Web Frame Work Using JavaScript

Vivek Sharma

WELCOME

Today's Overview

1. • Core Node JS
2. • Adding Workflow items to Node.js
3. • Considerations for the Mobile Development.
4. • Types of Mobile Apps

Learning Node

Node JS Philosophy:

- Small Core
- Small modules
- Small surface area.
- Simplicity and pragmatism

Node JS Patterns:

- Module Pattern
- Reactor Pattern
  - Call Back Pattern
- Observer Pattern
  - Event Emitter

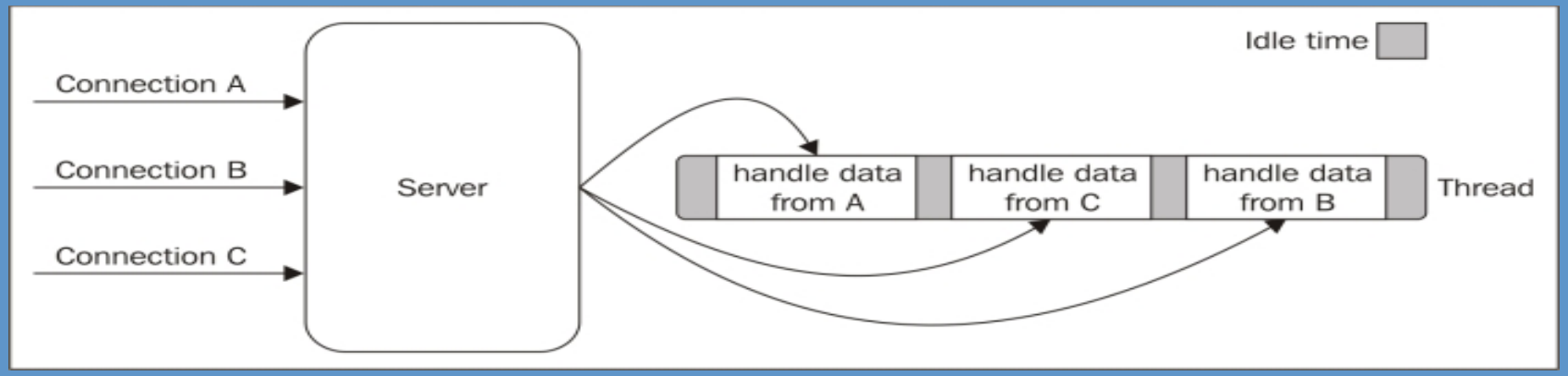
Problem Statement:

- **IO is Slow.**
  - Accessing the RAM is in the order of nanoseconds (10e-9 seconds), while accessing data on the disk or the network is in the order of milliseconds (10e-3 seconds).
- **Blocking IO**
  - The function call corresponding to an I/O request will block the execution of the thread until the operation completes.
  - Results in the multiple threaded approach. Each request, separate thread.
  - Threads consumes memory and causes context switches, so having a thread blocked…. Not a good idea.
- **Non Blocking IO**
  - Most OS has it already !!!! *epoll on Linux, kqueue on Mac OS X, and I/O Completion Port API (IOCP) on Windows.*
  - the system call always returns immediately without waiting for the data to be read or written.

Achieving Non-Blocking IO:

