Разработка мобильных приложений

Лекция 6

Функции

- Зачастую нам надо повторять одно и то же действие во многих частях программы.
- Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Объявление функции

```
function имя(параметры) {
  ...тело функции...
}

function showMessage() {
  alert( 'Всем привет!' );
}
```

Вызов функции

```
function showMessage() {
  alert( 'Всем привет!' );
}

showMessage();
showMessage();
```

Локальные переменные

- Переменные, объявленные внутри функции, видны только внутри этой функции.

```
function showMessage() {
  let message = "Привет, я JavaScript!";
 // локальная переменная
  alert( message );
}
showMessage(); // Привет, я JavaScript!
alert( message );
// будет ошибка, т.к. переменная видна
// только внутри функции
```

Внешние переменные

- У функции есть доступ к внешним переменным.
- Функция обладает полным доступом к внешним переменным и может изменять их значение.

```
let userName = 'Вася';
function showMessage() {
  userName = "Петя";
  // (1) изменяем значение внешней переменной
  let message = 'Привет, ' + userName;
  alert(message);
}
alert( userName ); // Вася перед вызовом функции
showMessage();
alert( userName ); // Петя, значение внешней
// переменной было изменено функцией
```

Параметры (аргументы)

- Позволяют передать внутрь функции любую информацию, используя параметры (также называемые аргументы функции).

```
function showMessage(from, text) {
// аргументы: from, text
  alert(from + ': ' + text);
}
showMessage('Аня', 'Привет!’);
// Аня: Привет!
showMessage('Аня', "Как дела?");
// Аня: Как дела?
```

- Функция всегда получает только копию значения

```
function showMessage(from, text) {
  from = '*' + from + '*’;

// немного украсим "from"
  alert( from + ': ' + text );

}
let from = "Аня";
showMessage(from, "Привет"); // *Аня*: Привет
// значение "from" осталось прежним,

// функция изменила значение локальной переменной

alert( from ); // Аня
```

Параметры по умолчанию

- Если параметр не указан, то его значением становится undefined.

- `showMessage("Аня");` Аня: undefined

```
function showMessage(from, text =
 "текст не добавлен") {
  alert( from + ": " + text );
}

showMessage("Аня");
```

Аня: текст не добавлен

Function Declaration (Объявление функции)

```
function sayHi() {
  alert( "Привет" );
}
```

Function Expression (Функциональное выражение)

```
let sayHi = function()
{
  alert( "Привет" );
};
```

- Без разницы, как определили функцию, это просто значение, хранимое в переменной.

- Смысл обоих примеров кода одинаков: "создать функцию и поместить её значение в переменную.

```
function sayHi() {
  alert( "Привет" );
}
alert( sayHi ); // выведет код функции
```

- Если после имени функции нет круглых скобок, то её вызова не произойдёт
- В JavaScript функции – это значения, поэтому мы и обращаемся с ними, как со значениями.
- Можно даже скопировать функцию в другую переменную.

```
function sayHi() {   // создаём
  alert( "Привет" );
}

let func = sayHi;    // копируем

func(); // Привет
// вызываем копию (работает)!
sayHi(); // Привет
// прежняя тоже работает (почему бы нет)
```

Функции-«колбэки»

- Это процесс передачи функций как значений (аргументов)

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  alert( "Вы согласны." );
}
function showCancel() {
  alert( "Вы отменили выполнение." );
}
// использование: функции showOk, showCancel передаются
// в качестве аргументов ask
ask("Вы согласны?", showOk, showCancel);
```

- Аргументы функции ещё называют функциями-колбэками или просто колбэками.
- Ключевая идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо.

Колбэк с использованием функционального выражения

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили выполнение."); }
);
```

Отличия Function Declaration от Function Expression.

- Function Declaration: функция объявляется отдельной конструкцией «function…» в основном потоке кода.

```
function sum(a, b) {
  return a + b;
}
```

- Function Expression: функция, созданная внутри другого выражения или синтаксической конструкции

```
let sum = function(a, b) {
  return a + b;
};
```

- Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.
- Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке).

