

JavaScript

OBJECT



Object

Undefined e Null

Comparação de Objetos

Herança

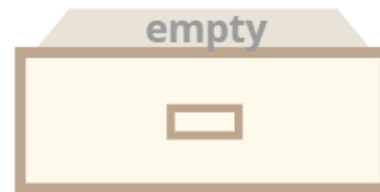
Object API

Object

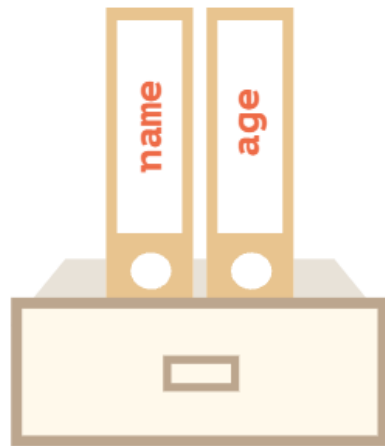
Um **objeto** é uma **coleção dinâmica** de **propriedades** definidas por **chaves**, que podem ser do tipo String ou Symbol, e valores que podem ser de qualquer tipo de dado



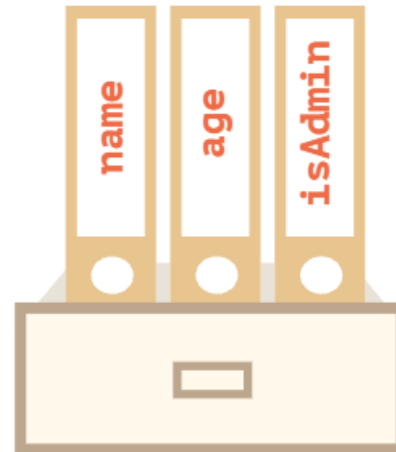
user →



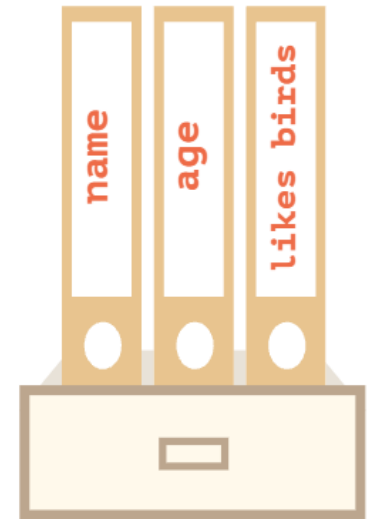
user →



user →



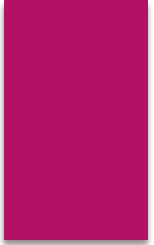
user →



```
const { name } = user;
```

```
const user = {  
  'name': 'Alex',  
  'address': '15th Park Avenue',  
  'age': 43  
}
```

Chave	Valor
title	Javascript Assertivo
author	Gabriel Ramos
pages	386
language	Português
available	true



É possível criar objetos de várias formas: pela **notação literal**, por meio de uma **função construtora** ou do **método create da Object API**

```
// MANEIRAS DE CRIAR OBJETOS
```

```
{}; // Através da notação Literal (mais usual)
```

```
new Object(); // Através da Função Construtora
```

```
Object.create(null); // Através da Object API  
// Protótipos (vamos ver adiante)
```

Object - notação literal

```
// MANEIRA LITERAL
```

```
book = {  
  title: "Javascript Assertivo",  
  author: "Gabriel Ramos",  
  pages: 386,  
  language: "Portugues",  
  available: true  
};  
// atente ao detalhe, virgulas entre chave-valor  
// A última linha não tem virgula  
// sempre feche com ponto-virgula  
// As chaves podem ser de qualquer tipo de dados  
console.log(book);  
// Mostre o objeto criado
```

Uma das diversas
maneiras de atribuir
propriedades a um objeto
é durante a sua
inicialização, pela
notação literal