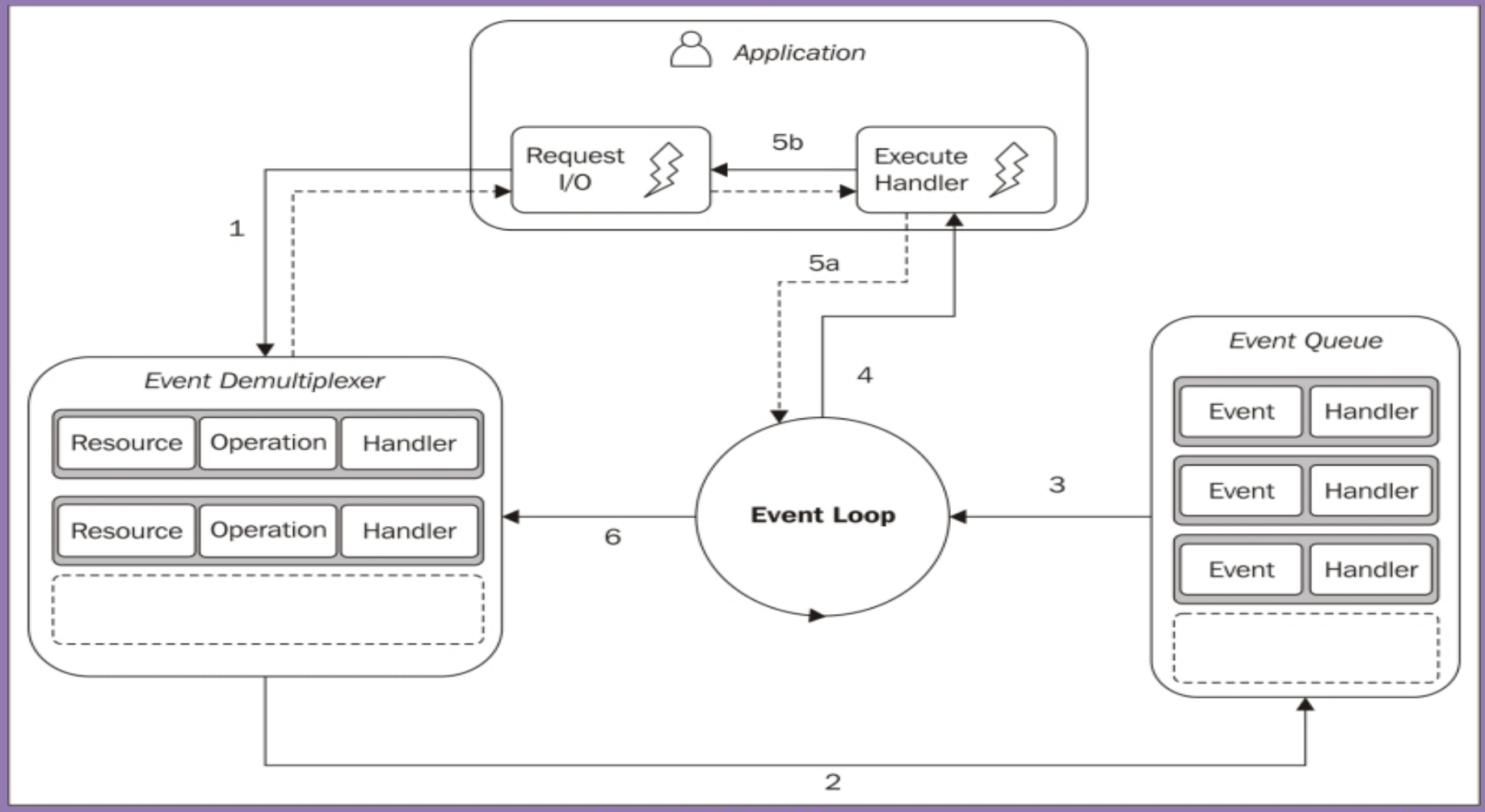
- busy-waiting.
  - Actively poll the resource within a loop until some actual data is returned.
  - The loop will consume precious CPU only for iterating over re sources that are unavailable most of the time hence not efficient.

- Event De multiplexing
  - Preferred way of handling Non-Blocking IO in most modern OS.
  - Collects and queues I/O events that come from a set of watched resources, and block until new events are available to process.



• Reactor Pattern:

– **Handles I/O by blocking until new events are available from a set of observed resources, and then reacting by dispatching each event to an associated handler.**



libuv for Reactor Pattern:

1. Serves to abstract all the inconsistencies across the different Operating Systems implementing the Event Demultiplexer.
2. Makes Node.js compatible with all the OS.
3. Implements the reactor pattern.

http://nikhilm.github.io/uvbook/

CallBack Pattern [Continous Passing Style]:

- Function passed to another function.
- The functions which are used to **propagate the results** of an operation.
- Replace the "**return**" instruction.
- **Closures** are ideal for the callback pattern. *
- Lets analyze further….

