

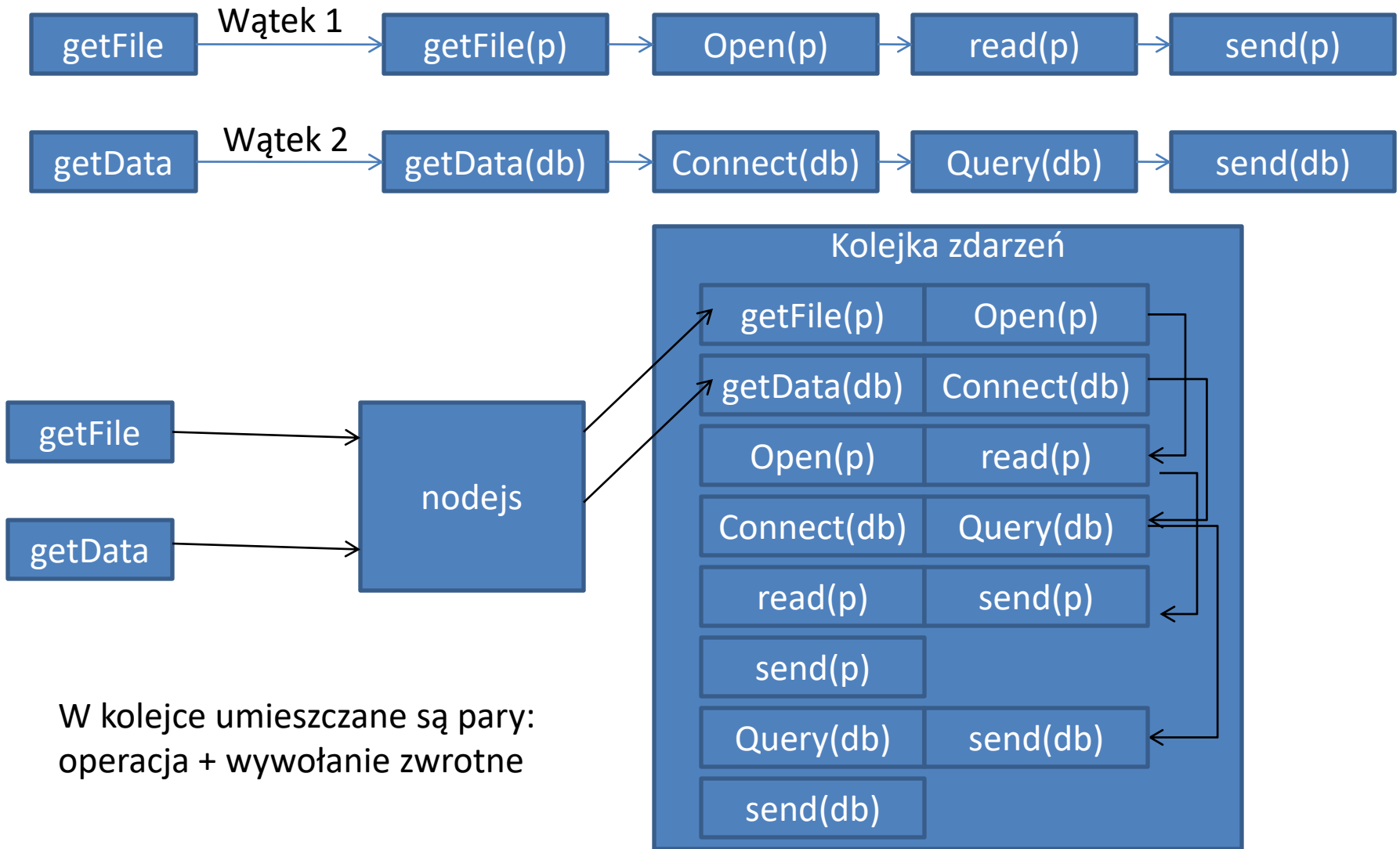
# **Programowanie z wykorzystaniem języków skryptowych**

