Разработка мобильных приложений

Лекция 8

Что такое класс?

- В JavaScript класс – это разновидность функции.
- В отличие от обычных функций, конструктор класса не может быть вызван без new.

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// доказательство: User - это функция
alert(typeof User); // function
```

Class Expression

- Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

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

Можно динамически создавать классы «по запросу»

```
function makeClass(phrase) {
  // объявляем класс и возвращаем его
  return class {
    sayHi() {
      alert(phrase);
    };
  };
}
// Создаём новый класс
let User = makeClass("Привет");
new User().sayHi(); // Привет
```

Свойства классов

```
class User {
  name = "Аноним";

  sayHi() {
    alert(`Привет, ${this.name}!`);
  }
}
new User().sayHi();
```

Свойства - геттеры и сеттеры

- Есть два типа свойств объекта.
- Первый тип это свойства-данные (data properties). Все свойства, которые использовались до текущего момента, были свойствами-данными.
- Второй тип свойств это свойства-аксессоры (accessor properties). По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

- Свойства-аксессоры представлены методами: «геттер» - для чтения и «сеттер» - для записи. При литеральном объявлении объекта они обозначаются get и set:

```
let obj = {
  get propName() {
    // геттер, срабатывает при чтении obj.propName
  },
  set propName(value) {
    // сеттер, срабатывает при записи obj.propName = value
  }
};
```

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

alert(user.fullName); // John Smith
```

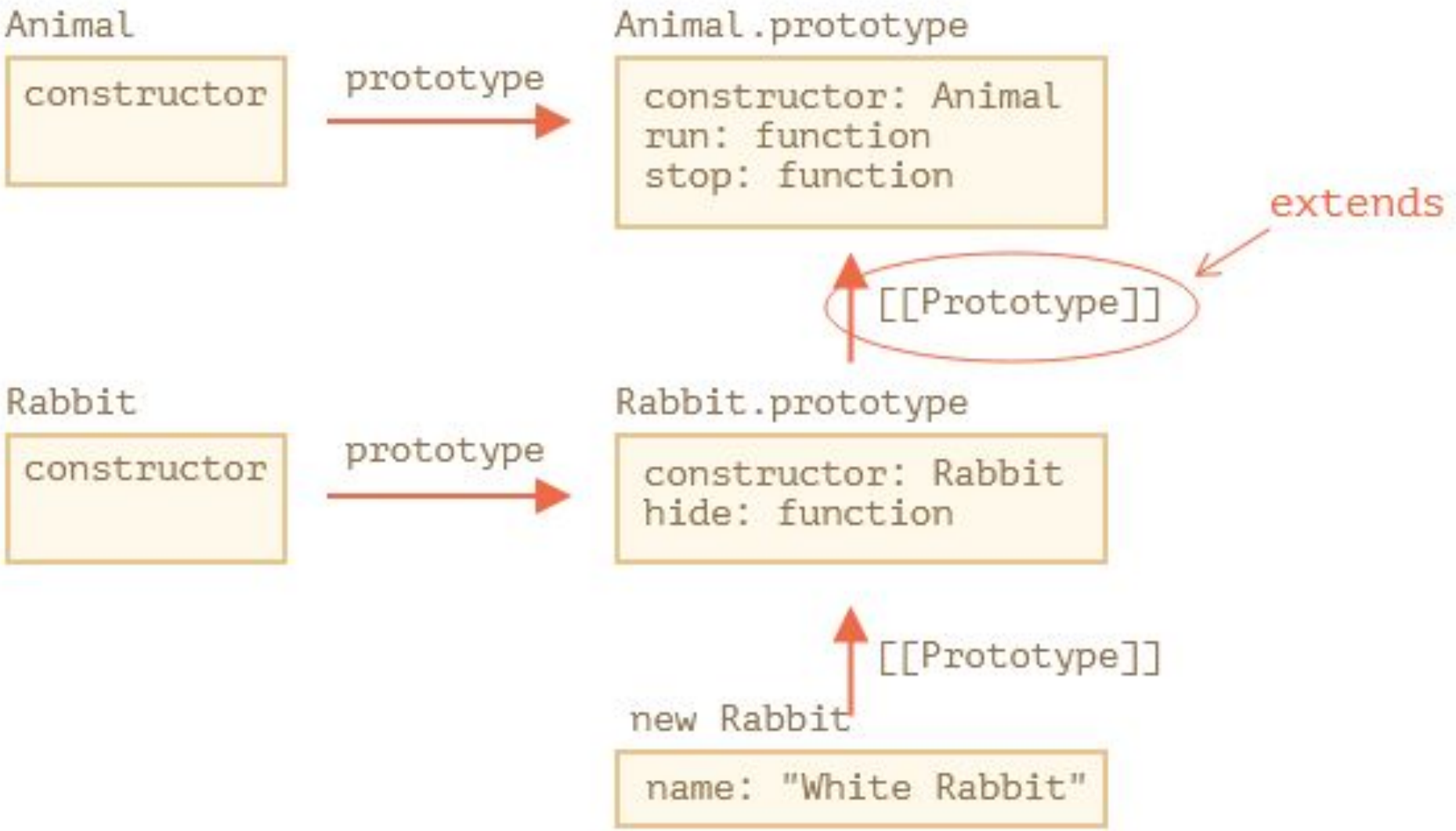
```
class MyClass {
  prop = value; // свойство
  constructor(...) { // конструктор
    // ...
  }
  method(...) {} // метод
  get something(...) {} // геттер
  set something(...) {} // сеттер
  // ...
}
```

