JavaScript Basics

Objects

Objects

Objects

- Complex type stores key-value pairs

Objects

- Complex type stores key-value pairs
- An empty object (“empty cabinet”) can be created using one of two syntaxes:

Objects

- Complex type stores key-value pairs
- An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
var user=new Object();

var user={};
```

Objects

- Complex type stores key-value pairs
- An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
var user=new Object();

var user={};
```

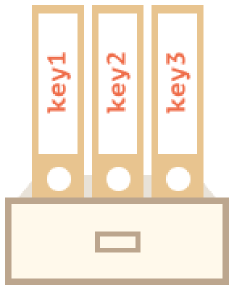
Imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key.

Objects

- Complex type stores key-value pairs
- An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
var user=new Object();

var user={};
```

Imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key.

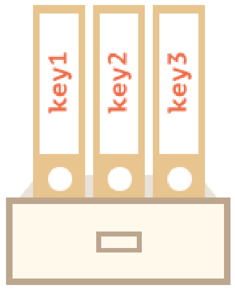


Objects

- Complex type stores key-value pairs
- An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
var user=new Object();

var user={};
```

Imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key.



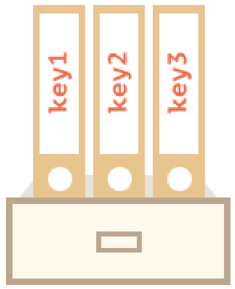
```
let user = {     // an object
  name: "John",
  age: 30
};
```

Objects

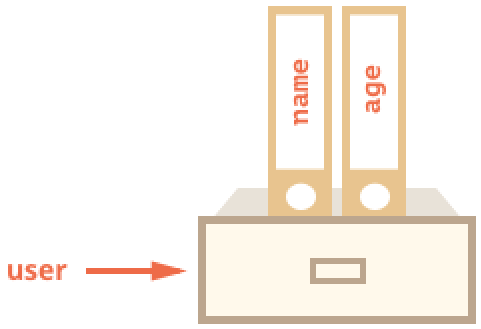
- Complex type stores key-value pairs
- An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
var user=new Object();

var user={};
```

Imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key.



```
let user = {      // an object
  name: "John",
  age: 30
};
```



Objects

Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

- You can add new property simply using dot notation.

Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

- You can add new property simply using dot notation.

```
user.isAdmin = true;
```

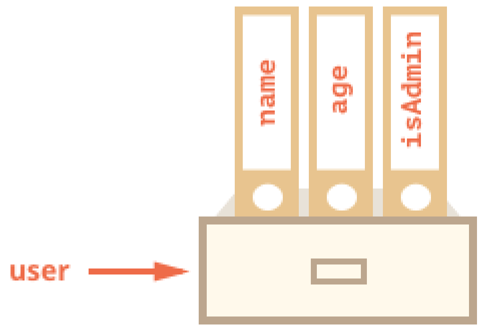
Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

- You can add new property simply using dot notation.

```
user.isAdmin = true;
```



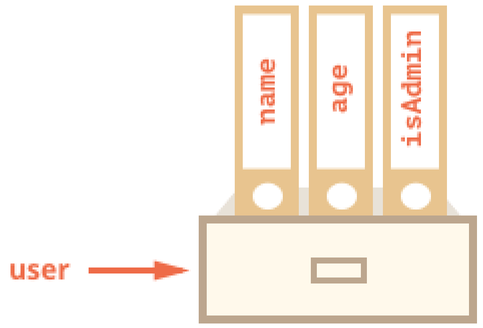
Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

- You can add new property simply using dot notation.

```
user.isAdmin = true;
```



- To remove a property, we can use delete operator:

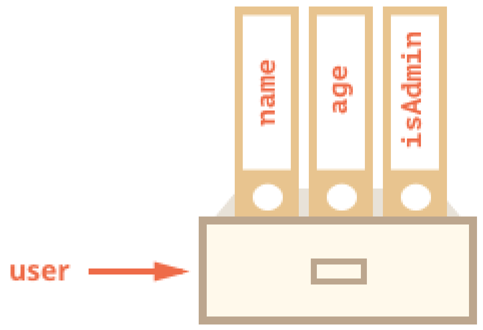
Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

- You can add new property simply using dot notation.