Функция sayHi была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте.

```
sayHi("Вася"); // Привет, Вася

function sayHi(name) {
  alert( `Привет, ${name}` );
}
```

Функции, объявленные при помощи Function Expression, создаются тогда, когда выполнение доходит до них.

```
sayHi("Вася"); // ошибка!

let sayHi = function(name) {
// магии больше нет
  alert( `Привет, ${name}` );
};
```

Когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него.

```
let age = prompt("Сколько Вам лет?", 18);
// в зависимости от условия объявляем функцию
if (age < 18) {
  function welcome() {
    alert("Привет!");
  }
} else {
  function welcome() {
    alert("Здравствуйте!");
  }
}
welcome(); // Error: welcome is not defined
```

Когда использовать Function Declaration, а когда Function Expression?

- В большинстве случаев, когда нам нужно создать функцию, предпочтительно использовать Function Declaration, т.к. функция будет видима до своего объявления в коде. Это позволяет более гибко организовывать код, и улучшает его читаемость.
- Таким образом, мы должны прибегать к объявлению функций при помощи Function Expression в случае, когда синтаксис Function Declaration не подходит для нашей задачи.

Функции-стрелки

- Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.

```
let func = (arg1, arg2, ...argN) => expression
```

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

```
let sum = (a, b) => a + b;

/* Более короткая форма для:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

```
// тоже что и
// let double = function(n) { return n * 2 }
let double = n => n * 2;

alert( double(3) ); // 6
```

Если нет аргументов, указываются пустые круглые скобки:

```
let sayHi = () => alert("Hello!");

sayHi();
```

- Функции-стрелки могут быть использованы так же, как и Function Expression.

```
let age = prompt("Сколько Вам лет?", 18)
;

let welcome = (age < 18) ?
  () => alert('Привет') :
  () => alert("Здравствуйте!");

welcome(); // теперь всё в порядке
```

Многострочные стрелочные функции

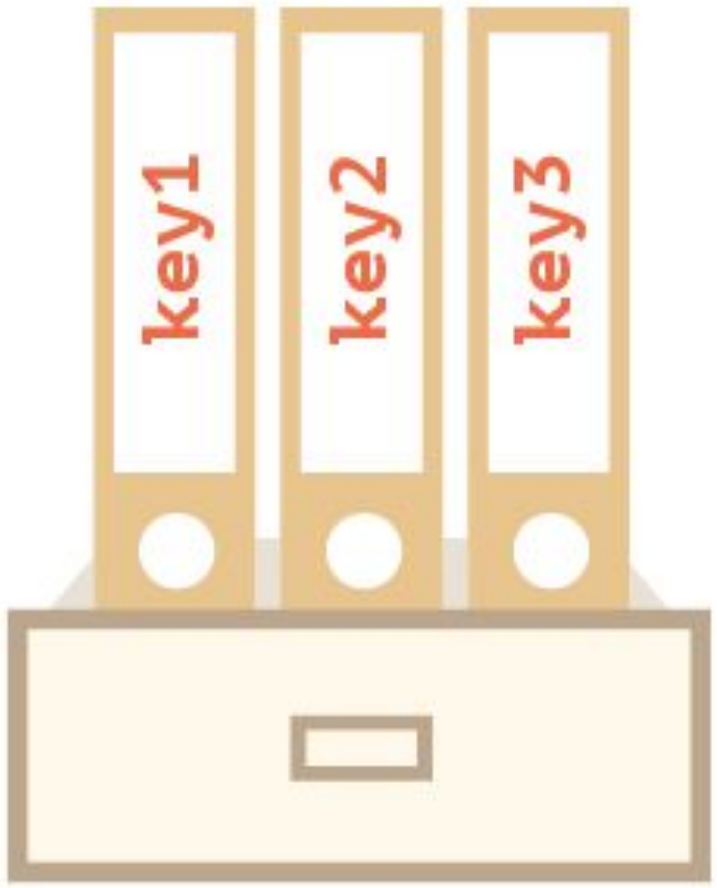
```
let sum = (a, b) => {
// фигурная скобка, открывающая тело многострочной функции
  let result = a + b;
  return result;
// при фигурных скобках для возврата значения
// нужно явно вызвать return
};