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} стоит.`);
  }
}

let animal = new Animal("Мой питомец");
```

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbit = new Rabbit("Мой кролик");
```

- Сейчас они полностью независимы.
- Чтобы Rabbit расширял Animal, кролики должны происходить от животных, т.е. иметь доступ к методам Animal и расширять функциональность Animal своими методами.
- Для того, чтобы наследовать класс от другого, нужно использовать ключевое слово "extends" и указать название родительского класса перед {..}.

```
// Наследуем от Animal указывая "extends Animal"
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }
}
let rabbit = new Rabbit("Белый кролик");
rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.hide(); // Белый кролик прячется!
```


Animal
constructor
prototype
Animal.prototype
constructor: Animal
run: function
stop: function
extends
[[Prototype]]
Rabbit
constructor
prototype
Rabbit.prototype
constructor: Rabbit
hide: function
[[Prototype]]
new Rabbit
name: "White Rabbit"

Переопределение методов

- Если мы определим свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

```
class Rabbit extends Animal {
  stop() {
    // ...
будет использован для rabbit.stop()
  }
}
```

Ключевое слово "super"

- super.method(...) вызывает родительский метод
- super(...) вызывает родительский конструктор (работает только внутри конструктора)
- При переопределении конструктора обязателен вызов конструктора родителя super() в конструкторе Child до обращения к this

```
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }
  stop() {
    super.stop(); // вызываем родительский метод stop
    this.hide(); // и затем hide
  }
}
let rabbit = new Rabbit("Белый кролик");
rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!
```

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Rabbit extends Animal {
  constructor(name) {
    this.name = name;
    this.created = Date.now();
  }
}

let rabbit = new Rabbit("Белый кролик"); // ОШИБКА! This не
определно
alert(rabbit.name);
```

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Rabbit extends Animal {
  constructor(name) {
    super(name);
    this.created = Date.now();
  }
}

let rabbit = new Rabbit("Белый кролик");
alert(rabbit.name);
```

Статические свойства и методы

- Можно присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими.

- В классе такие методы обозначаются ключевым словом static

```
class User {
  static staticMethod() {
    alert(this === User);
  }
}
User.staticMethod(); // true
```

- Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

- Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам.

Статические свойства

- Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

```
class Article {
  static publisher = "Илья Кантор";
}

alert( Article.publisher ); // Илья Кантор
```

Наследование статических свойств и методов

- Статические свойства и методы наследуются.
- Статические методы используются для функциональности, принадлежат классу «в целом», а не относятся к конкретному объекту класса.
- Статические свойства используются в тех случаях, когда мы хотели бы сохранить данные на уровне класса, а не какого-то одного объекта.