Shorthand Notation

// - ES6: Passando variáveis para o Objeto

```
const title = "Javascript Assertivo";  
const author = "Gabriel Ramos";  
const pages = 386;  
const language = "Portugues";  
const available = true;
```

```
const book = {  
  title,  
  author,  
  pages,  
  language,  
  available  
};
```

```
console.log(book);
```


Dependendo da chave é necessário declará-la **diretamente como String**. Respeite os nomes das chaves, como se fossem **variáveis**

```
// Respeite os nomes das chaves  
// como se fossem variáveis
```

```
const book = {  
  title: "Javascript Assertivo",  
  author: "Gabriel Ramos",  
  number-of-pages: 386, // NÃO FUNCIONA  
  "number-of-pages": 386, // MANEIRA CORRETA  
  language: "Portugues",  
  available: true  
};  
console.log(book);
```

Exceções:

```
const excecao = {  
  10: "Aceita",  
  0xff: "Óia",  
  dwa: "Javascript"  
};  
  
console.log(excecao);
```

Também é possível computar as **chaves** em tempo de execução

```
// COMPUTANDO CHAVES - outras possibilidades
```

```
const chave1 = "title";  
const chave2 = "author";  
const chave3 = "pages";  
const chave4 = "language";  
const chave5 = "available";
```

```
const bookc = {  
  [chave1]: "Javascript Assertivo",  
  [chave2]: "Gabriel Ramos",  
  [chave3]: 386,  
  [chave4]: "Portugues",  
  [chave5]: true };
```

```
console.log(bookc);
```

Object

Além da notação literal, é possível atribuir propriedades aos objetos **por meio da sua referência**

// Atribuição por meio de referencia

```
const book = {};  
book.title = "Javascript Assertivo";  
book.author = "Gabriel Ramos"  
book.pages = 386;  
book.language = "Portugues";  
book.available = true;
```

```
console.log(book);
```

// Funciona com função construtora
// e Object API

```
const book = new Object();
```

```
const book = Object.create(null);
```

Object computar chaves por referência

```
// computar chaves por referência
const key1 = "title";
const key2 = "author";
const key3 = "pages";
const key4 = "language";
const key5 = "available";

const book = {};

book[key1] = "Javascript Assertivo",
book[key2] = "Gabriel Ramos",
book[key3] = 386,
book[key4] = "Portugues",
book[key5] = true

console.log(book);
```

Assim como na notação literal, é possível computar as chaves de um objeto em tempo de execução por meio da sua referência

útil quando não sabemos os nomes das **propriedades** ex: book.title

Cada uma das propriedades de um objeto podem ser consultadas por meio da sua referência, **de forma direta**

```
// propriedades consultadas por  
// referênciade forma direta
```

```
const book = {  
  title: "Javascript Assertivo",  
  author: "Gabriel Ramos",  
  pages: 386,  
  language: "Portugues",  
  available: true };
```

```
console.log(book.title);  
console.log(book.author);  
console.log(book.pages);  
console.log(book.language);  
console.log(book.available);
```

É possível consultar cada uma das propriedades de um objeto por meio da computação das chaves

```
// propriedades consultadas por
// computação das chaves

const book = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};

for (let key in book) {
  console.log(key); // propriedades
  //console.log(book[key]); //valores
}
```

```
// copiando valores e propriedades
// book1 para book2
```

```
const book1 = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};

const book2 = {};
for (let key in book1) {
  book2[key] = book1[key];
}

console.log(book2);
```

Fantástico JS! Percorrendo propriedades e valores do objeto com for

Object computar chaves por referência

```
// computar chaves por referência
const key1 = "title";
const key2 = "author";
const key3 = "pages";
const key4 = "language";
const key5 = "available";

const book = {};

book[key1] = "Javascript Assertivo",
book[key2] = "Gabriel Ramos",
book[key3] = 386,
book[key4] = "Portugues",
book[key5] = true

console.log(book);
```

Assim como na notação literal, é possível computar as chaves de um objeto em tempo de execução por meio da sua referência

útil quando não sabemos os nomes das **propriedades** ex: book.title

Object Undefined e Null

O tipo **undefined** é retornado caso a chave não seja encontrada

Qual é a diferença entre os tipos **undefined** e **null**?

