

# Разработка мобильных приложений

Лекция 9

# Остаточные параметры (...)

- ▶ Вызывать функцию можно с любым количеством аргументов независимо от того, как она была определена.
- ▶ Например (лишние аргументы не вызовут ошибку, но, конечно, посчитаются только первые два):

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

```
alert( sum(1, 2, 3, 4, 5) );
```

- ▶ Остаточные параметры могут быть обозначены через три точки .... Буквально это значит: «собери оставшиеся параметры и положи их в массив»

```
function sumAll(...args) { // args – ИМЯ МАССИВА
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}
alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

- ▶ Мы можем положить первые несколько параметров в переменные, а остальные - собрать в массив

```
function showName(firstName, lastName, ...titles) {  
  alert( firstName + ' ' + lastName ); Юлий Цезарь  
  
  alert( titles[0] ); Консул  
  alert( titles[1] ); Император  
  alert( titles.length ); 2  
}
```

```
showName( "Юлий", "Цезарь", "Консул", "Император" );
```

# Остаточные параметры должны располагаться в конце!

```
function f(arg1, ...rest, arg2) {  
    // arg2 после ...rest ?!  
    // Ошибка  
}
```

# Переменная "arguments"

- ▶ Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами.

```
function showName() {  
    alert( arguments.length );  
    alert( arguments[0] );  
    alert( arguments[1] );  
    // Объект arguments можно перебирать  
    // for (let arg of arguments) alert(arg);  
}  
showName( "Юлий", "Цезарь"); Юлий Цезарь  
showName( "Илья"); Илья undefined
```

# Стрелочные функции не имеют "arguments"

- ▶ Если мы обратимся к `arguments` из стрелочной функции, то получим аргументы внешней «нормальной» функции.

```
function f() {  
  let showArg = () => alert(arguments[0]);  
  showArg(2);  
}
```

```
f(1); // 1
```

# Оператор расширения

- ▶ Например, есть встроенная функция `Math.max`. Она возвращает наибольшее число из списка:

```
alert( Math.max(3, 5, 1) ); // 5
```

- ▶ Допустим, у нас есть массив чисел `[3, 5, 1]`. Как вызвать для него `Math.max`?
- ▶ Просто так их не вставишь — `Math.max` ожидает получить список чисел, а не один массив.

```
let arr = [3, 5, 1];  
alert( Math.max(arr) ); // NaN
```



- ▶ Оператор расширения похож на остаточные параметры - тоже использует ..., но делает совершенно противоположное.
- ▶ Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов.

```
let arr = [3, 5, 1];
```

```
alert( Math.max(...arr) ); 5
```

```
// оператор "раскрывает" массив в список аргументов
```

- ▶ Этим же способом мы можем передать несколько итерируемых объектов.

```
let arr1 = [1, -2, 3, 4];
```

```
let arr2 = [8, 3, -8, 1];
```

```
alert( Math.max(...arr1, ...arr2) ); 8
```

- ▶ Мы даже можем комбинировать оператор расширения с обычными значениями:

```
let arr1 = [1, -2, 3, 4];
```

```
let arr2 = [8, 3, -8, 1];
```

```
alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); 25
```

- ▶ Оператор расширения можно использовать и для слияния массивов:

```
let arr = [3, 5, 1];
```

```
let arr2 = [8, 9, 15];
```

```
let merged = [0, ...arr, 2, ...arr2];
```

```
alert(merged);
```

```
// 0,3,5,1,2,8,9,15 (0, затем arr, затем 2, в конце arr2)
```

- ▶ Оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

```
let str = "Привет";  
alert( [...str] ); // П,р,и,в,е,т
```

- ▶ Под капотом оператор расширения использует итераторы, чтобы перебирать элементы. Так же, как это делает `for..of`.
- ▶ Цикл `for..of` перебирает строку как последовательность символов, поэтому из `...str` получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: `[...str]`.

# Генераторы

- ▶ Обычные функции возвращают только одно-единственное значение (или ничего).
- ▶ Генераторы могут породить (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

# Функция-генератор

- ▶ Для объявления генератора используется специальная синтаксическая конструкция: `function*`, которая называется «функция-генератор».

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}
```

- ▶ Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор», для управления её выполнением.