Приватные и защищённые методы и свойства

- Один из важнейших принципов объектно-ориентированного программирования – разделение внутреннего и внешнего интерфейсов.

- Внутренний интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса.
- Внешний интерфейс – методы и свойства, доступные снаружи класса.

Приватные и защищённые методы и свойства

- В JavaScript есть два типа полей (свойств и методов) объекта:
- Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
- Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Защищённые методы и свойства

- Защищённые свойства обычно начинаются с префикса _.
- Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению.

```
class CoffeeMachine {
  // ...
  constructor(power) {
    this._power = power;
  }
  get power() {
    return this._power;
  }
}
// создаём кофеварку
let coffeeMachine = new CoffeeMachine(100);
alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W
coffeeMachine.power = 25; // Error (no setter)
```

Приватные свойства

- Есть новшество в языке JavaScript, которое почти добавлено в стандарт: оно добавляет поддержку приватных свойств и методов.

- Приватные свойства и методы должны начинаться с #. Они доступны только внутри класса.

```
class CoffeeMachine {
  #waterLimit = 200;
  #checkWater(value) {
    if (value < 0) throw new Error("Отрицательный
 уровень воды");
    if (value > this.#waterLimit) throw new Error(
"Слишком много воды");
  }
}
let coffeeMachine = new CoffeeMachine();
// снаружи  нет доступа к приватным методам класса
coffeeMachine.#checkWater(); // Error
coffeeMachine.#waterLimit = 1000; // Error
```

- На уровне языка # является специальным символом, который означает, что поле приватное. Мы не можем получить к нему доступ извне или из наследуемых классов.

- Приватные поля не конфликтуют с публичными. У нас может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

Оператор instanceof

- Оператор instanceof позволяет проверить, к какому классу принадлежит объект, с учётом наследования.

- Такая проверка может потребоваться во многих случаях. К примеру для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

Синтаксис

```
class Rabbit {}
let rabbit = new Rabbit();

// это объект класса Rabbit?
alert( rabbit instanceof Rabbit ); // true
```

Примеси

- В JavaScript можно наследовать только от одного объекта. Объект имеет единственный [[Prototype]]. И класс может расширить только один другой класс.
- Иногда это может ограничивать нас. Например, у нас есть класс StreetSweeper и класс Bicycle, а мы хотим создать их смесь: StreetSweepingBicycle.
- Или у нас есть класс User, который реализует пользователей, и класс EventEmitter, реализующий события. Мы хотели бы добавить функционал класса EventEmitter к User, чтобы пользователи могли легко генерировать события.

Примеси

- Примесь – это класс, методы которого предназначены для использования в других классах, причём без наследования от примеси.

- Другими словами, примесь определяет методы, которые реализуют определённое поведение. Мы не используем примесь саму по себе, а используем её, чтобы добавить функционал другим классам.

```
// примесь
let sayHiMixin = {
  sayHi() {
    alert(`Привет, ${this.name}`);
  },
  sayBye() {
    alert(`Пока, ${this.name}`);
  }
};

// использование:
class User {
  constructor(name) {
    this.name = name;
  }
}
// копируем методы
Object.assign(User.prototype, sayHiMixin);
// теперь User может сказать Привет
new User("Вася").sayHi(); // Привет, Вася!
```

- Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть легко добавлены в прототип любого класса.
- Это не наследование, а просто копирование методов.
- Примеси могут наследовать друг другу.

Обработка ошибок, "try..catch"

- Конструкция try..catch состоит из двух основных блоков: try, и затем catch

```
try {
    // код...
} catch (err) {
    // обработка ошибки
}
```

Обработка ошибок, "try..catch"

- Работает она так:
- Сначала выполняется код внутри блока try {...}.
- Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
- сли же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

```
try {
    alert('Начало блока try');
    lalala; // ошибка, переменная не определена!
    alert('Конец блока try (никогда не выполнится)');
} catch(err) {
    alert(`Возникла ошибка!`);
}
```