Synchronous CPS:

```
function add(a, b) { // direct style
  return a + b; *
}
function add(a, b, callback) { // CPS
  callback(a + b);
}
```

Invocation changes:

```
console.log('before');
add(1, 2, function(result) {
  console.log('Result: ' + result);
});
console.log('after')
```

Result:

```
before
Result: 3
after
```

Asynchronous CPS:

```
function addAsync(a, b, callback) {
  setTimeout(function() { // simulates the asynchronous invocation of the callBack
    callback(a + b);
  }, 100);
```

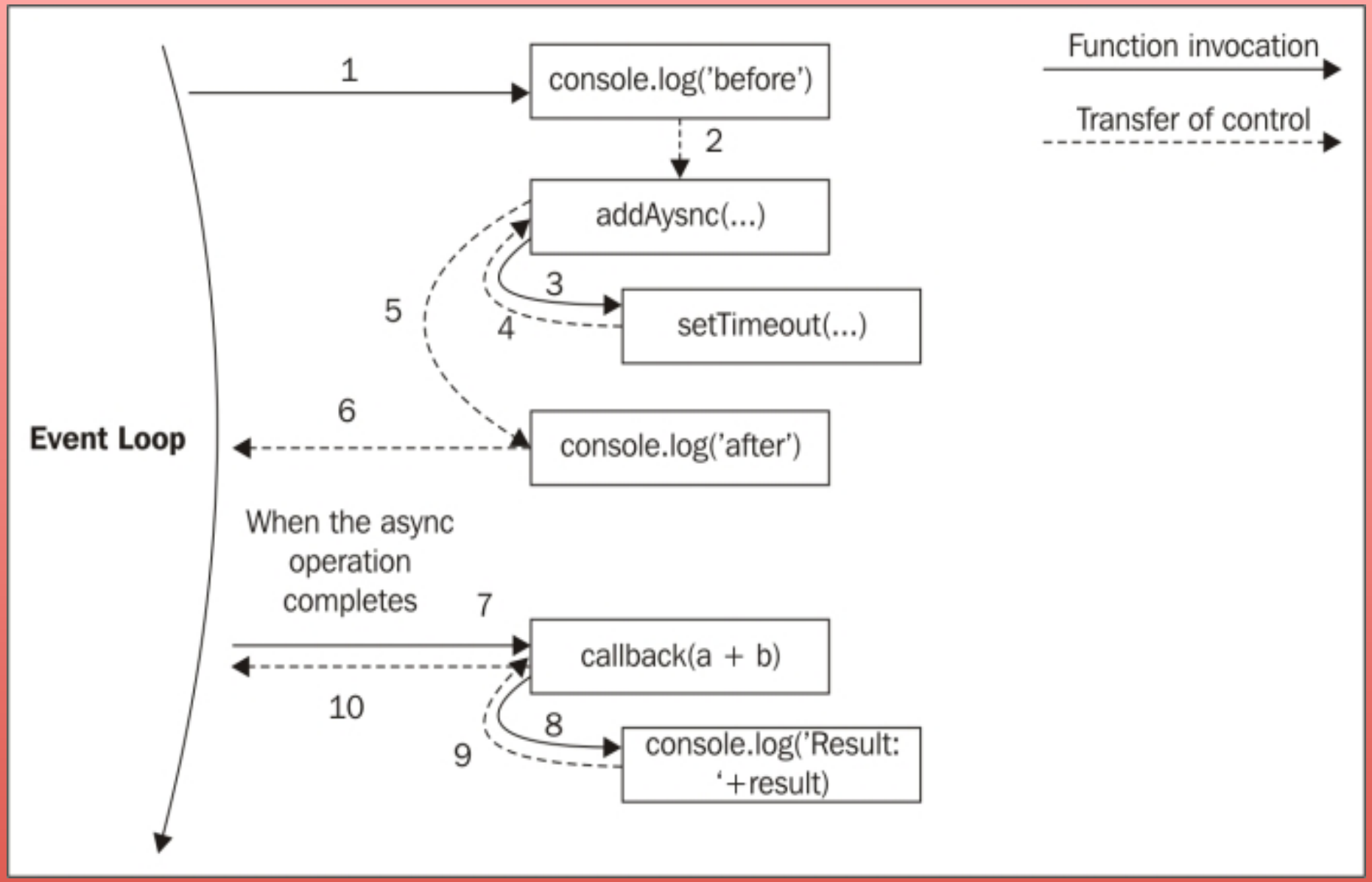
Invocation changes:

```
console.log('before');
addAsyc(1, 2, function(result) {
  console.log('Result: ' + result);
});
console.log('after')
```

Result:

before
After
Result :3

Asynchronous CPS:



Non CPS CallBacks !!!!:

```
var result = [1, 5, 7].map(function(element) {
  return element – 1;
});
```

the presence of a callback doesn't necessarily means the function asynchronous or is using a continuation-passing style.