```
user.isAdmin = true;
```



- To remove a property, we can use delete operator:

```
delete user.age;
```

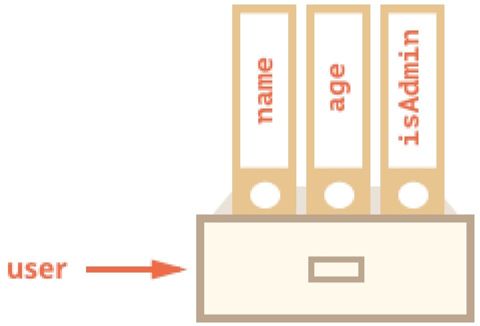
Objects

- We can add, remove and read properties from objects any time. Property values are accessible using the dot notation:

```
// get fields of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

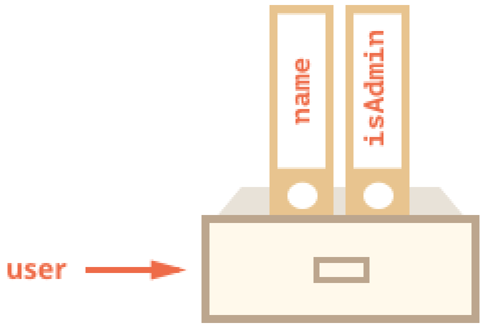
- You can add new property simply using dot notation.

```
user.isAdmin = true;
```



- To remove a property, we can use delete operator:

```
delete user.age;
```



Square Brackets

Square Brackets

For multiword properties, the dot access doesn’t work:

Square Brackets

For multiword properties, the dot access doesn’t work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square Brackets

For multiword properties, the dot access doesn’t work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square brackets also provide a way to obtain the property name as the result of any expression

Square Brackets

For multiword properties, the dot access doesn’t work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square brackets also provide a way to obtain the property name as the result of any expression

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Square Brackets

For multiword properties, the dot access doesn’t work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square brackets also provide a way to obtain the property name as the result of any expression

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Property value shorthand

Square Brackets

For multiword properties, the dot access doesn’t work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square brackets also provide a way to obtain the property name as the result of any expression

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Property value shorthand

In real code we often use
existing variables as values
for property names.

Square Brackets

For multiword properties, the dot access doesn't work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square brackets also provide a way to obtain the property name as the result of any expression

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Property value shorthand

In real code we often use existing variables as values for property names.

```
{
    name: name,
    age: age
}
```

Square Brackets

For multiword properties, the dot access doesn’t work:

```
user.like animals=false //error

user["like animals"]=true //OK
```

Square brackets also provide a way to obtain the property name as the result of any expression

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Property value shorthand

In real code we often use existing variables as values for property names.

```
{
    name: name,
    age: age
}
```

```
{
    name,
    age
}
```

Existence check

Existence check

To check existence of some property, you can use in operator :

Existence check

To check existence of some property, you can use in operator :

```
("name" in user) //true
```

Existence check

To check existence of some property, you can use in operator :

```
("name" in user) //true
```

You can also check for undefined to check existence but it fails when value stored is undefined

Existence check

To check existence of some property, you can use in operator :

```
("name" in user) //true
```

You can also check for undefined to check existence but it fails when value stored is undefined

```
(user.name === undefined)
//OK but fails when value stored is undefined
```

Existence check

To check existence of some property, you can use in operator :

```
("name" in user) //true
```

You can also check for undefined to check existence but it fails when value stored is undefined

```
(user.name === undefined)
//OK but fails when value stored is undefined
```

for..in loop

Existence check

To check existence of some property, you can use in operator :

```
("name" in user) //true
```

You can also check for undefined to check existence but it fails when value stored is undefined

```
(user.name === undefined)
//OK but fails when value stored is undefined
```

for..in loop

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

Existence check

To check existence of some property, you can use in operator :

```
("name" in user) //true
```