- try..catch работает только для ошибок, возникающих во время выполнения кода

- Чтобы try..catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код.

```
try {
  {{{{{{{{{{
} catch(e) {
  alert("Движок не может понять этот код, он некорректен");
}
```

Объект ошибки

- Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок catch:

```
try {
  // ...
} catch(err) { // <-- объект ошибки, можно исполь
зовать другое название вместо err
  // ...
}
```

Для всех встроенных ошибок этот объект имеет два основных свойства:

name

Имя ошибки. Например, для неопределённой переменной это "ReferenceError".

message

Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это:

stack

Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

```
try {
  lalala; // ошибка, переменная не определена!
} catch(err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined at (...стек вызовов)

  // Можем также просто вывести ошибку целиком
  // Ошибка приводится к строке вида "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
```

Оператор «throw»

- Оператор throw генерирует ошибку.
- Синтаксис:

```
throw <объект ошибки>
```

- Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенными ошибками).

- В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки.

```
let error1 = new Error(message);
// или
let error2 = new SyntaxError(message);
let error3 = new ReferenceError(message);
```

- Блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.
- Техника «проброс исключения» выглядит так:
- Блок catch получает все ошибки.
- В блоке catch(err) {...} мы анализируем объект ошибки err.
- Если мы не знаем как её обработать, тогда делаем throw err.

try…catch…finally

- Конструкция try..catch может содержать ещё одну секцию: finally.
- Если секция есть, то она выполняется в любом случае:
- после try, если не было ошибок,
- после catch, если ошибки были.
- Секцию finally часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от того, будет ошибка или нет.

```
try {
    ... пробуем выполнить код...
} catch(e) {
    ... обрабатываем ошибки ...
} finally {
    ... выполняем всегда ...
}
```

- Переменные внутри try..catch..finally локальны
- Блок finally срабатывает при любом выходе из try..catch, в том числе и return.

```
function func() {
    try {
        return 1;

    } catch (e) {
        /* ... */
    } finally {
        alert('finally');
    }
}
alert(func());
// сначала срабатывает alert из finally, а затем этот код
```

try..finally

- Конструкция try..finally без секции catch также полезна. Её применяют когда не хотят здесь обрабатывать ошибки (пусть выпадут), но хотят быть уверены, что начатые процессы завершились.

```
function func() {
  // начать делать что-то,
  // что требует завершения (например, измерения)
  try {
    // ...
  } finally {
    // завершить это, даже если все упадёт
  }
}
```

Промисы

- Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети.
- Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции.
- Promise (по англ. promise, будем называть такой объект «промис») – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а промис делает результат доступным для кода, который подписан на него, когда результат готов.

Синтаксис создания Promise:

```
let promise = new Promise(function(resolve, reject) {
  // …
});
```

- Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат.
- Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript.
- Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:
- resolve(value) — если работа завершилась успешно, с результатом value.
- reject(error) — если произошла ошибка, error – объект ошибки.

Может быть что-то одно: либо результат, либо ошибка

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("...")); // игнорируется
  setTimeout(() => resolve("...")); // игнорируется
});
```

Исполнитель должен вызвать что-то одно: resolve или reject.
Состояние промиса может быть изменено только один раз.

Потребители: then, catch, finally

- Объект Promise служит связующим звеном между исполнителем («создающим» кодом) и функциями-потребителями, которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов .then, .catch и .finally.

then

- Наиболее важный и фундаментальный метод – .then.

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

- Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.
- Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

catch

- Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает тоже самое
- Вызов .catch(f) – это сокращённый, «укороченный» вариант .then(null, f).

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});

// .catch(f) это тоже самое, что promise.then(null, f)
promise.catch(alert);
// выведет "Error: Ошибка!" спустя одну секунду
```

finally

- По аналогии с блоком finally из обычного try {...} catch {...}, у промисов также есть метод finally.
- Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой.
- finally хорошо подходит для очистки, например остановки индикатора загрузки, его ведь нужно остановить вне зависимости от результата.

```
new Promise((resolve, reject) => {
  /* сделать что-
то, что займёт время, и после вызвать resolve/reject */
})
  // выполнится, когда промис завершится, независимо от того, у
спешно или нет
  .finally(() => остановить индикатор загрузки)
  .then(result => показать результат, err => показать ошибку)
```

Цепочка промисов

- Идея состоит в том, что результат первого промиса передаётся по цепочке обработчиков .then.

