super.stop(), 1000); // call parent stop
after 1sec
  }
}
```

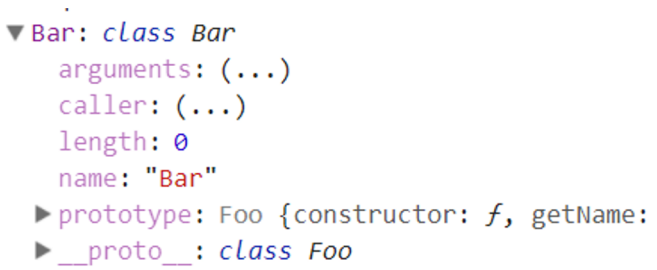
ES6 Classes

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

  class Square extends Polygon {
    constructor(sideLength) {
      super(sideLength, sideLength);
    }
    area() {
      return this.height * this.width;
    }
    sideLength(newLength) {
      this.height = newLength;
      this.width = newLength;
    }
  }

  var square = new Square(2);
<· undefined
> square
<· ▼Square {height: 2, width: 2}
     height: 2
     width: 2
   ▼__proto__: Polygon
     ▶area: ƒ area()
     ▶constructor: class Square
     ▶sideLength: ƒ sideLength(newLength)
     ▶__proto__: Object
```

```
class Car {
 constructor(brand) {
  this.carname = brand;
 }
 present() {
  return 'I have a ' + this.carname;
 }
}

class Model extends Car {
 constructor(brand, mod) {
  super(brand);
  this.model = mod;
 }
 show() {
  return this.present() + ', it is a ' + this.model;
 }
}

mycar = new Model("Ford", "Mustang");
console.log( mycar.show());  // "I have a Ford, it is a Mustang"
```

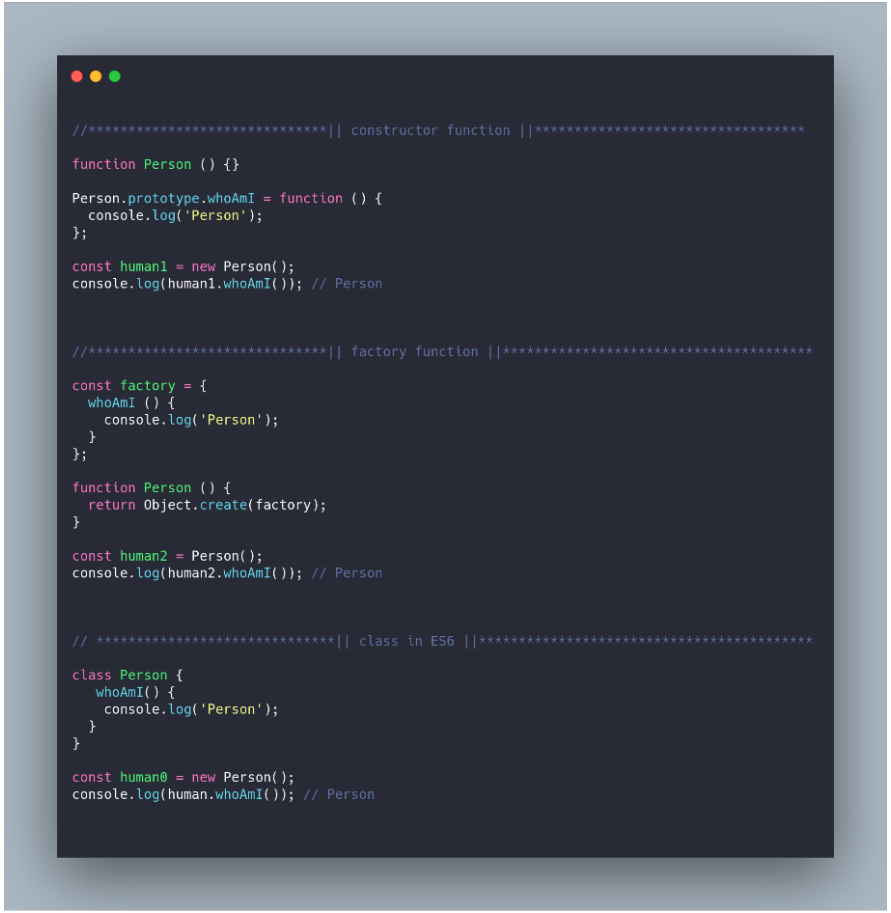
Example

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

getName() {
    return this._name;
  }
}
class Bar extends Foo {
  getName() {
    return super.getName() + ' Doe';
  }
}
var bar = new Bar('John');
console.log(bar.getName()); // John Doe
```



OOP styles

- Constructor function – old-school style
- Factory function - alternative style, more scaleble but require additional coding skiils
- Class in ES6 – new-school style



```
//*****************************|| constructor function ||*********************************

function Person () {}

Person.prototype.whoAmI = function () {
  console.log('Person');
};

const human1 = new Person();
console.log(human1.whoAmI()); // Person



//*****************************|| factory function ||************************************

const factory = {
  whoAmI () {
    console.log('Person');
  }
};

function Person () {
  return Object.create(factory);
}

const human2 = Person();
console.log(human2.whoAmI()); // Person



// *****************************|| class in ES6 ||*************************************

class Person {
   whoAmI() {
    console.log('Person');
  }
}

const human0 = new Person();
console.log(human.whoAmI()); // Person
```

JSON

JavaScript Object Notation
is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format, with a diverse range of applications, such as serving as a replacement for XML in AJAX systems.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON comparison with other formats

YAML

YAML version 1.2 is a superset of JSON; prior versions were not strictly compatible. For example, escaping a slash (/) with a backslash (\) is valid in JSON, but was not valid in YAML. Such escaping is common practice when injecting JSON into HTML to protect against cross-site scripting attacks.
YAML supports comments, while JSON does not

```
first name: John
last name: Smith
age: 25
address:
        street address: 21 2nd Street
        city: New York
        state: NY
        postal code: '10021'
phone numbers:
        - type: home
          number: 212 555-1234
        - type: fax
          number: 646 555-4567
sex:
        type: male
```

JSON comparison with other formats

XML

Note that the XML examples below don't encode the data type (e.g. that age is a number), and would need something like a schema to encode the same information as the JSON example above does as is. Note that the tag names are in camelCase, since XML tag names cannot have spaces in them (in contrast to JSON and YAML keys), although many programmers use Snake_case to overcome this limitation (i.e. <ken_block>)

```
<person firstName="John" lastName="Smith" age="25">      <address
streetAddress="21 2nd Street" city="New York" state="NY" postalCode="10021" />
          <phoneNumbers>
                    <phoneNumber type="home" number="212 555-1234"/>
                    <phoneNumber type="fax" number="646 555-4567"/>
          </phoneNumbers>
          <sex type="male"/>
</person>
```

Homework

YOU SHOULD CREATE FIGURE BUILDER FABRIC
THAT CAN OUTPUT A CERTAIN FIGURE CLASS BASED ON INPUT

1. Create class hierarchy with base class Figure
2. Create a few classes that will be inherit from Figura and have more specific constructors
3. Each inherited class should redefine methods of finding square
4. Create Figure Builder function
5. Test it with different variations of inputs

FE Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

serhii_shcherbak@epam.com

With the note [FE Online UA Training Course Feedback]

Thank you.

Q&A

UA Frontend Online LAB