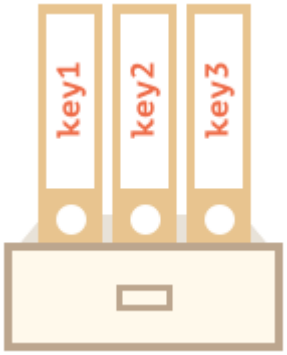
Objects

As we know from the chapter Data types, there are seven data types in JavaScript. Six of them are called “primitive”, because their values contain only a single thing (be it a string or a number or whatever).

In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

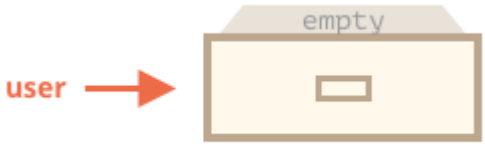
An object can be created with figure brackets {…} with an optional list of *properties*. A property is a “key: value” pair, where `key`is a string (also called a “property name”), and `value` can be anything.

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It’s easy to find a file by its name or add/remove a file.



An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```



Usually, the figure brackets {...} are used. That declaration is called an *object literal*.

Literals and properties

We can immediately put some properties into {...} as “key: value” pairs:

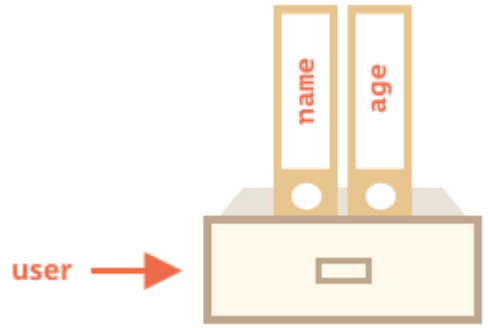
```
let user = {     // an object
 name: "John",  // by key "name" store value "John"
 age: 30        // by key "age" store value 30
};
```

A property has a key (also known as “name” or “identifier”) before the colon ":" and a value to the right of it.

In the user object, there are two properties:

1. The first property has the name "name" and the value "John".
2. The second one has the name "age" and the value 30.

The resulting user object can be imagined as a cabinet with two signed files labeled “name” and “age”.



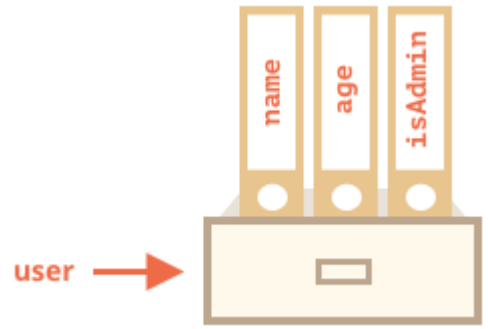
We can add, remove and read files from it any time.

Property values are accessible using the dot notation:

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

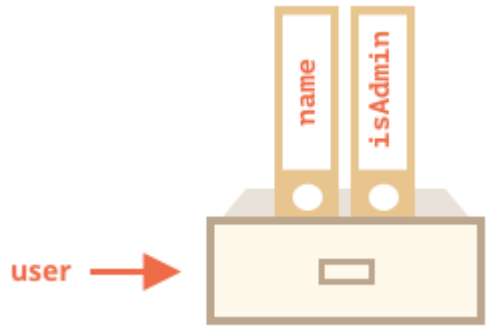
The value can be of any type. Let’s add a boolean one:

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



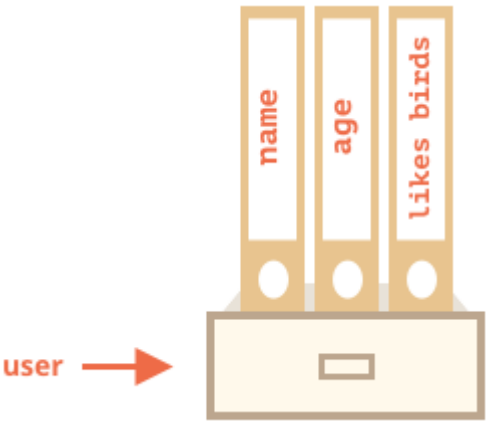
To remove a property, we can use delete operator:

```
delete user.age;
```



We can also use multiword property names, but then they must be quoted:

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // multiword property name must be quoted
};
```



The last property in the list may end with a comma:

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

That is called a “trailing” or “hanging” comma. Makes it easier to add/remove/move around properties, because all lines become alike.

Square brackets

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

```
// this would give a syntax error
user.likes birds = true
```

That's because the dot requires the key to be a valid variable identifier. That is: no spaces and other limitations.

There's an alternative "square bracket notation" that works with any string:

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Now everything is fine. Please note that the string inside the brackets is properly quoted (any type of quotes will do).

Square brackets also provide a way to obtain the property name as the result of any expression – as opposed to a literal string – like from a variable as follows:

```
let key = "likes birds";

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

Here, the variable `key` may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility. The dot notation cannot be used in a similar way.

For instance:

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

let key = prompt("What do you want to know about the user?", "name");

// access by variable
alert( user[key] ); // John (if enter "name")
```