Marcin Bernaś

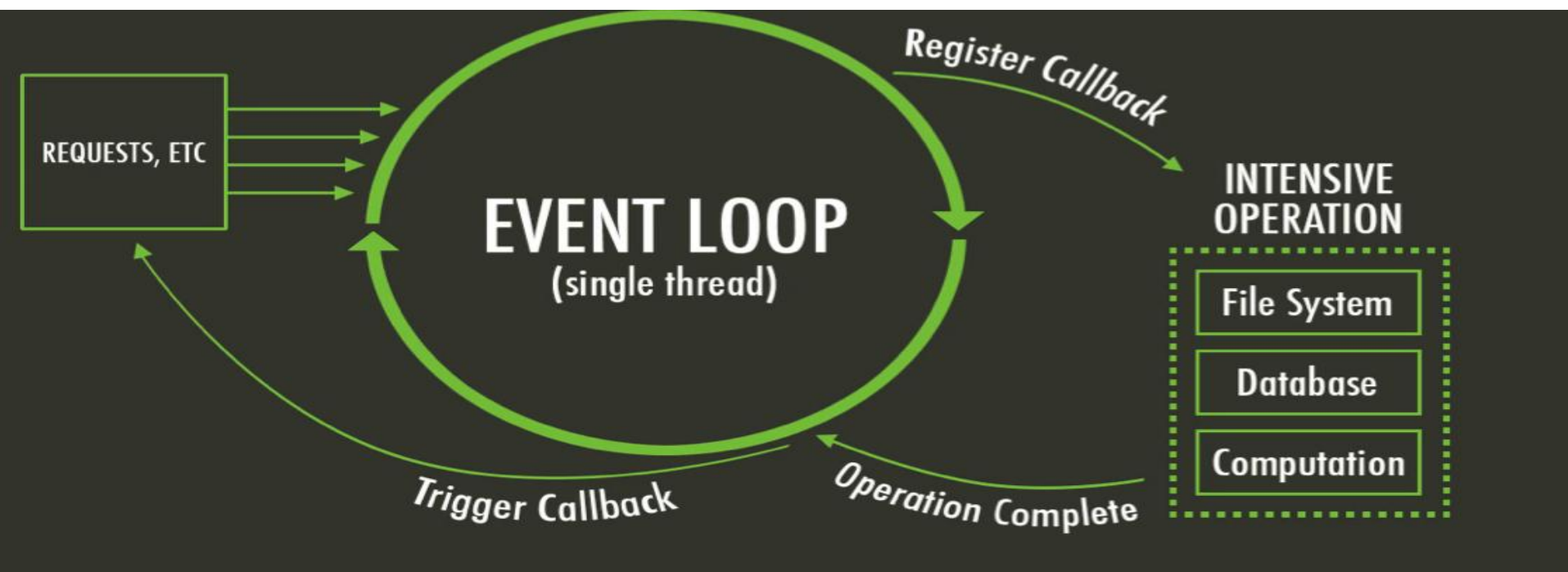
# Idea powstania node.js....

- Wykonywanie asynchronicznego przetwarzania w jednym wątku zamiast klasycznego wielowątkowego przetwarzania co zmniejsza narzut i opóźnienia (zwiększania skalowalność)
- Skalowanie płaskie zamiast hierarchicznego
- Idealne do zastosowań w aplikacjach wymagających mnóstwo żądań bez skomplikowanych obliczeń matematycznych
- W przypadku skomplikowanych obliczeń nie korzysta z dobrodziejstw przetwarzania równoległego
- Mniejsze kłopoty z zrównolegleniem procesów