É possível consultar uma determinada chave por meio do operador **in**

O tipo **undefined** é retornado caso a chave não seja encontrada

```
// undefined
const book = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};
```

```
console.log(book.publisher);
```

```
// undefined - Propriedade sequer existe
// null - Ausência de valor
```

undefined

Propriedade sequer existe

null

Ausência de valor

É possível consultar uma determinada chave por meio do operador **in**

```
// undefined
const book = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};
console.log("title" in book);
console.log("author" in book);
console.log("pages" in book);
console.log("language" in book);
console.log("available" in book);
console.log("publisher" in book);
// Neste caso não existe a propriedade publisher
```

Operador in
(chave **in** objeto) – Pode ser utilizado para consultar propriedade antes de usar

Não atribua para **undefined** ou **null** com intenção de **apagar uma propriedade**

```
// undefined
const book = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};

book.available = undefined;
// muda apenas o valor

console.log(book);
console.log("available" in book);
// a propriedade continua existindo
```

```
// null, o mesmo resultado
const book = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};

book.available = null;
// muda apenas o valor tbm

console.log(book);
console.log("available" in book);
// a propriedade continua existindo
```

As propriedades de um objeto podem ser apagadas por meio do operador **delete**

```
// delete
const book = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos",
  pages: 386,
  language: "Portugues",
  available: true
};
```

```
delete book.available;
console.log(book);
console.log("available" in book);
```

```
// Adicionar uma propriedade a um objeto
// A sintaxe é: object.property = value
```

```
// por referencia direta
book.publisher = 'Casa do código';
```

```
//computação de chaves por referencia
book['publisher'] = 'Casa do código';
```

Object Comparação de Objetos

A comparação dos objetos é feita por meio da sua referência, assim, **ainda que dois objetos tenham exatamente as mesmas propriedades eles serão considerados diferentes**

```
// comparação entre dois objetos
const book1 = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos"
};
const book2 = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos"
};
console.log(book1 == book2); // false
console.log(book1 === book2); // false

// se comparar o mesmo objeto é igual
console.log(book1 == book1); // true
console.log(book2 === book2); // true
```

Uma das formas para comparar os objetos é **analisando cada uma das suas propriedades** por meio da comparação das chaves e valores

```
const book1 = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos"
};
const book2 = {
  title: "Javascript Assertivo",
  author: "Gabriel Ramos"
};
let equal = true;
for (let key in book1) {
  if (book1[key] !== book2[key])
    equal = false; }

console.log(equal);
for (let key in book2) {
  if (book2[key] !== book1[key])
    equal = false; }
console.log(equal);
```

< O código ao lado faz uma comparação superficial

Para garantir que um objeto é igual ao outro é **necessário verificar seus protótipos**

Object Herança

O principal objetivo da herança é **permitir o reuso de código** por meio do compartilhamento de propriedades entre objetos, **evitando a duplicação**

Na linguagem JavaScript a herança é realizada entre **objetos** e não classes

O JS traz herança baseada em protótipo

```
// Aqui dois objetos simples  
// com 2 propriedades em comum
```

```
const scheme = {  
  name: "Scheme",  
  year: 1975,  
  paradigm: "Functional" };
```

```
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  paradigm: "Functional" };
```

```
console.log(scheme);  
console.log(javascript);
```

```
// Aqui criamos um terceiro objeto  
// vamos ver como reaproveitar isso
```

```
const functionalLanguage = {  
  paradigm: "Functional" };
```

```
const scheme = {  
  name: "Scheme",  
  year: 1975,  
  paradigm: "Functional" };
```

```
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  paradigm: "Functional"  
};  
console.log(functionalLanguage);  
console.log(scheme);  
console.log(javascript);
```


A propriedade `__proto__` é uma referência para o **protótipo** do objeto

```
// PROTÓTIPO
const functionalLanguage = {
  paradigm: "Functional"
};
const scheme = {
  name: "Scheme",
  year: 1975,
  __proto__: functionalLanguage
};
const javascript = {
  name: "JavaScript",
  year: 1995,
  __proto__: functionalLanguage
};
console.log(functionalLanguage);
console.log(scheme);
console.log(javascript);
```

Porque a propriedade `paradigm` não foi exibida dentro do objeto?