- ▶ Основным методом генератора является `next()`. При вызове он запускает выполнение кода до ближайшей инструкции `yield <значение>` (значение может отсутствовать, в этом случае оно предполагается равным `undefined`). По достижении `yield` выполнение функции приостанавливается, а соответствующее значение - возвращается во внешний код:
- ▶ Результатом метода `next()` всегда является объект с двумя свойствами:
  - ▶ `value`: значение из `yield`.
  - ▶ `done`: `true`, если выполнение функции завершено, иначе `false`.



```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}
```

```
let generator = generateSequence();
```

```
let one = generator.next();
```

```
alert(JSON.stringify(one)); // {value: 1, done: false}
```

- ▶ Повторный вызов `generator.next()` возобновит выполнение кода и вернёт результат следующего `yield`:

```
let two = generator.next();
```

```
alert(JSON.stringify(two)); // {value: 2, done: false}
```

- ▶ И, наконец, последний вызов завершит выполнение функции и вернёт результат return:

```
let three = generator.next();
```

```
alert(JSON.stringify(three)); // {value: 3, done: true}
```

# Перебор генераторов

- ▶ Возвращаемые генераторами значения можно перебирать через for..of:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}
```

```
let generator = generateSequence();
```

```
for(let value of generator) {  
  alert(value); // 1, затем 2  
}
```

- ▶ перебор через `for..of` игнорирует последнее значение, при котором `done: true`. Поэтому, если мы хотим, чтобы были все значения при переборе через `for..of`, то надо возвращать их через `yield`

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
let generator = generateSequence();  
  
for(let value of generator) {  
  alert(value); // 1, затем 2, затем 3  
}
```

# Композиция генераторов

- ▶ Это особенная возможность генераторов, которая позволяет прозрачно «встраивать» генераторы друг в друга.

```
function* generateSequence(start, end) {  
  for (let i = start; i <= end; i++) yield i;  
}
```

- ▶ Мы хотели бы использовать её при генерации более сложной последовательности:
- ▶ сначала цифры 0..9 (с кодами символов 48...57)
- ▶ за которыми следуют буквы в верхнем регистре A..Z (коды символов 65...90)
- ▶ за которыми следуют буквы алфавита a..z (коды символов 97...122)



```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
function* generatePasswordCodes() {
  yield* generateSequence(48, 57); // 0..9
  yield* generateSequence(65, 90); // A..Z
  yield* generateSequence(97, 122); // a..z
}

let str = '';
for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}
alert(str); // 0..9A..Za..z
```

- ▶ Директива `yield*` делегирует выполнение другому генератору. Этот термин означает, что `yield* gen` перебирает генератор `gen` и прозрачно направляет его вывод наружу. Как если бы значения были сгенерированы внешним генератором.
- ▶ Результат - такой же, как если бы мы встроили код из вложенных генераторов

# yield - дорога в обе стороны

- ▶ yield не только возвращает результат наружу, но и может передавать значение извне в генератор.

```
function* gen() {  
    // Передаём вопрос во внешний код и ожидаем ответа  
    let result = yield "2 + 2 = ?";  
    alert(result);  
}  
  
let generator = gen();  
let question = generator.next().value;  
    // <-- yield возвращает значение  
generator.next(4); // --> передаём результат в генератор
```

- ▶ Первый вызов `generator.next()` - всегда без аргумента, он начинает выполнение и возвращает результат первого `yield` "2+2=?". На этой точке генератор приостанавливает выполнение.
- ▶ Затем, результат `yield` переходит во внешний код в переменную `question`.
- ▶ При `generator.next(4)` выполнение генератора возобновляется, а 4 выходит из присваивания как результат: `let result = 4`.
- ▶ Обратите внимание, что внешний код не обязан немедленно вызывать `next(4)`. Ему может потребоваться время. Это не проблема, генератор подождёт.

# Модули

- ▶ По мере роста нашего приложения, мы обычно хотим разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями.
- ▶ Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости.
- ▶ Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули.

# Что такое модуль?

- ▶ Модуль - это просто файл. Один скрипт - это один модуль.
- ▶ Модули могут загружать друг друга и использовать директивы **export** и **import**, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:
- ▶ **export** отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
- ▶ **import** позволяет импортировать функциональность из других модулей.

```
export function sayHi(user) {  
    alert(`Привет, ${user}!`);  
}
```

```
import {sayHi} from './sayHi';
```

```
alert(sayHi); // функция  
sayHi('John'); // Привет, Илья!
```

# Основные возможности модулей

- ▶ В модулях всегда используется режим `use strict`.
- ▶ Своя область видимости переменных
- ▶ Код в модуле выполняется только один раз при импорте: если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.
- ▶ В модуле на верхнем уровне `this` не определён (`undefined`).



