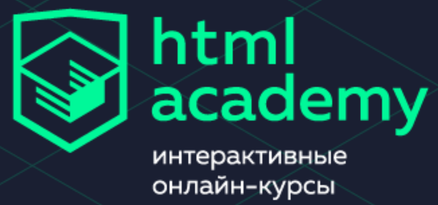


Раздел 4:

HTTP

В этой лекции

– Async/await

– Stream API

– HTTP

– Сервер

Async/await

Конструкции языка, позволяющие привести асинхронный код к синхронному виду. Важно! Что это всего лишь представление кода, сам код остаётся асинхронным по своей природе

Async

Async

– Ключевое слово для функции или метода

Async

– Ключевое слово для функции или метода

– Функция, помеченная как *async* всегда возвращает *Promise*

Async

– Ключевое слово для функции или метода

– Функция, помеченная как *async* всегда возвращает *Promise*

– *Async* функцию никогда не имеет смысла вызывать внутри `*try ... catch*` блока

Await

Await

– Ключевое слово, доступное внутри *async*-функции

Await

– Ключевое слово, доступное внутри *async*-функции

– Позволяет дождаться результата работы *Promise*

Await

- Ключевое слово, доступное внутри *async*-функции
- Позволяет дождаться результата работы *Promise*
- Если в результате *await* работы *Promise* выбросил исключение, то его можно обработать при помощи блока `*try ... catch*`

Особенности

Особенности

– Любая функция, которая возвращает *Promise* работает так же как *async*-функция

Особенности

– Любая функция, которая возвращает *Promise* работает так же как *async*-функция

– При вызове из *async*-функции всегда нужно следить за тем, чтобы обрабатывался результат *Promise*

Особенности

– Любая функция, которая возвращает *Promise* работает так же как *async*-функция

– При вызове из *async*-функции всегда нужно следить за тем, чтобы обрабатывался результат *Promise*

– *await* никогда не замыкается — не может быть вложен в другую функцию (так же как *return*)

Особенности

– *Любая функция, которая возвращает Promise работает так же как async-функция*

– При вызове из *async*-функции всегда нужно следить за тем, чтобы обрабатывался результат *Promise*

– *await* никогда не замыкается — не может быть вложен в другую функцию (так же как *return*)

– конструкторы не могут быть асинхронными

Особенности

– *Любая функция, которая возвращает Promise работает так же как async-функция*

– При вызове из *async*-функции всегда нужно следить за тем, чтобы обрабатывался результат *Promise*

– *await* никогда не замыкается — не может быть вложен в другую функцию (так же как *return*)

– конструкторы не могут быть асинхронными

– геттеры и сеттеры не могут быть асинхронными

Код на *Promise*

```
let exists = false;
mkdir(dir).
    catch((e) => {
      if (e.code === `EEXIST`) {
        exists = true;
        return Promise.resolve();
      }
      return Promise.reject(e);
    }).
    then(() => writeFile(path, `Hello, world!`)).
    then(() => readFile(path)).
    then((data) => console.log(data.toString())).
    then(() => unlink(path)).
    then(() => exists ? Promise.resolve() : rmdir(dir)).
    catch((e) => console.error(e));
```

Код на *async/await*

```
let created = false;
try {
  await mkdir(dir);
  created = true;
} catch (e) {
  if (e.code !== `EEXIST`) {
    throw e;
  }
}

await writeFile(path, `Hello, world!`);
const data = await readFile(path);
console.log(data.toString());
await unlink(path);

if (created) {
  await rmdir(dir);
}
```

Async / await

```
const resolveAfter2Seconds = (x) => {
  return new Promise((resolve) => setTimeout(() => resolve(x), 2000));
};

const add1 = async (x) => {
  const a = resolveAfter2Seconds(20);
  const b = resolveAfter2Seconds(30);
  return x + await a + await b;
};

add1(10).then(v => console.log(v)); // 60 через 2 сек.

const add2 = async (x) => {
  const a = await resolveAfter2Seconds(20);
  const b = await resolveAfter2Seconds(30);
  return x + a + b;
};

add2(10).then(v => console.log(v)); // 60 через 4 сек.
```

Async / await

```
const resolveAfter2Seconds = (x) => {
  return new Promise((resolve) => setTimeout(() => resolve(x), 2000));
};

const iterate = async (num) => {
  let sum = 0;
  for (let i = num; i--;) {
    sum += await resolveAfter2Seconds(10);
  }
  return sum;
};

console.time(`iterate`);
iterate(5).then(() => console.timeEnd(`iterate`)); // iterate: ???
```