# Model wywołań zwrotnych vs wątkowość



# Node.js pętla zdarzeń



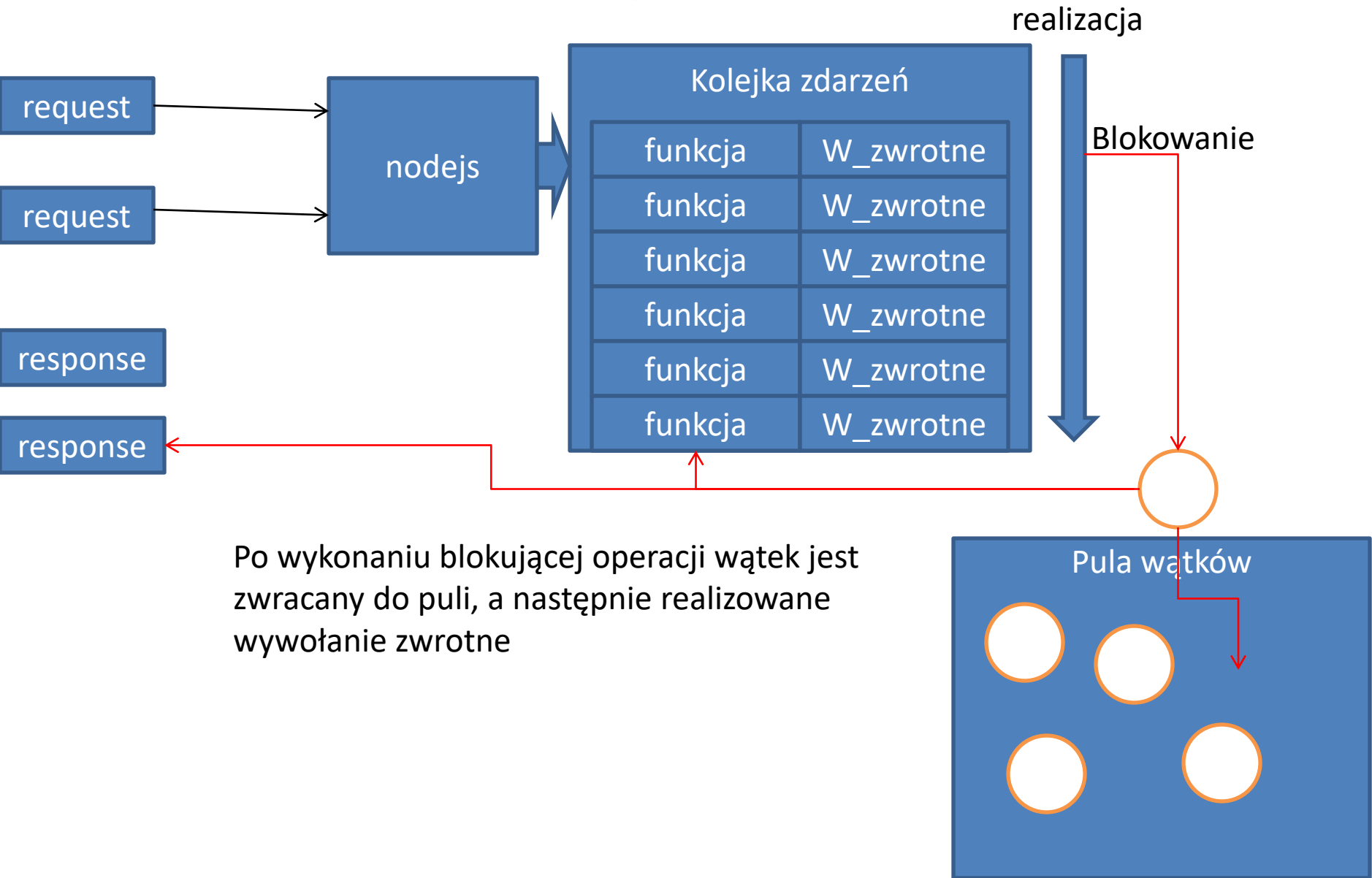
Zastosowany model wymaga określonego podejścia do tworzenia kodu:

- Unikanie synchronicznego kodu w celu zapobiegania blokowania kodu
- Co objawia się przez zastosowanie funkcji zwrotnych

# Blokujące We-Wy

- Model sprawdza się dopóki I/O nie są blokowane:
  - Odczytywanie pliku
  - Odpytywanie bazy danych
  - Żądanie gniazda
  - Uzyskanie dostępu do usługi zdalnej