alert( sum(1, 2) ); // 3
```

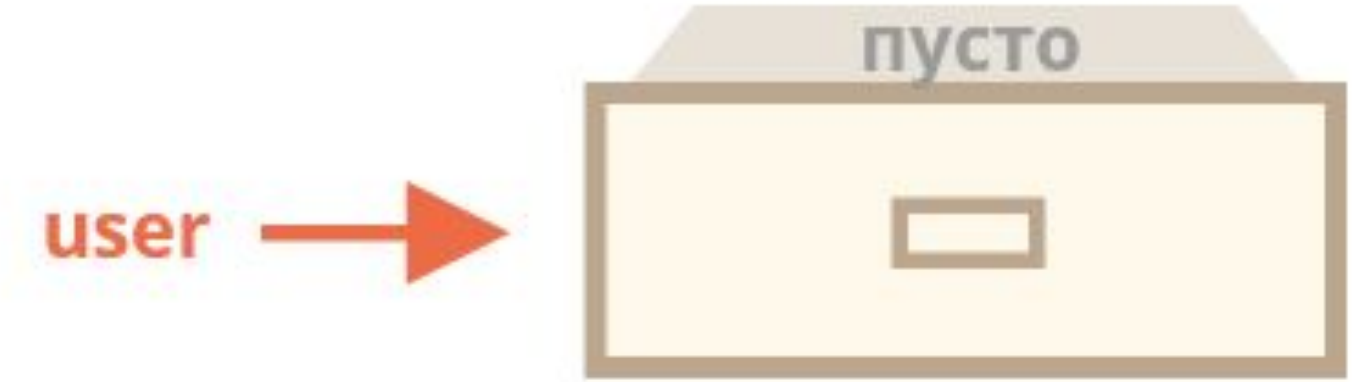
Объекты (Objects)

- Объекты используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка.

- Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.


key1
key2
key3

```
// синтаксис "конструктор объекта"
let user = new Object();
// синтаксис "литерал объекта"
let user = {};
```

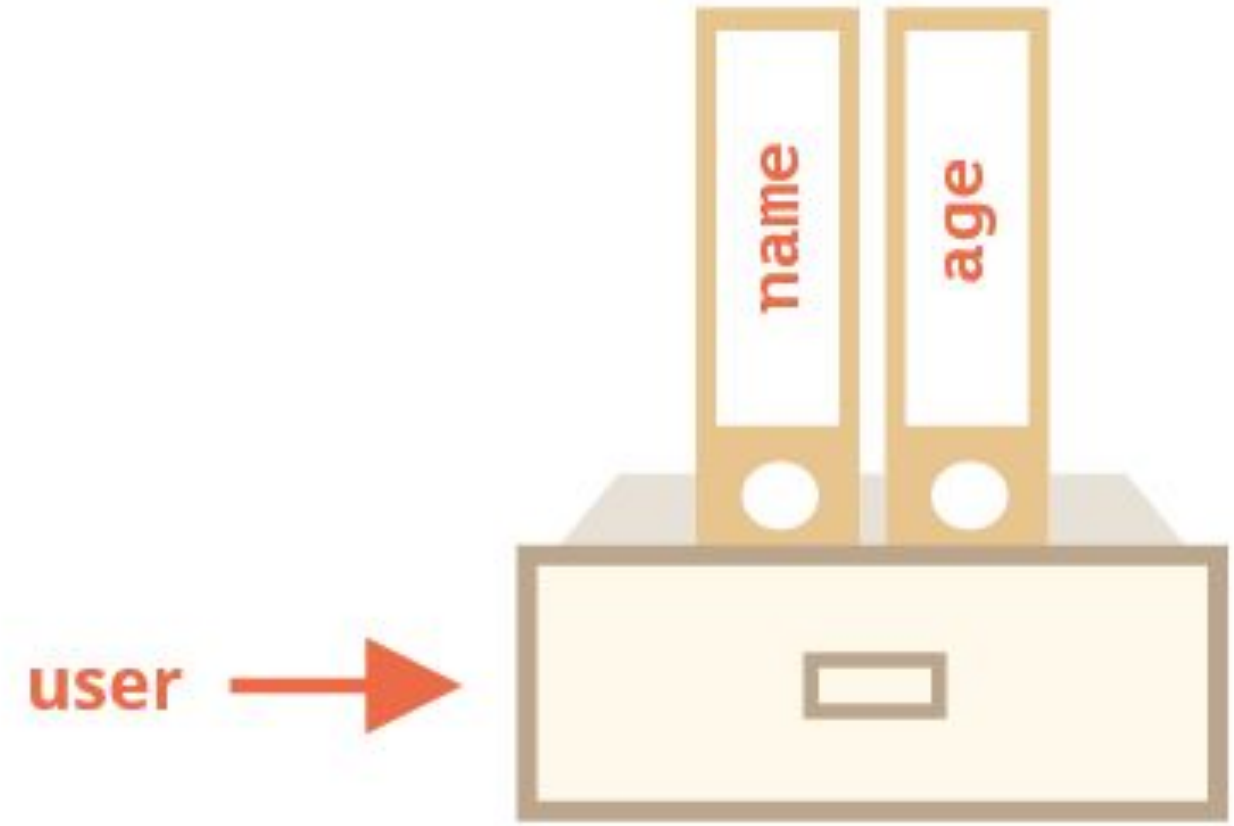


- Обычно используют вариант с фигурными скобками {...}. Такое объявление называют литералом объекта или литеральной нотацией.

Литералы и свойства