Async / await

```
const resolveAfter2Seconds = (x) => {
  return new Promise((resolve) => setTimeout(() => resolve(x), 2000));
};

const iterate = async (num) => {
  let sum = 0;
  for (let i = num; i--;) {
    sum += await resolveAfter2Seconds(10);
  }
  return sum;
};

console.time(`iterate`);
iterate(5).then(() => console.timeEnd(`iterate`)); // iterate: ~10s
```

HTTP 1.1

протокол передачи данных. Описывает в каком виде обмениваются информацией клиент и сервер

HTTP 1.1 Запрос

```
GET /index.php HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

HTTP 1.1 Ответ

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1270
Cache-Control: max-age=604800
Date: Tue, 24 Oct 2017 11:08:24 GMT
Etag: "359670651+ident"
Expires: Tue, 31 Oct 2017 11:08:24 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (dca/53DB)
Vary: Accept-Encoding
X-Cache: HIT

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
```

HTTP 1.1

HTTP 1.1

– строка запроса / строка статуса (*number*, *string*)

```
GET /data.json HTTP/1.1
HTTP/1.1 200 OK
```

HTTP 1.1

– строка запроса / строка статуса (*number*, *string*)

```
GET /data.json HTTP/1.1
HTTP/1.1 200 OK
```

– строки заголовков в виде пар ключ-значение (*object*)

```
Content-Type: application/json
```

HTTP 1.1

– строка запроса / строка статуса (*number*, *string*)

```
GET /data.json HTTP/1.1
HTTP/1.1 200 OK
```

– строки заголовков в виде пар ключ-значение (*object*)

```
Content-Type: application/json
```

– пустая строка (???)

HTTP 1.1

– строка запроса / строка статуса (*number*, *string*)

```
GET /data.json HTTP/1.1
HTTP/1.1 200 OK
```

– строки заголовков в виде пар ключ-значение (*object*)

```
Content-Type: application/json
```

– пустая строка (???)

– тело (*Stream*)

HTTP методы чтения

HTTP методы чтения

– GET — запрос на получение информации с сервера

HTTP методы чтения

– `GET` — запрос на получение информации с сервера

– `HEAD` — запрос для проверки, обновилась ли информация на сервере и стоит ли заново её скачать или можно оставить закешированную версию

HTTP методы чтения

– `GET` — запрос на получение информации с сервера

– `HEAD` — запрос для проверки, обновилась ли информация на сервере и стоит ли заново её скачать или можно оставить закешированную версию

– `OPTIONS` — запрос для проверки, какие запросы можно делать на этот ресурс

HTTP методы записи

HTTP методы записи

– POST — запрос на создание новой записи на сервере

HTTP методы записи

– POST — запрос на создание новой записи на сервере

– PUT — запрос на перезапись существующей информации на сервере

HTTP методы записи

– POST — запрос на создание новой записи на сервере

– PUT — запрос на перезапись существующей информации на сервере

– DELETE — запрос на удаление существующей информации на сервере

HTTP методы записи

– POST — запрос на создание новой записи на сервере

– PUT — запрос на перезапись существующей информации на сервере

– DELETE — запрос на удаление существующей информации на сервере

– PATCH — запрос на частичную перезапись существующей информации на сервере

Коды состояния HTTP

– 1xx — информационные сообщения

Коды состояния HTTP

- 1xx — информационные сообщения
- 2xx — успешные сообщения

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда
  - `307 Temporary Redirect` — ресурс переехал временно

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда
  - `307 Temporary Redirect` — ресурс переехал временно
- `4xx` — ошибка в запросе клиента

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда
  - `307 Temporary Redirect` — ресурс переехал временно
- `4xx` — ошибка в запросе клиента
  - `400 Bad Request` — неправильный запрос

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда
  - `307 Temporary Redirect` — ресурс переехал временно
- `4xx` — ошибка в запросе клиента
  - `400 Bad Request` — неправильный запрос
  - `404 Not Found` — запрашиваемый ресурс не найден

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда
  - `307 Temporary Redirect` — ресурс переехал временно
- `4xx` — ошибка в запросе клиента
  - `400 Bad Request` — неправильный запрос
  - `404 Not Found` — запрашиваемый ресурс не найден
- `5xx` — ошибки сервера

Коды состояния HTTP

- `1xx` — информационные сообщения
- `2xx` — успешные сообщения
  - `200 OK` — все хорошо, можно продолжать
- `3xx` — перенаправление
  - `301 Moved Permanently` — ресурс переехал навсегда
  - `307 Temporary Redirect` — ресурс переехал временно
- `4xx` — ошибка в запросе клиента
  - `400 Bad Request` — неправильный запрос
  - `404 Not Found` — запрашиваемый ресурс не найден
- `5xx` — ошибки сервера
  - `500 Internal Server Error` — произошла внутренняя ошибка