# Obsługa wątków



# Blokowanie a nieblokowane żądania

Przykład :: Czytanie i wyświetlanie informacji

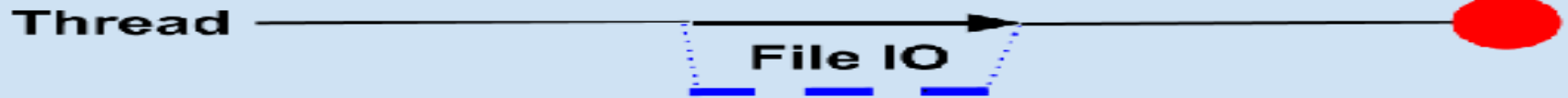
## Synchronous I/O

Thread waits during I/O operation



## Asynchronous I/O

Thread DON'T wait during I/O operation



## Przykład blokowania

- Odczyt w pliku
- Operacja zależna - wyświetlanie
- Pozostałe operacje

```
var data = fs.readFileSync( "test.txt" );  
console.log( data );  
console.log( "Do other tasks" );
```



# Poprawne podejście



Wywołanie zwrotne

- Wczytaj dane z pliku

Wczytanie ukończone -> Wyświetl dane

- Inne operacje

```
fs.readFile( "test.txt", function( err, data ) {  
  console.log(data);  
});  
Console.log(„other activities”);
```

# Planowanie modelu zdarzeń

- Utworzenie wywołania do jednego z wywołań biblioteki blokującego we-wy (pliki/baza danych)
- Dodawanie procesu nasłuchiwanie zdarzeń (`http.request` / `server.connection`)
- Tworzenie emiterów zdarzeń
- Opcja `process.nextTick` opcja do pobrania w następnym cyklu
- Zastosowanie liczników

# Implementowanie liczników

- Limit czasu – wykonanie zdarzenia po określonym czasie (zabezpieczenia)
- Interwał – realizacja funkcji co pewien czas
- Natychmiastowy – dodanie akcji na początek pętli zdarzeń do wykonania ASAP

# Limit czasu

- Metoda:

`setTimeout(wyw_zwrotne, opóźnienie (ms),[argumenty])`

przykład:

`myTimeout=setTimeout(myFun,1000);` - wywoła funkcję po sekundzie

`clearTimeout(myTimeout);`

# Przykład:

```
function czE(nazwa){return console.timeEnd(nazwa)};  
function czB(nazwa){return console.time(nazwa)};  
czB(„a”); czB(„b”); czB(„c”); czB(„c”);  
setTimeout(czE,2000,„a”);  
  setTimeout(czE,1000,„b”);  
setTimeout(czE,500,„c”);  
setTimeout(czE,10,„d”);
```

wynik?

# Interwał

- Metoda:

`setInterval(wyw_zwrotne, opóźnienie (ms),[argumenty])`

przykład:

`myInterval=setInterval(myFun,1000);` - wywoła funkcję co sekundę

`clearTimeout(myInterval);`

Funkcję:

`myinterval.ref()` - dodanie do pętli zdarzeń

`Myinterval.unref()` - usunięcie z pętli zdarzeń

# Przykład:

```
var x=0, y=0, z=0;
function updateX(){
  x+=1;};
function updateY(){
  y+=1;};
function updateZ(){
  z+=1;};
function disp(){
  console.log(x+", "+y+", "+z);
}
setInterval(updateX,500);
setInterval(updateY,1000);
setInterval(updateZ,100);
setInterval(disp,1000);
```

wynik?

```
C:\proj_js>node skrypt2.js
1,1,9
3,2,19
6,3,29
8,4,39
10,5,49
12,6,59
14,7,69
16,8,79
18,9,89
20,10,99
22,11,109
24,12,119
26,13,129
28,14,139
```

# Licznik natychmiastowy

- Metoda:

`setInterval(wyw_zwrotne,[argumenty])`

przykład:

`myInterval=setInterval(myFun);` - wywoła funkcję zaraz po  
zakończeniu operacji We-Wy, prze innymi licznikami

`clearTimeout(myInterval);`



# Funkcja nextTick

- Metoda:  
`process.nextTick(wywołanie_zwrotne)`
- W odróżnieniu od poprzedniego, wywołany przed aktywowaniem zdarzeń we-wy.
- Ze względu na pierwszeństwo ograniczono liczbę w jednym cyklu do (1000). Ustawiane:  
`process.maxTickDepth`