```
let user = {      // объект
  // под ключом "name" хранится значение "John"
  name: "John",
  // под ключом "age" хранится значение 30
  age: 30
};
```

- У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

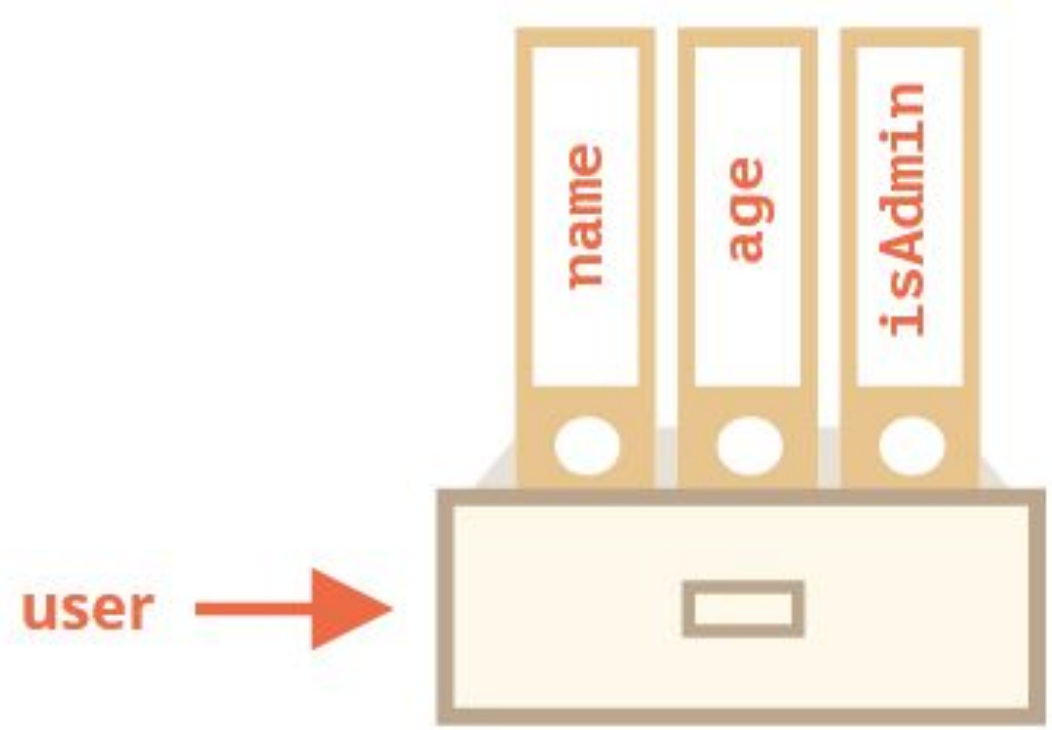

name
age
user

- Для обращения к свойствам используется запись «через точку»:

```
alert( user.name ); // John
alert( user.age ); // 30
```

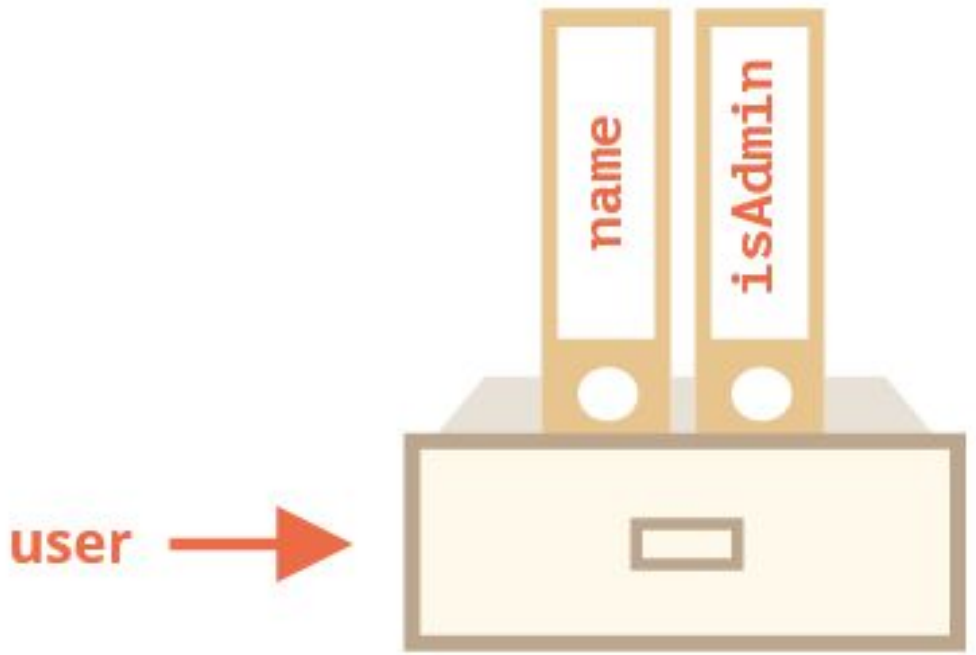
- Значение может быть любого типа.

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



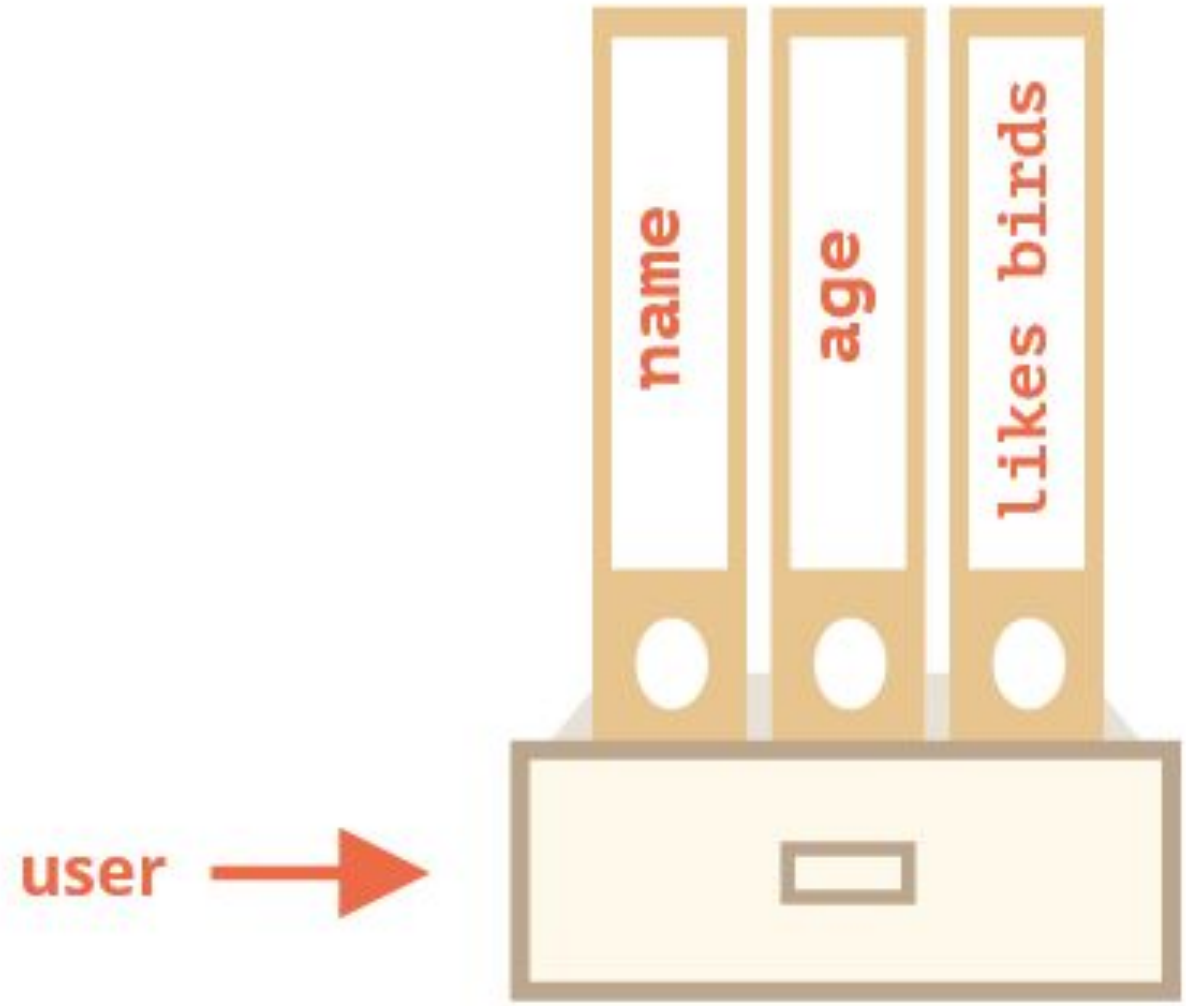
- Для удаления свойства используется оператор delete:

```
delete user.age;
```



- Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

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


name
age
likes birds
user

- Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

```
let user = {};
// присваивание значения свойству
user["likes birds"] = true;
// получение значения свойства
alert(user["likes birds"]); // true
// удаление свойства
delete user["likes birds"];
```

- Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

```
let key = "likes birds";
// то же самое, что и user["likes birds"] = true;
user[key] = true;
```

Вычисляемые свойства

- Можно использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства.

```
let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {
  [fruit]: 5,
// имя свойства будет взято из переменной fruit
};

alert( bag.apple ); // 5, если fruit="apple"
```

Зарезервированные слова разрешено использовать как имена свойств

```
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

Свойство из переменной

- В реальном коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем.

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ...другие свойства
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

- Если названия свойств и переменных совпадают можно использовать сокращённую запись

```
function makeUser(name, age) {
  return {
    name, // то же самое, что и name: name
    age   // то же самое, что и age: age
    // ...
  };
}
```

Проверка существования свойства

- Особенность объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет! При обращении к свойству, которого нет, возвращается undefined.
- Чтобы проверить существование свойства достаточно сравнить его с undefined.

```
let user = {};

alert( user.noSuchProperty === undefined );
;
// true означает "свойства нет"
```

- Также существует специальный оператор "in" для проверки существования свойства в объекте. Слева от оператора in должно быть имя свойства
- `"key" in object`

```
let user = { name: "John", age: 30 };
alert( "age" in user );
// true, user.age существует
alert( "blabla" in user );
// false, user.blabla не существует
```

```
let user = { age: 30 };

let key = "age";
alert( key in user );
// true, имя свойства было
// взято из переменной key
```

Цикл «for...in»

- Для перебора всех свойств объекта используется цикл for..in

```
for (key in object) {
  // тело цикла выполняется
  // для каждого свойства объекта
}
```