So what to use when designing an API:

My two cents:

- Don’t confuse.* [http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony]
- Prefer the direct style for purely synchronous functions.
- Use blocking API only when they don't affect the ability of the application to serve concurrent requests.

Call Back Conventions:

1. Specific Conventions.
2. Followed by not just the core team but also usually followed by the user defined modules
   - Call Backs comes last.
   - Error comes first.
   - Error propagation done by passing the error to the callback.
   - Never throw the exception in asynchronous callbacks.

Call Back Comes last:

fs.readFile(filename, [options], callback)

Error comes first :

- Errors in CPS are propagated via callbacks.
- any error produced by a CPS function is always passed as the first argument of the callback, and any actual result is passed starting from the second argument.
- If the operation succeeds without errors, the first argument will be null or undefined.

```
fs.readFile('foo.txt', 'utf8', function(err, data) {
 if(err)// Would be null or undefined in case of operation successful
   handleError(err);
 else
   processData(data);
})
```

Propagating Errors:

Errors should be propagated via callbacks.

```
var fs = require('fs');
function readJSON(filename, callback) {
 fs.readFile(filename, 'utf8', function(err, data) {
  var parsed;
  if(err)
   //propagate the error and exit the current function
   return callback(err);
  try {
   //parse the file contents
   parsed = JSON.parse(data);
  } catch(err) {
   //catch parsing errors
   return callback(err);
  }
```

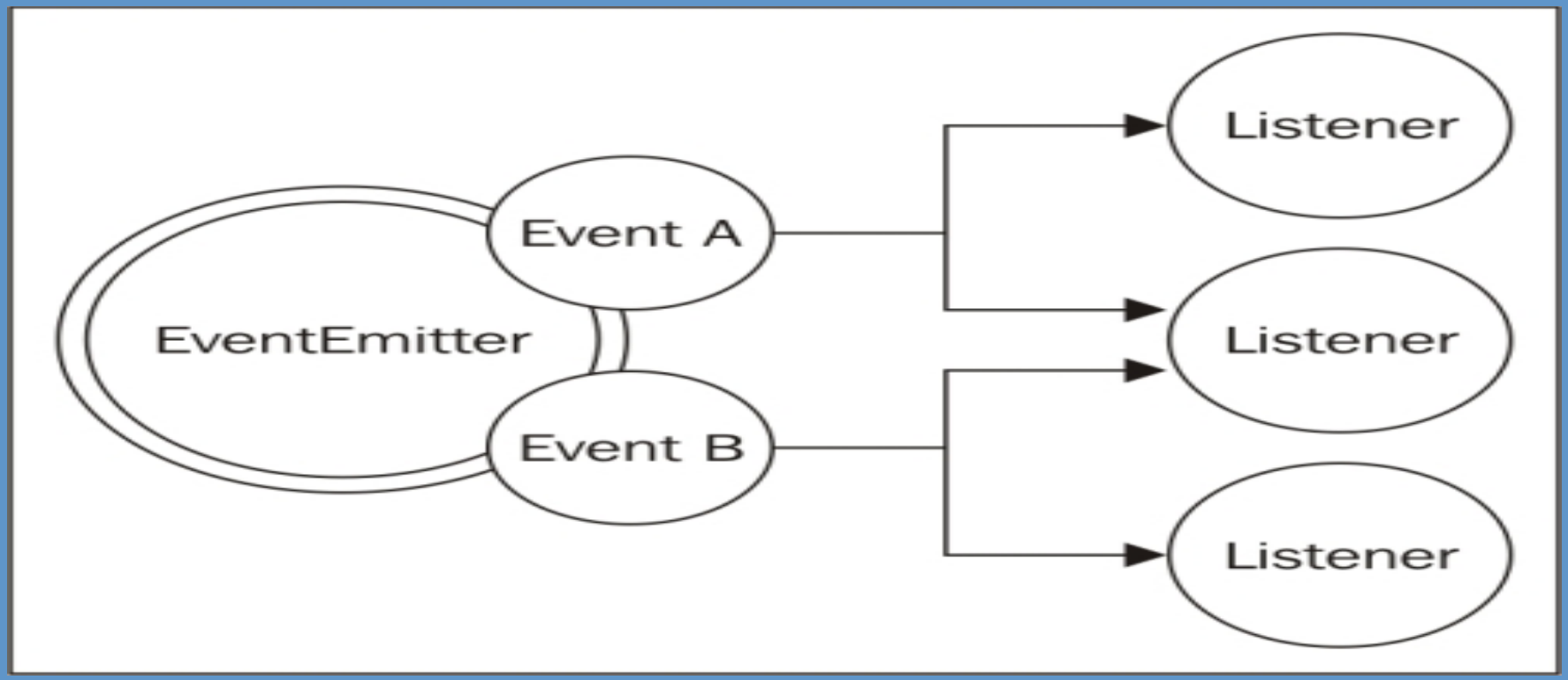
Uncaught Exceptions:

- Should never throw the exceptions in the asynchronous callback.
- Throwing will cause the exception to jump up to the event loop and never be propagated to the next callback.
- Application cannot recover from such a state.
- The stack in which the block operates is different from the one in which our callback is invoked

NodeJS/Tutorial_1/Node_JS_AsynchronousWay/errorHandling/errorHandling_1.js

The Observer Pattern:

- Defines an object (called subject), which can notify a set of observers (or listeners), when a change in its state happens.
- Can notify multiple observers unlike the callback pattern.*
- At the heart resides the Event Emitter. **



Event Emitter Basics :

- Getting an event emitter reference:
  ```
  var EventEmitter = require('events').EventEmitter;
  var eeInstance = new EventEmitter();
  ```

- Essential Methods [Each returns the eventEmitter instance hence can be chained]:
  - **on(event, listener):** Allows you to register a new listener (a function) for the given event type (a string)
  - **once(event, listener):** Allows you to register a new listener, which is then removed after the event is emitted for the first time
  - **emit(event, [arg1], [...]):** Produces a new event and provides additional arguments to be passed to the listeners
  - **removeListener(event, listener):** Removes a listener for the specified event type.

- The listener function signature : function([arg1], [....])
  NodeJS/Tutorial_3/EventEmitterPattern/main.js

What to use when ????

Two Cents:

- Use CPS[Asynchronous Callbacks] when result need to be returned in an asynchronous way.
- Use Event Emitter when there is an immediate need to communicate one or more listeners that some thing in the workflow has just happened.
- But still there is a lot of confusion since both the solutions can pretty much serve the same purpose.

```
function helloEvents() {
  var eventEmitter = new EventEmitter();
  setTimeout(function() {
    eventEmitter.emit('hello', 'world');
  }, 100);
  return eventEmitter;
}
function helloCallback(callback) {
  setTimeout(function() {
    callback('hello', 'world');
  }, 100);
}"
```

Different Mindsets:

Synchronous coding come natural

- Natural Order

Asynchronous : Not so Easy !!!

- Serial Execution requires special handling.
- New paradigm : Parallel Execution.
- Single Callback functions and Event Emitters.
- Explicit Error Handling required…

Synchronous Code:

```
var filenames = fs.readdirsync('/temp/')
for(var i= 0 ; i < filenames.length; i++){
      console.log(filenames[i])
}
console.log('Done!!!!!')
```

Asynchronous Code:

```
fs.readdir('/temp/', function(err,filenames){
        for(var i= 0 ; i < filenames.length; i++){
                console.log(filenames[i])
        }
        console.log('Done!!!!!');
})
```

Asynch Control Flows:

Transition from the synchronous to asynchronous paradigm can be frustrating.