# Przykład

```
var fs = require("fs");

fs.stat("skrypt3.js",function(err,stats){
  if (stats) {console.log("plik ok");}
});

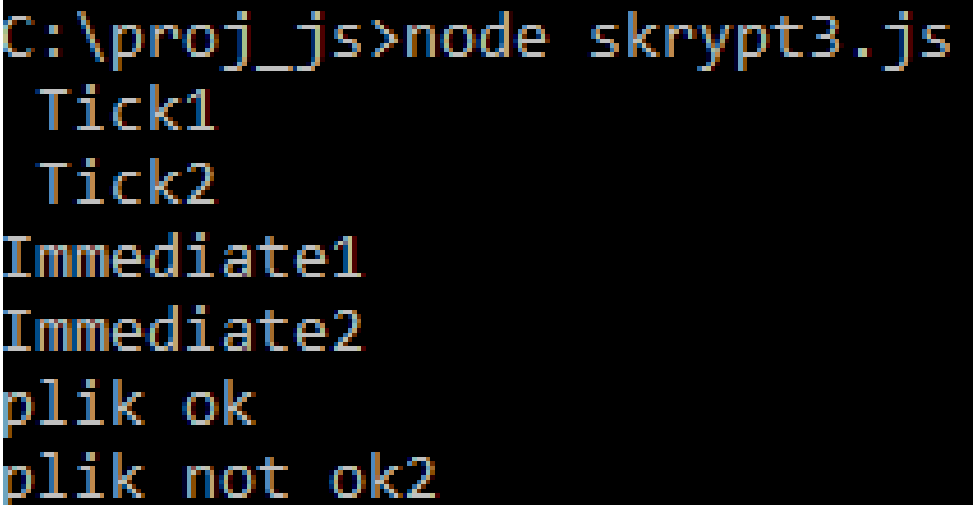
fs.stat("skryptx.js",function(err,stats){
  if (stats) console.log("plik ok2");
  else console.log("plik not ok2");
});

setImmediate(function(){
  console.log("Immediate1");
});

setImmediate(function(){
  console.log("Immediate2");
});

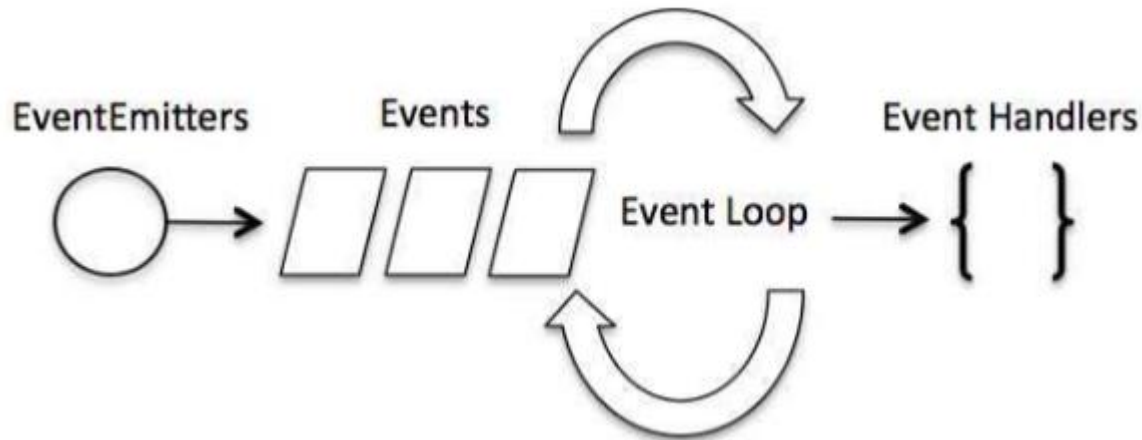
process.nextTick(function(){
  console.log(" Tick1");
});

process.nextTick(function(){
  console.log(" Tick2");
});
```