O `console.log` só mostra as propriedades dentro no próprio objeto.

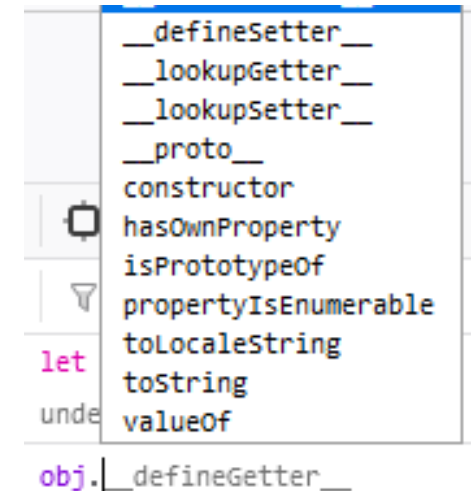
Existem mais duas maneiras de acessar protótipos, veremos adiante

Quando você consulta a propriedade de um objeto e ela **não existe**, automaticamente **o JS vai ao protótipo desse objeto** e assim sucessivamente

```
const functionalLanguage = {
  paradigm: "Functional"
};
const scheme = {
  name: "Scheme",
  year: 1975,
  __proto__: functionalLanguage
};
const javascript = {
  name: "JavaScript",
  year: 1995,
  __proto__: functionalLanguage
};
console.log(functionalLanguage);
console.log(scheme.paradigm);
console.log(javascript.paradigm);
```

Todo objeto em JS **tem protótipo**
Inclusive `__proto__`
Vamos ver `hasOwnProperty`

```
let obj = {};
obj.
```



O método `hasOwnProperty` pode ser utilizado para determinar se uma propriedade pertence ao objeto

```
const functionalLanguage = {  
  paradigm: "Functional"  
};  
const scheme = {  
  name: "Scheme",  
  year: 1975,  
  __proto__: functionalLanguage  
};  
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  __proto__: functionalLanguage  
};
```

```
// aqui podemos verificar que paradigm // faz  
// parte o objeto  
for (let key in scheme) {  
  console.log(key);  
}  
  
// aqui podemos verificar que paradigm existe  
// mas esta em um dos seus protótipos  
for (let key in scheme) {  
  console.log(key, scheme.hasOwnProperty(key));  
}
```

O método `Object.setPrototypeOf` permite a interação com o protótipo do objeto

```
const functionalLanguage = {
  paradigm: "Functional"
};
const scheme = {
  name: "Scheme",
  year: 1975,
};
Object.setPrototypeOf(scheme, functionalLanguage);

const javascript = {
  name: "JavaScript",
  year: 1995,
};
Object.setPrototypeOf(javascript, functionalLanguage);

for (let key in scheme) {
  console.log(key, scheme.hasOwnProperty(key));
}
```

Por questões de clareza
no código, utilize
`Object.setPrototypeOf`

Com o método Object.create é possível criar um objeto passando o seu protótipo por parâmetro

```
// No Object.create já declaramos o protótipo
// na criação do objeto
const functionalLanguage = {
  paradigm: "Functional"
};
const scheme = Object.create(functionalLanguage);
scheme.name = "Scheme";
scheme.year = 1975;
const javascript = Object.create(functionalLanguage);
javascript.name = "JavaScript";
javascript.year = 1995;

for (let key in scheme) {
  console.log(key, scheme.hasOwnProperty(key));
}
```