Stream

Паттерн, который позволяет читать данные по частям

Stream extends EventEmitter

Stream extends EventEmitter

– Stream является EventEmitter с фиксированным кол-вом событий

Stream extends EventEmitter

– Stream является EventEmitter с фиксированным кол-вом событий

– Stream бывает четырёх-видов:

Stream extends EventEmitter

- Stream является EventEmitter с фиксированным кол-вом событий
- Stream бывает четырёх-видов:
  - Write (в который можно только писать)

Stream extends EventEmitter

- Stream является EventEmitter с фиксированным кол-вом событий
- Stream бывает четырёх-видов:
  - Write (в который можно только писать)
  - Read (из которого можно только читать)

Stream extends EventEmitter

- Stream является EventEmitter с фиксированным кол-вом событий
- Stream бывает четырёх-видов:
  - Write (в который можно только писать)
  - Read (из которого можно только читать)
  - Duplex (в который можно и читать и писать)

Stream extends EventEmitter

- Stream является EventEmitter с фиксированным кол-вом событий
- Stream бывает четырёх-видов:
  - Write (в который можно только писать)
  - Read (из которого можно только читать)
  - Duplex (в который можно и читать и писать)
  - Transform (который трансформирует данные на лету)

Read Stream

Read Stream

– Поток из которого можно читать данные

Read Stream

– Поток из которого можно читать данные

– Основные события:

Read Stream

- Поток из которого можно читать данные
- Основные события:
  - *data* — часть данных загружена и готова для чтения

Read Stream

- Поток из которого можно читать данные
- Основные события:
  - *data* — часть данных загружена и готова для чтения
  - *end* — данные закончились

Read Stream

- Поток из которого можно читать данные
- Основные события:
  - *data* — часть данных загружена и готова для чтения
  - *end* — данные закончились
  - *error* — при чтении данных произошла ошибка

Read Stream

- Поток из которого можно читать данные
- Основные события:
  - *data* — часть данных загружена и готова для чтения
  - *end* — данные закончились
  - *error* — при чтении данных произошла ошибка
- Основные методы:

Read Stream

- Поток из которого можно читать данные
- Основные события:
  - *data* — часть данных загружена и готова для чтения
  - *end* — данные закончились
  - *error* — при чтении данных произошла ошибка
- Основные методы:
  - *pipe(destination, options)* — позволяет перенаправить вывод в *Write Stream*

Write Stream

Write Stream

– Поток в который можно писать данные

Write Stream

– Поток в который можно писать данные

– Основные события:

Write Stream

- Поток в который можно писать данные
- Основные события:
  - *drain* — сообщает, что переданные данные были успешно записаны и можно продолжить писать данные

Write Stream

- Поток в который можно писать данные
- Основные события:
  - *drain* — сообщает, что переданные данные были успешно записаны и можно продолжить писать данные
  - *error* — при чтении данных произошла ошибка

Write Stream

- Поток в который можно писать данные
- Основные события:
  - *drain* — сообщает, что переданные данные были успешно записаны и можно продолжить писать данные
  - *error* — при чтении данных произошла ошибка
- Основные методы:

Write Stream

- Поток в который можно писать данные
- Основные события:
  - *drain* — сообщает, что переданные данные были успешно записаны и можно продолжить писать данные
  - *error* — при чтении данных произошла ошибка
- Основные методы:
  - *write(chunk, encoding, callback)* — позволяет записать часть данных

Write Stream

- Поток в который можно писать данные
- Основные события:
  - *drain* — сообщает, что переданные данные были успешно записаны и можно продолжить писать данные
  - *error* — при чтении данных произошла ошибка
- Основные методы:
  - *write(chunk, encoding, callback)* — позволяет записать часть данных
  - *end(chunk, encoding, callback)* — возвращает последнюю часть данных и сообщает о том, что данные закончились

Для того чтобы читать и писать данные нужно понимать, что за данные мы читаем или пишем

Как понять какие данные мы читаем?

Как понять какие данные мы читаем?

– По имени файла

Как понять какие данные мы читаем?

– По имени файла

– По содержимому

Как понять какие данные мы читаем?

– По имени файла

– По содержимому

– Попробовать угадать

Как понять какие данные мы читаем?

– По имени файла

– По содержимому

– Попробовать угадать

– Откуда-то знать заранее

Media Type (MIME Type/Content Type)

Специальный формат для указания типа данных.