A terminal window with a black background and multi-colored text (cyan, yellow, red). It shows the command 'C:\proj\_js>node skrypt3.js' and its output: 'Tick1', 'Tick2', 'Immediate1', 'Immediate2', 'plik ok', and 'plik not ok2'.

```
C:\proj_js>node skrypt3.js
Tick1
Tick2
Immediate1
Immediate2
plik ok
plik not ok2
```

# Emiterzy zdarzeń



- Emitery wchodzą w skład obiektu EventEmitter
- Zdarzenie emitowane jest poprzez funkcję `emit(zdarzenie,[argumenty]);`
- Obsługę definiujemy poprzez funkcję `.on(zdarzenie, funkcja)`
- Funkcja `.once` reaguje tylko jednokrotnie
- Zdarzenie to najczęściej tekst: „bum;)”

# Przykład

```
var events = require('events');  
var EventEmitter = new events.EventEmitter();  
  
//Create an event handler:  
var myEventHandler = function () {  
  console.log('I hear a scream!');  
}  
  
//Assign the event handler to an event:  
eventEmitter.on('scream', myEventHandler);  
  
//Fire the 'scream' event:  
eventEmitter.emit('scream');
```

# Info

- Dostępne funkcje: `listeners(nazwa)` – zwraca tablicę funkcji powiązanych ze zdarzeniem
- `removeListener(nazwa)` – usuwa zdarzenie
- `setMaxListeners(n)` – ostrzeżenie o przekroczeniu (10)
- Możliwe dołączanie emitera do własnych obiektów: konieczne dziedziczenie:

```
Myobj.prototype.__proto__ =  
events.EventEmitter.prototype;
```

# Przekazywanie parametrów

- Funkcja anonimowa: to funkcja która nie ma nazwy:

- Wywołanie:

`oblicz(nazwa_funkcji);`

Lub

`Oblicz(function(){nazwa_funkcji(par1,par2);});`

- Jeżeli chcemy przesłać parametry dodatkowe, możemy to zrealizować poprzez funkcję anonimową

# Przykład

```
01 var events = require('events');
02 function CarShow() {
03   events.EventEmitter.call(this);
04   this.seeCar = function(make){
05     this.emit('sawCar', make);
06   };
07 }
08 CarShow.prototype.__proto__ = events.EventEmitter.prototype;
09 var show = new CarShow();
10 function logCar(make){
11   console.log("Widziano samochód. Marka: " + make);
12 }
13 function logColorCar(make, color){
14   console.log("Widziano samochód. Kolor: %s Marka: %s", color, make);
15 }
16 show.on("sawCar", logCar);
17 show.on("sawCar", function(make){
18   var colors = ['czerwony', 'niebieski', 'czarny'];
19   var color = colors[Math.floor(Math.random()*3)];
20   logColorCar(make, color);
21 });
22 show.seeCar("Ferrari");
23 show.seeCar("Porsche");
24 show.seeCar("Bugatti");
25 show.seeCar("Lamborghini");
26 show.seeCar("Aston Martin");
```

# Domknięcie

```
function logCar(logMsg,callback){
  process.nextTick(function(){
    callback(logMsg)});
}
```

```
var owoce=["gruszka","sliwka","granat"];
for (var idx in owoce){
  var message = "Zjadłem dziś:" + owoce[idx];
  logCar(message,function(){
    console.log("Wywołanie zwrotne:"+message);
  });
}
```

```
var owoce=["gruszka","sliwka","granat"];
for (var idx in owoce){
  var message = "Zjadłem dziś:" + owoce[idx];
  (function(msg){
    logCar(msg,function(){
      console.log("Wywołanie zwrotne:"+msg);
    });
  })(message);
}
```

**Domknięcia to funkcje których funkcje wewnętrzne odwołują się do niezależnych (wolnych) zmiennych. Innymi słowy, funkcje zadeklarowane wewnątrz domknięcia 'pamiętają' środowisko w którym zostały utworzone.**

```
C:\proj_js>node skrypt5.js
Wywołanie zwrotne:Zjadłem dziś:granat
Wywołanie zwrotne:Zjadłem dziś:granat
Wywołanie zwrotne:Zjadłem dziś:granat
Wywołanie z domknięciem:Zjadłem dziś:gruszka
Wywołanie z domknięciem:Zjadłem dziś:sliwka
Wywołanie z domknięciem:Zjadłem dziś:granat
```