- Начальный промис успешно выполняется через 1 секунду (*),
- Затем вызывается обработчик в .then (**).
- Возвращаемое им значение передаётся дальше в следующий обработчик .then (***)
- …и так далее.

```
new Promise(function (resolve, reject) {
    setTimeout(() => resolve(1), 1000); // (*)
}).then(function (result) { // (**)
    alert(result); // 1
    return result * 2;
}).then(function (result) { // (***)
    alert(result); // 2
    return result * 2;
}).then(function (result) {
    alert(result); // 4
    return result * 2;
});
```

Async/await

- Специальный синтаксис для работы с промисами.

```
async function f() {
  return 1;
}
```

- У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

```
async function f() {
    return 1;
  }

f().then(alert); // 1
```

- Ключевое слово async перед функцией гарантирует, что эта функция в любом случае вернёт промис.

Await

- Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
  });
  let result = await promise;
  // будет ждать, пока промис не выполнится
  alert(result); // "готово!"
}
f();
```

- Хотя await и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.
- По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

- await нельзя использовать в обычных функциях
- await нельзя использовать на верхнем уровне вложенности
- await работает только внутри async-функций
- await работает с «thenable»-объектами. Как и promise.then, await позволяет работать с промис-совместимыми объектами. Идея в том, что если у объекта можно вызвать метод then, этого достаточно, чтобы использовать его с await.

Асинхронные методы классов

- Для объявления асинхронного метода достаточно написать async перед именем:

```
class Waiter {
    async wait() {
      return await Promise.resolve(1);
    }
  }
new Waiter()
   .wait()
   .then(alert); // 1
```

Обработка ошибок

- Когда промис завершается успешно, await promise возвращает результат. Когда завершается с ошибкой - будет выброшено исключение. Как если бы на этом месте находилось выражение throw

```
async function f() {
    throw new Error("Упс!");
}
```

- Промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем await выбросит исключение.
- Такие ошибки можно ловить, используя try..catch, как с обычным throw:

```
async function f() {
    try {
      let response = await fetch('http://no-such-url');
    } catch(err) {
      alert(err); // TypeError: failed to fetch
    }
  }
f();
```

async/await отлично работает с Promise.all

```
// await будет ждать массив с
// результатами выполнения всех промисов
let results = await Promise.all([
    fetch(url1),
    fetch(url2),
    ...
  ]);
```

- Ключевое слово async перед объявлением функции:
- Обязывает её всегда возвращать промис.
- Позволяет использовать await в теле этой функции.

- Ключевое слово await перед промисом заставит JavaScript дождаться его выполнения, после чего:
- Если промис завершается с ошибкой, будет сгенерировано исключение, как если бы на этом месте находилось throw.
- Иначе вернётся результат промиса.

Map

- Map – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

- new Map() – создаёт коллекцию.
- map.set(key, value) – записывает по ключу key значение value.
- map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.
- map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.
- map.delete(key) – удаляет элемент по ключу key.
- map.clear() – очищает коллекцию от всех элементов.
- map.size – возвращает текущее количество элементов.

```
let map = new Map();

map.set("1", "str1");    // строка в качестве ключа
map.set(1, "num1");      // цифра как ключ
map.set(true, "bool1");  // булево значение как ключ
// Map сохраняет тип ключей,
// так что в этом случае сохранится 2 разных значения:
alert(map.get(1)); // "num1"
alert(map.get("1")); // "str1"

alert(map.size); // 3
```

Map может использовать объекты в качестве ключей.