Изначально появился для указания типа вложенных файлов в электронных письмах (Multipurpose Internet Mail Extensions)

Типы данных (media types)

- application/json
- image/jpeg
- application/x-www-form-urlencoded
- multipart/form-data
- application/octet-stream
- миллион других =)

www.iana.org/assignments/media-types/media-types.xhtml

MIME Type

developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

MIME Type

top-level type name / subtype name

Имя класса данных

Имя типа данных

developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

MIME Type

developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

MIME Type

image / jpeg

Картинка В формате JPEG

developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

MIME Type

developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

MIME Type

text / html

Текст Формат HTML

developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

Сервер

Запуск сервера

Запуск сервера

– Создать сервер:

Запуск сервера

- Создать сервер:
  - *http.createServer((req, res) => res.end());*

Запуск сервера

- Создать сервер:
  - *http.createServer((req, res) => res.end());*
- Запустить сервер:

Запуск сервера

- Создать сервер:
  - *http.createServer((req, res) => res.end());*
- Запустить сервер:
  - *http.listen(3000);*

Node server

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Примечания

Примечания

– По умолчанию порт для работы с http — это порт 80, т.е. если в адресе не указан порт, то браузер по умолчанию будет использовать порт 80 для HTTP и порт 443 для HTTPS

Примечания

- По умолчанию порт для работы с http — это порт 80, т.е. если в адресе не указан порт, то браузер по умолчанию будет использовать порт 80 для HTTP и порт 443 для HTTPS
- На некоторых операционных системах порты (*nix) <2000 запрещены для использования с правами обычного пользователя

http.IncomingMessage — Объект запроса

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.IncomingMessage — Объект запроса

– Объект реализует все методы и события Read Stream

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.IncomingMessage — Объект запроса

– Объект реализует все методы и события Read Stream

– *message.headers* — заголовки запроса (пары ключ-значение)

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.IncomingMessage — Объект запроса

– Объект реализует все методы и события Read Stream

– *message.headers* — заголовки запроса (пары ключ-значение)

– *message.httpVersion* — версия протокола HTTP, которую запросил клиент

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.IncomingMessage — Объект запроса

– Объект реализует все методы и события Read Stream

– *message.headers* — заголовки запроса (пары ключ-значение)

– *message.httpVersion* — версия протокола HTTP, которую запросил клиент

– *message.method* — метод, по которому обратился клиент

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.IncomingMessage — Объект запроса

- Объект реализует все методы и события Read Stream
- *message.headers* — заголовки запроса (пары ключ-значение)
- *message.httpVersion* — версия протокола HTTP, которую запросил клиент
- *message.method* — метод, по которому обратился клиент
- *message.url* — локальный адрес который запросил клиент (например, *'/status?name=ryan'*)

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.IncomingMessage — Объект запроса

- Объект реализует все методы и события Read Stream
- *message.headers* — заголовки запроса (пары ключ-значение)
- *message.httpVersion* — версия протокола HTTP, которую запросил клиент
- *message.method* — метод, по которому обратился клиент
- *message.url* — локальный адрес который запросил клиент (например, *'/status?name=ryan'*)
  - для работы с *URL* есть модуль *url*, который позволяет распарсить *URL* из строки в объект и обратно

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_incomingmessage

http.ServerResponse — объект ответа

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse

http.ServerResponse — объект ответа

– Объект реализует все методы и события *Write Stream*

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse

http.ServerResponse — объект ответа

– Объект реализует все методы и события *Write Stream*

– *response.statusCode* — поле, хранит код ответа (например: 200, 404, 501)

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse

http.ServerResponse — объект ответа

– Объект реализует все методы и события *Write Stream*

– *response.statusCode* — поле, хранит код ответа (например: 200, 404, 501)

– *response.statusMessage* — поле, хранит сообщение ответ (например: 'OK', 'Not Found')

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse

http.ServerResponse — объект ответа

– Объект реализует все методы и события *Write Stream*

– *response.statusCode* — поле, хранит код ответа (например: 200, 404, 501)

– *response.statusMessage* — поле, хранит сообщение ответ (например: 'OK', 'Not Found')

– *response.writeHead()* — метод, который отправляет заголовок

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse

http.ServerResponse — объект ответа

– Объект реализует все методы и события *Write Stream*

– *response.statusCode* — поле, хранит код ответа (например: 200, 404, 501)

– *response.statusMessage* — поле, хранит сообщение ответ (например: 'OK', 'Not Found')