# Łączenie wywołań zwrotnych

- Przetwarzanie asynchroniczne nie gwarantuje kolejności wykonania. Aby je zapewnić (dla jednej funkcji), można skorzystać z wywołań zwrotnych.
- Rozwiązanie przypomina rekurencję:
  - Funkcja sama się wywołuje
  - Warunek stopu

# Przykład

```
function zjadamy(owoc,callback){
    console.log("zjadlem:"+owoc);
    if (owoce.length){
        process.nextTick(function(){
            callback();
        });
    }
}
```

```
function uczta(owoce){
    var owoc = owoce.pop();
    zjadamy(owoc,function(){
        uczta(owoce);
    });
}
```

```
var owoce = ["gruszka","sliwka","ananas","awokado","wisnia"];
uczta(owoce);
```

---

```
var owoce2 = ["gruszka","sliwka","ananas","awokado","wisnia"];
uczta(owoce2);
```

```
C:\proj_js>node skrypt6.js
zjadlem:wisnia
zjadlem:awokado
zjadlem:ananas
zjadlem:sliwka
zjadlem:gruszka
```

# Obsługa danych

- Przekształcanie danych w plikach JSON
- Dwie funkcje:
- JSONstr=„{name:mama,numer:1234}”;
- Var obj = JSON.parse(JSONstr);
- Użycie: obj.mama , obj.numer ...
- obj.dod=„info”;
- JSONstr = JSON.stringify(obj);

# Buffer

- Przetwarzanie danych w blokach
- Sposoby tworzenia:
  - `var buf = new Buffer(10); //10 bajtów`
  - `var buf = new Buffer([10, 20, 30, 40, 50]);`
  - `var buf = new Buffer("Simply Easy Learning", "utf-8");`
- Zapisywanie / czytanie z buforu:
- `buf.write(string[, offset][, length][, encoding])`
- `buf.toString([encoding][, start][, end])`

# Wczytywanie przykład

```
buf = new Buffer(26);  
for (var i = 0 ; i < 26 ; i++) {  
  buf[i] = i + 97;  
}
```

```
console.log( buf.toString('ascii'));  
// outputs: abcdefghijklmnopqrstuvwxyz  
console.log( buf.toString('ascii',0,5));  
// outputs: abcde  
console.log( buf.toString('utf8',0,5));  
// outputs: abcde  
console.log( buf.toString(undefined,0,5));  
// encoding defaults to 'utf8', outputs abcde
```

# Przykłady zastosowań

- Do operacji na blokach tekstowych przed wysłaniem
- `.toJSON(buf)` – format JSON (tablica)
- `.concat` – dodawanie buforów
- `.compare` – porównanie
- `.copy` – kopiowanie elementów
- `.slice` – wycięcie kawałka buforu

# Ciągi

- Ciągi (Streams) to obiekty umożliwiające wczytywanie i zapisywanie danych w sposób ciągły. Nodejs definiuje 4 typy ciągów
- Readable – ciągi do odczytu.
- Writable – ciągi umożliwiające zapis.
- Duplex – ciągi w których możliwy jest zarówno odczyt jak i zapis.
- Transform – Typ Duplex, który jest wykorzystywany do modyfikacji danych w trakcie przesyłania.