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

for (let key in user) {
  // ключи
  alert( key );  // name, age, isAdmin
  // значения ключей
  alert( user[key] ); // John, 30, true
}
```

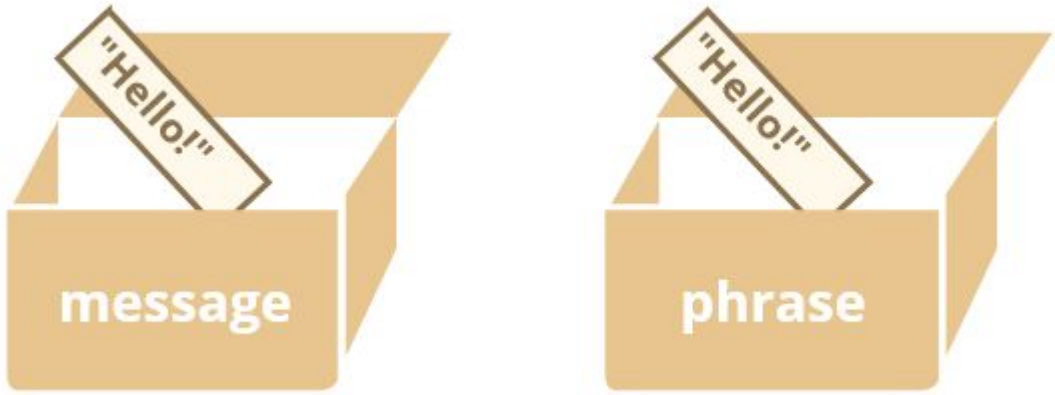
Копирование по ссылке

- Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке».

- Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению».

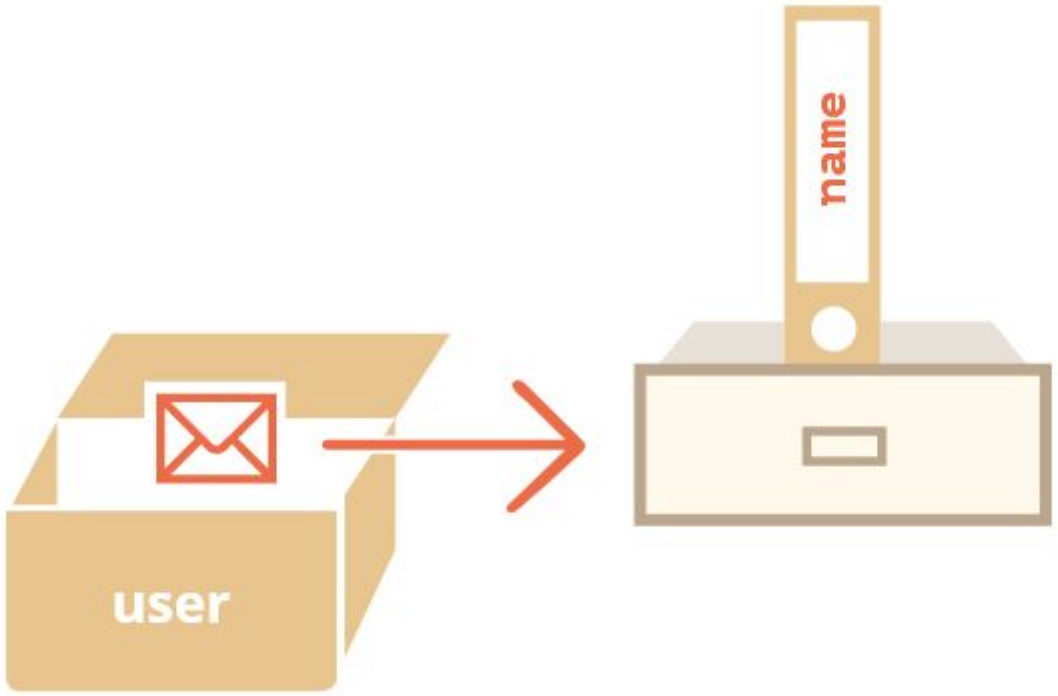
```
let message = "Hello!";
let phrase = message;
```

В результате мы имеем две независимые переменные, каждая из которых хранит строку "Hello!"



- Объекты ведут себя иначе.
- Они хранят не сам объект, а его «адрес в памяти», другими словами «ссылку» на него.

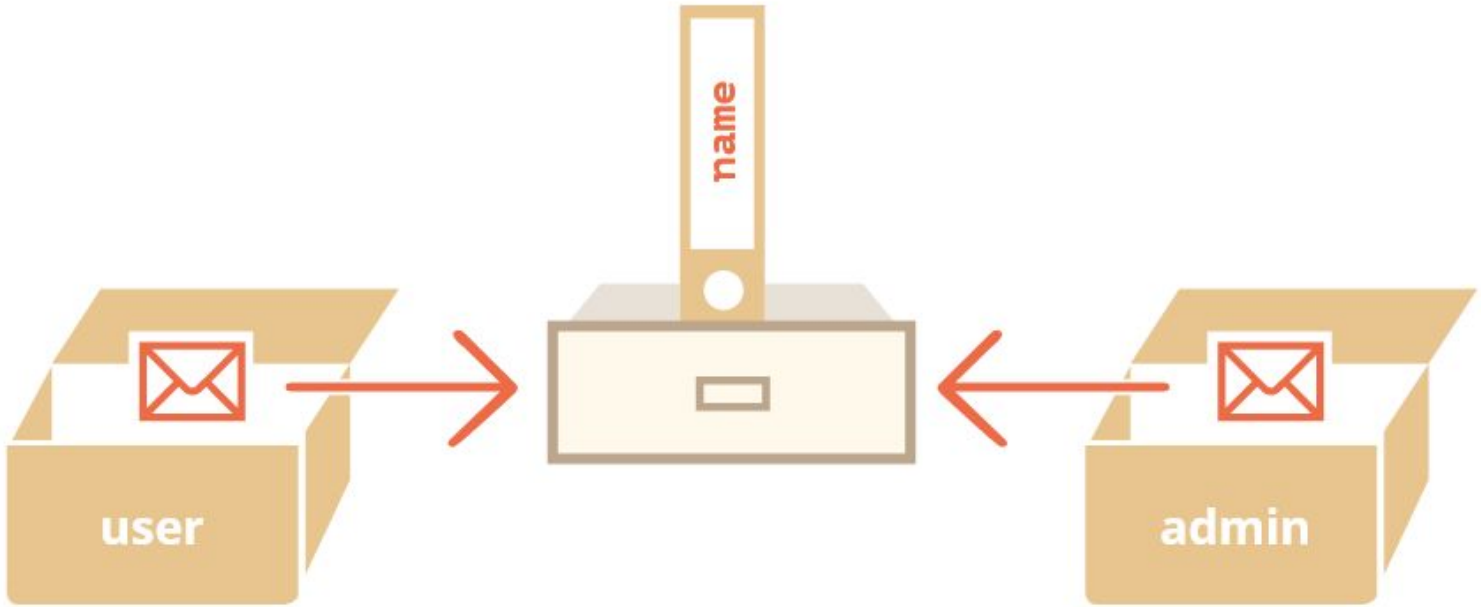
```
let user = {
  name: "John"
};
```



- Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти.
- Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется.

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

let admin = user; // копируется ссылка
```