– *response.writeHead()* — метод, который отправляет заголовок

– *response.write()* — метод, который отправляет часть тела ответа

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse

http.ServerResponse — объект ответа

– Объект реализует все методы и события *Write Stream*

– *response.statusCode* — поле, хранит код ответа (например: 200, 404, 501)

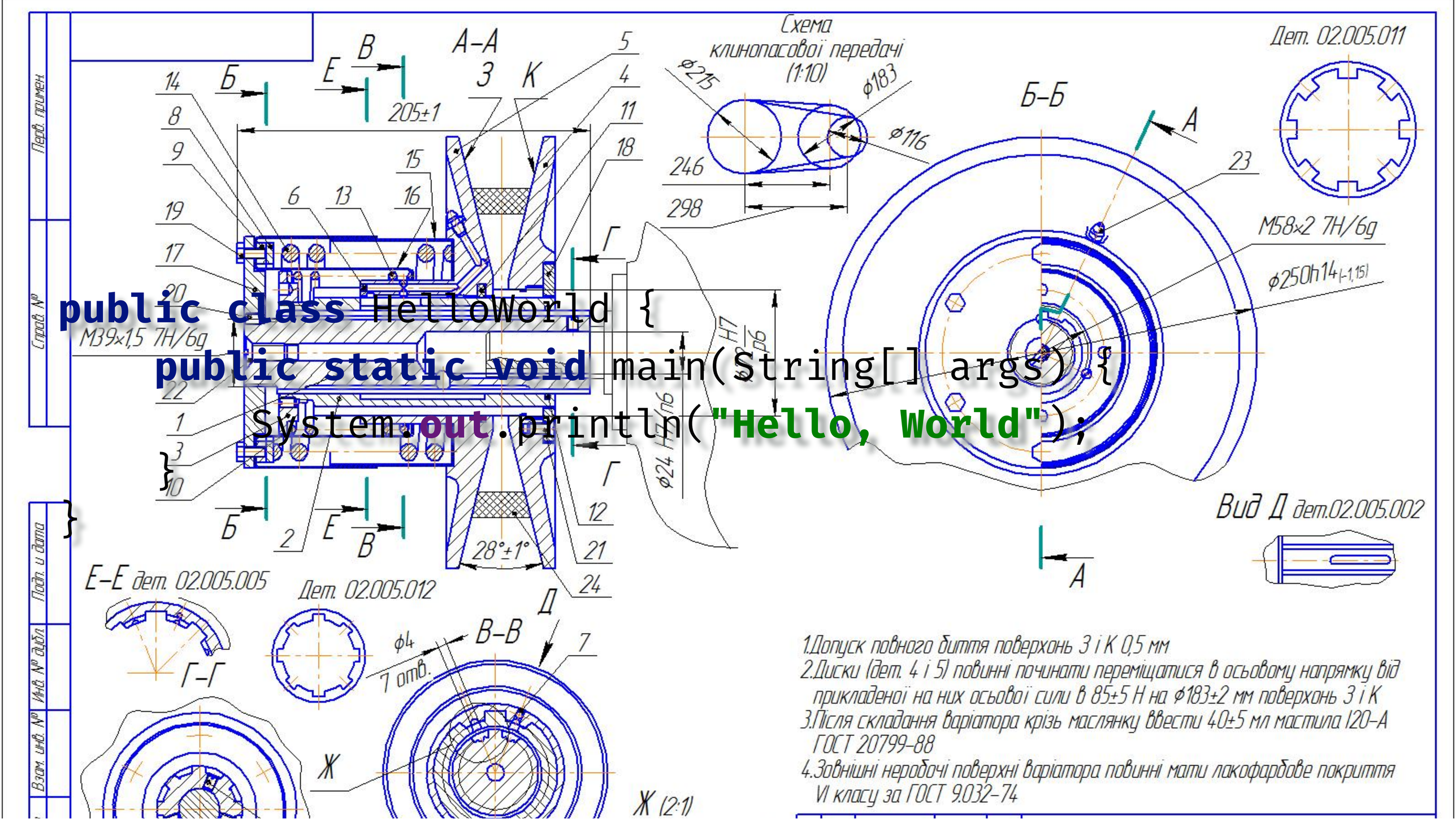
– *response.statusMessage* — поле, хранит сообщение ответ (например: 'OK', 'Not Found')

– *response.writeHead()* — метод, который отправляет заголовок

– *response.write()* — метод, который отправляет часть тела ответа

– *response.end(data, encoding, callback)* — метод, который сообщает, что ответ записан

nodejs.org/dist/latest-v9.x/docs/api/http.html#http_class_http_serverresponse




```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```