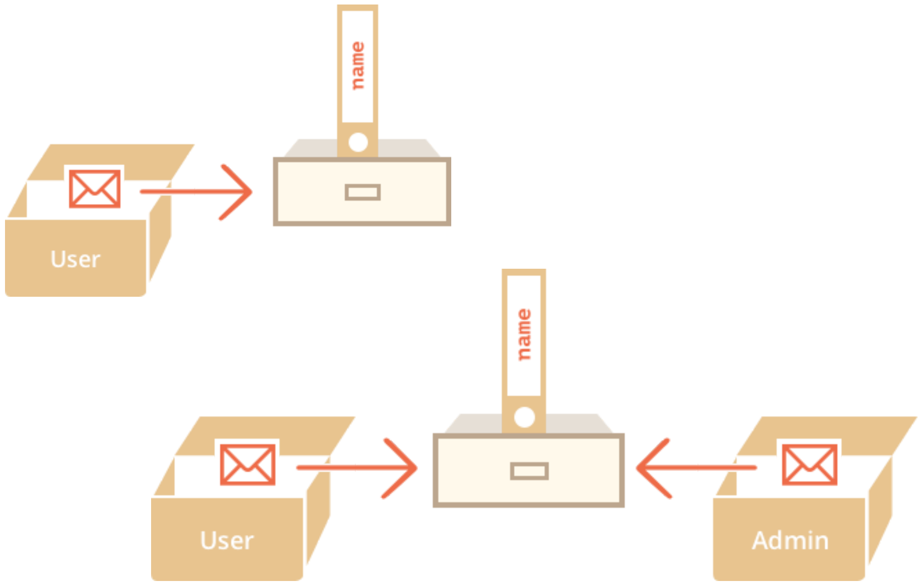
You can also check for undefined to check existence but it fails when value stored is undefined

```
(user.name === undefined)
//OK but fails when value stored is undefined
```

for..in loop

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

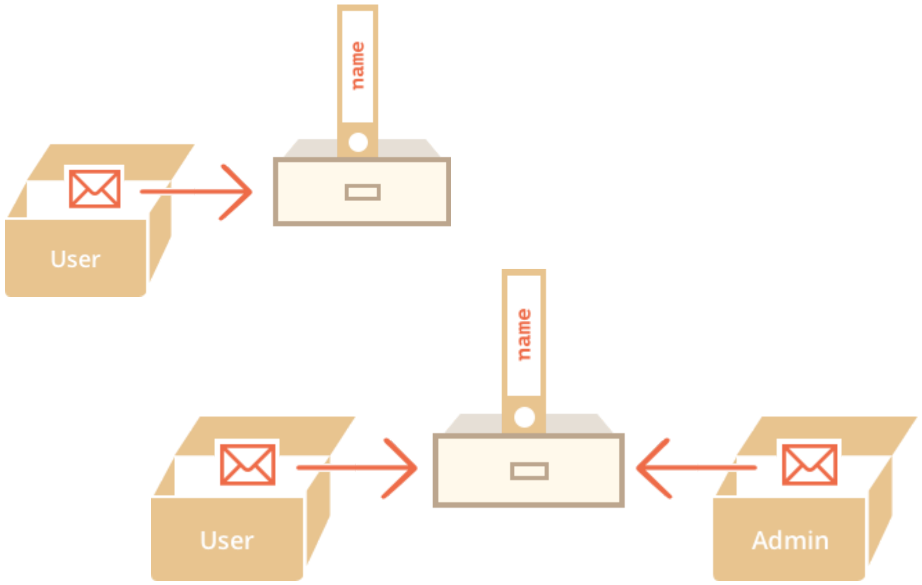
```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for(let key in user) {
  alert( key );
  alert( user[key] );
}
```



```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both vari
alert( a === b ); // true
```

```
let a = {}
let b = {}; // two independent obje

alert( a == b ); // false
```