Сравнение объектов

- Операторы равенства == и строгого равенства === для объектов работают одинаково.

- Два объекта равны только в том случае, если это один и тот же объект.

```
let a = {};
let b = a; // копирование по ссылке

alert( a == b );  true
alert( a === b ); true
```

```
let a = {};
let b = {}; // два независимых объекта

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

Объект, объявленный через const, может быть изменён.

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

user.age = 25;  // (*)
alert(user.age); // 25
```

- объявление const защищает от изменений только само значение user. А в нашем случае значение user – это ссылка на объект, и это значение мы не меняем. В строке (*) мы действуем внутри объекта, мы не переназначаем user

Клонирование и объединение объектов, Object.assign

- Нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их.

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

let clone = {}; // новый пустой объект

// скопируем все свойства user в него
for (let key in user) {
  clone[key] = user[key];
}

// теперь в переменной clone находится абсолютно независимый клон объекта.
clone.name = "Pete"; // изменим в нём данные

alert( user.name );
// в оригинальном объекте значение свойства `name` осталось прежним – John.
```

- Object.assign(dest, [src1, src2, src3...])

- Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.
- Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.
- Если принимающий объект уже имеет свойство с таким именем, оно будет перезаписано.

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

// свойство name перезапишется,
// свойство isAdmin добавится
Object.assign(user, { name: "Pete", isAdmin:
true });

// now user = { name: "Pete", isAdmin: true }
```

Клонирование объекта

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

let clone = Object.assign({}, user);
```