# Описание экспорта

- ▶ Именованный экспорт:

```
export { myFunction }; // экспорт ранее объявленной функции
```

```
export const foo = Math.sqrt(2); // экспорт константы
```

- ▶ Дефолтный экспорт (экспорт по умолчанию) (один на скрипт):

```
export default function() {} // или 'export default class {}'
```

```
// тут не ставится точка с запятой
```

- ▶ Именованная форма более применима для экспорта нескольких величин. Во время импорта, можно будет использовать одно и то же имя, чтобы обратиться к соответствующему экспортируемому значению.
- ▶ Касательно экспорта по умолчанию (default), он может быть только один для каждого отдельного модуля (файла). Дефолтный экспорт может представлять собой функцию, класс, объект или что-то другое. Это значение следует рассматривать как "основное", так как его будет проще всего импортировать.

# Использование именованного экспорта

```
// модуль "my-module"  
function cube(x) {  
    return x * x * x;  
}  
const foo = Math.PI + Math.SQRT2;  
export { cube, foo };
```

- ▶ Затем можем импортировать модуль:

```
import { cube, foo } from 'my-module';  
console.log(cube(3)); // 27  
console.log(foo);    // 4.555806215962888
```

# Использование export default

- ▶ Если мы хотим экспортировать единственное значение или иметь резервное значение (fallback) для данного модуля, мы можем использовать export default.

```
export default function cube(x) {  
    return x * x * x;  
}
```

- ▶ Затем, в другом скрипте можно импортировать это значение по умолчанию таким образом:

```
import cube from 'my-module';  
console.log(cube(3)); // 27
```

# Импорт всего содержимого модуля

```
import * as myModule from './modules/my-module';
```

- ▶ Этот код вставляет объект `myModule` в текущую область видимости, содержащую все экспортированные значения из модуля, находящегося в файле `/modules/my-module.js`. В данном случае, доступ к импортируемым значениям можно осуществить с использованием имени модуля (в данном случае `"myModule"`) в качестве пространства имен.

```
myModule.doAllTheAmazingThings();
```

# Импорт единичного значения из модуля

```
import {myExport} from '/modules/my-module';
```

- ▶ Определенное ранее значение, названное myExport, которое было экспортировано из модуля my-module либо неявно (если модуль был экспортирован целиком), либо явно (с использованием инструкции export), позволяет вставить myExport в текущую область видимости.

# Импорт нескольких единичных значений

```
import {foo, bar} from '/modules/my-module';
```

- ▶ Код вставляет оба значения foo и bar в текущую область видимости.

# Импорт значений с использованием более удобных имен

```
import {reallyReallyLongModuleName as shortName}  
  from '/modules/my-module';
```

- ▶ Вы можете переименовать значения, когда импортируете их. Код вставляет `shortName` в текущую область видимости.



# Переименование нескольких значений в одном импорте

```
import {  
    reallyReallyLongModuleName as shortName,  
    anotherLongModuleName as short  
} from '/modules/my-module';
```

- ▶ Код, который импортирует несколько значений из модуля, используя более удобные имена.

# Импорт модуля для использования его побочного эффекта

```
import '/modules/my-module';
```

- ▶ Импорт всего модуля только для использования побочного эффекта от его вызова, не импортируя что-либо. Это запускает глобальный код модуля, но в действительности не импортирует никаких значений.

# Импорт значения по умолчанию

- ▶ Есть возможность задать дефолтный export (будь то объект, функция, класс или др.). Инструкция import затем может быть использована для импорта таких значений.
- ▶ Простейшая версия прямого импорта значения по умолчанию:

```
import myDefault from '/modules/my-module';
```

- ▶ Возможно также использование такого синтаксиса с другими вариантами из перечисленных выше (импорт пространства имен или именованный импорт). В таком случае, импорт значения по умолчанию должен быть определён первым.

```
import myDefault, * as myModule from '/modules/my-module';
```

```
import myDefault, {foo, bar} from '/modules/my-module';
```