Copying by reference



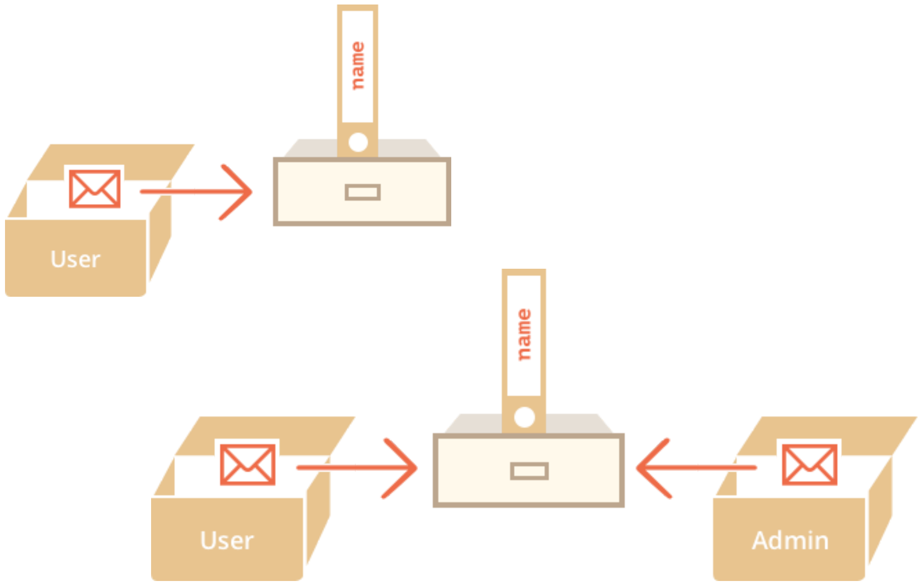
Comparision by reference

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both vari
alert( a === b ); // true
```

```
let a = {}
let b = {}; // two independent obje

alert( a == b ); // false
```

Copying by reference

Objects are copied and stored by reference unlike primitive types are copied by value



Comparision by reference

Two objects are equal only if they are the same object.

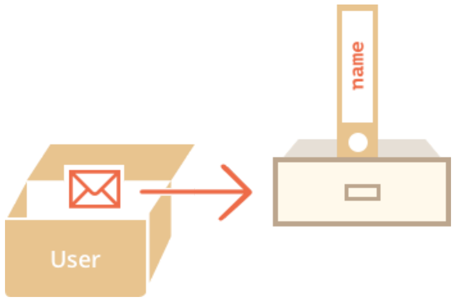
```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both vari
alert( a === b ); // true
```

```
let a = {}
let b = {}; // two independent obje

alert( a == b ); // false
```

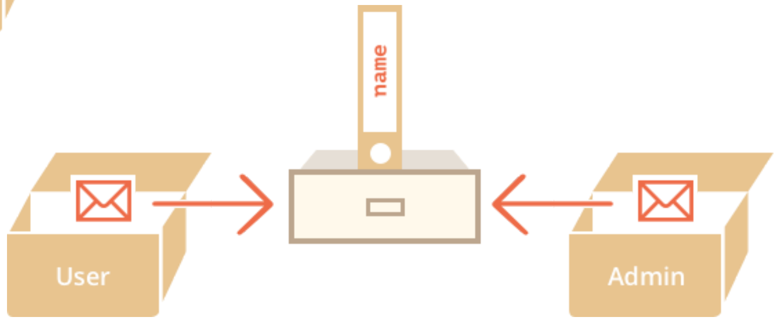
Copying by reference

Objects are copied and stored by reference unlike primitive types are copied by value

```
let user = {
  name: "John"
};
```



```
let admin = user;
```



Comparision by reference

Two objects are equal only if they are the same object.

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both vari
alert( a === b ); // true
```

```
let a = {}
let b = {}; // two independent obje

alert( a == b ); // false
```

Const object

An object declared const can be changed.

```
const user = {
  name: "John"
};

user.age = 25; //OK

//but you can't do this

user={
    name: "Varma"
} //Throws const assignment error
```

Cloning and merging

How to make duplicate copy of object instead of just reference?

We can use method Object.assign to clone an object or merge properties of other object in to object

```
//Syntex
// Object.assign(dest[, src1, src2, src3...])

let user = { name: "John" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// copies all properties from permissions1 and permissions2 into
Object.assign(user, permissions1, permissions2);

// now user = { name: "John", canView: true, canEdit: true }
```

Using Array functions on Objects

Can we use array methods like map on objects? Not directly.

For plain objects, the following methods are available

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

//Object.keys(user) = [name, age]

//Object.values(user) = ["John", 30]

//Object.entries(user) = [ ["name","John"], ["age",30] ]
```

Nested Objects