- Każdy typ jest instancją EventEmitter oraz ma zaimplementowane następujące zdarzenia:
- data – zdarzenie jest emitowane w przypadku danych w buforze,
- end – zdarzenie jest emitowane w przypadku pustego buforu,
- error – zdarzenie jest emitowane w przypadku błędu odczytu lub zapisu,
- finish – zdarzenie jest emitowane w przypadku zakończenia transmisji (flush)



# Read stream

```
var fs = require("fs");
var data = "";
var readerStream = fs.createReadStream('input.txt');
readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
    data += chunk;
});

readerStream.on('end',function(){
    console.log(data);
});

readerStream.on('error', function(err){
    console.log(err.stack);
});

console.log("Program Ended");
```

# Write stream

```
var fs = require("fs");
var data = 'Simply Easy Learning';
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
writerStream.write(data, 'UTF8');
// Mark the end of file
writerStream.end();

// Handle stream events --> finish, and error
writerStream.on('finish', function() {
  console.log("Write completed.");
});

writerStream.on('error', function(err){
  console.log(err.stack);
});

console.log("Program Ended");
```

# Kierowanie strumieni

Pipe umożliwia przekierowanie Wyjścia jednego strumienia na wejście innego

```
var fs = require("fs");  
var readerStream = fs.createReadStream('input.txt');  
var writerStream = fs.createWriteStream('output.txt');  
  
// read input.txt and write data to output.txt  
readerStream.pipe(writerStream);  
  
console.log("Program Ended");
```

# Strumie DUPLEX

- Umożliwiają zapis i odczyt
- Przykładem jest gniazdo TCP/IP
- Możliwe jest tworzenie własnych strumieni Duplex poprzez prototypowanie:
- `Var util = require('util');`
- `Util.inherits(MyDuplexStream,stream.Duplex);`
- Obiekt z `stream.Duplex.call(this,opt);`
- Zaimplementować: `_write` / `_read`

# Strumienie Transform

- Umożliwia korzystanie ze strumieni umożliwiających kodowanie. Przykład:

```
var fs = require("fs");  
var zlib = require('zlib');
```

```
// Decompress the file input.txt.gz to input.txt  
fs.createReadStream('input.txt.gz')  
  .pipe(zlib.createGunzip())  
  .pipe(fs.createWriteStream('input.txt'));
```

```
console.log("File Decompressed.");
```

# Obsługa Plików

```
var fs = require("fs");  
// Asynchronous read  
fs.readFile('input.txt', function (err, data) {  
  if (err) {  
    return console.error(err);  
  }  
  console.log("Asynchronous read: " + data.toString());  
});  
// Synchronous read  
var data = fs.readFileSync('input.txt');  
console.log("Synchronous read: " + data.toString());  
  
console.log("Program Ended");
```

# Rozbicie na wiele zdarzeń

- Kaskada open->read->close

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, function(err,
bytes){
  if (err){
    console.log(err);
  }

  // Print only read bytes to avoid junk.
  if(bytes > 0){
    console.log(buf.slice(0, bytes).toString());
  }

  // Close the opened file.
  fs.close(fd, function(err){
    if (err){
      console.log(err);
    }
    console.log("File closed successfully.");
  });
});
});
```

# Obstuga stdin

```
var stdin = process.stdin;
```

```
// without this, we would only get streams  
once enter is pressed  
stdin.setRawMode( true );
```

```
// resume stdin in the parent process (node  
app won't quit all by itself  
// unless an error or process.exit() happens)  
stdin.resume();
```

```
// i don't want binary, do you?  
stdin.setEncoding( 'utf8' );
```

```
// on any data into stdin  
var ciag = "";  
stdin.on( 'data', function( key ){  
  // ctrl-c ( end of text )  
  if ( key === '\u0003' ) {  
    ciag="";  
  }else{  
    // write the key to stdout all normal like  
    ciag+=key;  
    console.log( ciag +"\n" ); }  
});
```



# Pozostałe

- Zmienne globalne:
  - `__filename` - nazwa pliku wykonanego
  - `__dirname` – ścieżka
- Dodatkowe moduły globalne:
  - OS Module - udostępnia funkcje specyficzne dla danego OS.
  - Path Module - umożliwia przetwarzanie ścieżek - normalizacja.
  - Net Module – udostępnia definiowanie strumieni dla klient-serwer.
  - DNS Module – umożliwia obsługę serwera DNS (nslookup).
  - Domain Module – umożliwia zarządzanie operacjami I/O (grupowanie)
  - Process Module – umożliwia zarządzania procesami potomnymi, uruchamianie, rozgałęzianie, ...

# Następny wykład

- Rozwiązania WEB