CUIDADO: Sem o seu protótipo o objeto perde algumas operações importantes

```
// Cuidado com Object.create SEM PROTÓTIPO
// SE PASSAR NULL ACABA A CADEIA DE PROTÓTIPO
const functionalLanguage = Object.create(null);
functionalLanguage.paradigm = "Functional";

const scheme = Object.create(functionalLanguage);
scheme.name = "Scheme";
scheme.year = 1975;

const javascript = Object.create(functionalLanguage);
javascript.name = "JavaScript";
javascript.year = 1995;

for (let key in scheme) {
  console.log(key, scheme.hasOwnProperty(key));
}
```

Caso a mesma propriedade exista no objeto e no seu protótipo, a propriedade do próprio objeto é retornada, fazendo sombra à propriedade do protótipo

```
// Mesma propriedade no objeto e no protótipo
const functionalLanguage = Object.create({});
functionalLanguage.paradigm = "Functional"; // AQUI no protótipo

const scheme = Object.create(functionalLanguage);
scheme.name = "Scheme";
scheme.year = 1975;

const javascript = Object.create(functionalLanguage);
javascript.name = "JavaScript";
javascript.year = 1995;
javascript.paradigm = "OO"; // E AQUI na base

for (let key in javascript) {
  console.log(key, javascript[key]);
} // mostra base
```

Mesma propriedade no objeto e no seu protótipo usando `getPrototypeOf`

//Mesma propriedade no objeto e no protótipo com `getPrototypeOf`

```
const functionalLanguage = Object.create({});  
functionalLanguage.paradigm = "Functional"; // AQUI no protótipo
```

```
const scheme = Object.create(functionalLanguage);  
scheme.name = "Scheme";  
scheme.year = 1975;
```

```
const javascript = Object.create(functionalLanguage);  
javascript.name = "JavaScript";  
javascript.year = 1995;  
javascript.paradigm = "OO"; // E AQUI na base
```

```
console.log(javascript); // mostra o objeto todo - BASE  
console.log(javascript.paradigm); // mostra paradigm - BASE  
console.log(javascript.__proto__.paradigm); // sobe um nível e acessa fl  
console.log(Object.getPrototypeOf(javascript).paradigm);  
// pega o protótipo do objeto javascript e pega a propriedade paradigm
```


Object API – Interagindo com objetos

O método **Object.assign** faz a cópia das propriedades dos objetos passados por parâmetro para o objeto alvo, que é retornado

```
// Object.assign faz a cópia das propriedades // pega tudo de source e coloca em target
```

```
const target = { a: 1, b: 2 };
```

```
const source = { b: 4, c: 5 };
```

```
const returnedTarget = Object.assign(target, source);
```

```
console.log(target); // output: Object { a: 1, b: 4, c: 5 }
```

```
console.log(returnedTarget); // output: Object { a: 1, b: 4, c: 5 }
```

Object API – Interagindo com objetos

O método **Object.Keys** retorna as chaves das propriedades do objeto

```
// Object.keys() retorna as chaves em um array
```

```
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  paradigm: "OO and Functional"  
};
```

```
console.log(Object.keys(javascript));
```

Object API – Interagindo com objetos

O método **Object.values** retorna os valores das propriedades do objeto

```
// Object.values Retorna os valores em um array
```

```
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  paradigm: "OO and Functional"  
};
```

```
console.log(Object.values(javascript));
```

Object API – Interagindo com objetos

O método **Object.entries** retorna as propriedades do objeto em pares de chave e valor

```
// Object.entries Retorna pares chave valor
```

```
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  paradigm: "OO and Functional"  
};
```

```
console.log(Object.entries(javascript));
```

Object API – Interagindo com objetos

O método **Object.is** compara dois objetos, considerando os tipos de dados, de forma similar ao operador **===**

```
// Object.is compara objetos
```

```
const javascript = {  
  name: "JavaScript",  
  year: 1995,  
  paradigm: "OO and Functional"  
};
```

```
console.log(Object.is(javascript, javascript));
```