```
let john = { name: "John" };
let visitsCountMap = new Map();

// объект john - это ключ для значения в объекте Map

visitsCountMap.set(john, 123);
alert(visitsCountMap.get(john)); // 123
```

Как объект Map сравнивает ключи

- Чтобы сравнивать ключи, объект Map использует алгоритм SameValueZero. Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа.

- Этот алгоритм не может быть заменён или модифицирован.

Перебор Map

- Для перебора коллекции Map есть 3 метода:

- map.keys() – возвращает итерируемый объект по ключам,
- map.values() – возвращает итерируемый объект по значениям,
- map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

```
let recipeMap = new Map([
    ["огурец", 500],
    ["помидор", 350],
    ["лук",    50]
  ]);
```

```
// перебор по ключам (овощи)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // огурец, помидор, лук
}
```

```
// перебор по значениям (числа)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}
```

```
// перебор по элементам в формате [ключ, значение]
for (let entry of recipeMap) {
 // то же самое, что и recipeMap.entries()
  alert(entry); // огурец,500 (и так далее)
}
```

Set

- Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

- new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
- set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
- set.delete(value) – удаляет значение, возвращает true если value было в множестве на момент вызова, иначе false.
- set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.
- set.clear() – удаляет все имеющиеся значения.
- set.size – возвращает количество элементов в множестве.

- при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

Перебор объекта Set

- Можно перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach:

```
let set = new Set(["апельсин", "яблоко", "банан"]);
for (let value of set) alert(value);
// то же самое с forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

- Set имеет те же встроенные методы, что и Map:

- set.keys() – возвращает перебираемый объект для значений,
- set.values() – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,
- set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

Массив: перебирающие методы

- **forEach**
- Метод «arr.forEach(callback[, thisArg])» используется для перебора массива.
- Он для каждого элемента массива вызывает функцию callback.
- Этой функции он передаёт три параметра callback(item, i, arr):
- item – очередной элемент массива.
- i – его номер.
- arr – массив, который перебирается.

```
var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(item, i, arr) {
  alert( i + ": " + item + " (массив:" + arr + ")"
 );
});
```

filter

- Метод «arr.filter(callback[, thisArg])» используется для фильтрации массива через функцию.

- Он создаёт новый массив, в который войдут только те элементы arr, для которых вызов callback(item, i, arr) возвратит true.

```
var arr = [1, -1, 2, -2, 3];

var positiveArr = arr.filter(function(number) {
  return number > 0;
});

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

find

- find() возвращает значение первого найденного в массиве элемента, которое удовлетворяет условию переданному в callback функции. В противном случае возвращается undefined.

- Метод findIndex(), возвращает индекс найденного в массиве элемента вместо его значения.

map

- Метод «arr.map(callback[, thisArg])» используется для трансформации массива.

- Он создаёт новый массив, который будет состоять из результатов вызова callback(item, i, arr) для каждого элемента arr.

```
var names = ['HTML', 'CSS', 'JavaScript'];

var nameLengths = names.map(function(name) {
  return name.length;
});

// получили массив с длинами
alert( nameLengths ); // 4,3,10
```

every/some

- Эти методы используются для проверки массива.

- Метод «arr.every(callback[, thisArg])» возвращает true, если вызов callback вернёт true для каждого элемента arr.
- Метод «arr.some(callback[, thisArg])» возвращает true, если вызов callback вернёт true для какого-нибудь элемента arr.

```
var arr = [1, -1, 2, -2, 3];

function isPositive(number) {
  return number > 0;
}

alert( arr.every(isPositive) );
 // false, не все положительные
alert( arr.some(isPositive) );
 // true, есть хоть одно положительное
```

reduce/reduceRight

- Метод «arr.reduce(callback[, initialValue])» используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата.
- Метод reduce используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива».
- Он применяет функцию callback по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.
- Аргументы функции callback(previousValue, currentItem, index, arr):
- previousValue – последний результат вызова функции, он же «промежуточный результат».
- currentItem – текущий элемент массива, элементы перебираются по очереди слева-направо.
- index – номер текущего элемента.
- arr – обрабатываемый массив.

```
var arr = [1, 2, 3, 4, 5]

// для каждого элемента массива запустить функцию,
// промежуточный результат передавать первым аргументом далее
var result = arr.reduce(function(sum, current) {
  return sum + current;
}, 0);

alert( result ); // 15
```

- Метод arr.reduceRight работает аналогично, но идёт по массиву справа-налево.