1. Things which were very simply achieved in the synchronous world without an issue, can be tricky in Async world such as:
   ① Iterating over a collection and perform an asynch operations.
   ② Executing the tasks in a sequence.
   ③ Waiting for a set of operations to complete.

2. In this session we would try to touch on these problems and try to solve them using the various techniques such as:
   1. Pure JS.
   2. Async NPM
   3. Promise.

Call Back Hell????

- Several Level of indentation.
- Abundance of Closures and in-place callback definitions.
- Code is rendered as an unreadable and unmanageable blob.

```
asyncFoo(function(err) {
  asyncBar(function(err) {
    asyncFooBar(function(err) {
      [...]
    });
  });
})
```

- Severe Anti Pattern.
- Shaped in a formation of doom also called the Pyramid of Doom resulting is:
  - Poor Readability.[Cannot make out where the function starts and where it ends.]
  - Variable names can be overstepping. *
  - Closures come at a small price in terms of performances and memory consumption. **

Call Back Hell????

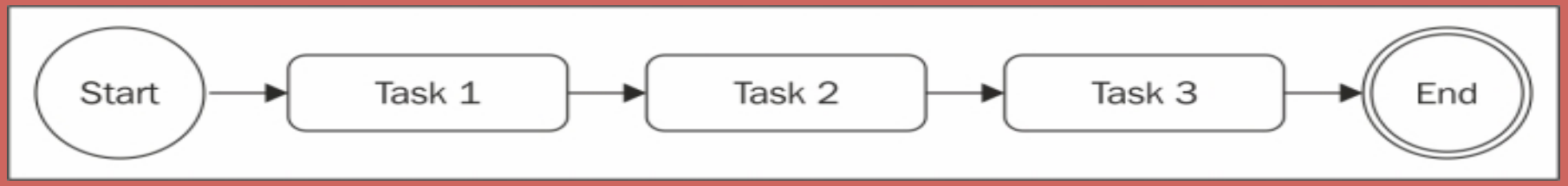
```
doAsync1(function () {
  doAsync2(function () {
    doAsync3(function () {
      doAsync4(function () {
    })
  })
})
```

```
var fs = require('fs')
var path = require('path')

module.exports = function (dir, cb) {
  fs.readdir(dir, function (er, files) { // [1]
    if (er) return cb(er)
    var counter = files.length
    var errored = false
    var stats = []

    files.forEach(function (file, index) {
      fs.stat(path.join(dir,file), function (er, stat) { // [2]
        if (errored) return
        if (er) {
          errored = true
          return cb(er)
        }
        stats[index] = stat // [3]

        if (--counter == 0) { // [4]
          var largest = stats
            .filter(function (stat) { return stat.isFile() }) // [5]
            .reduce(function (prev, next) { // [6]
              if (prev.size > next.size) return prev
              return next
            })
          cb(null, files[stats.indexOf(largest)]) // [7]
        }
      })
    })
  })
}
```

Pure JS

- Without the aid of any external JS library, controlling the flow of a set of asynchronous tasks, require specific patterns and techniques.
- Avoiding CallBack Hell by CallBack Discipline.
  - Exit as soon as possible. Use
    - return,
    - continue
    - break, depending on the context, to immediately exit the current statement instead of writing (and nesting) complete if/else statements. *
  - Create named functions for callbacks, keeping them out of closures and passing intermediate results as arguments. **
  - Modularize the code. ***

Sequential Flow:



There are different variations of this flow:

1. Executing a set of known tasks in sequence, without chaining or propagating results
2. Using the output of a task as the input for the next (also known as chain, pipeline, or waterfall)
3. Iterating over a collection while running an asynchronous task on each element, one after the other

Plain JS: Sequential Execution

```
function task1(callback) {
  asyncOperation(function() {
    task2(callback);
  });
}
function task2(callback) {
  asyncOperation(function(result) {
    task3(callback);
  });
}
function task3(callback) {
  asyncOperation(function() {
    callback();
  });
}
task1(function() {
  //task1, task2, task3 completed
});"
```

The example at :
NodeJS/Tutorial_3/Server_Basic/plane_js/plane_js_sequential_execution/readFilesSequential.js
Is indeed an example of sequential execution wherein each task invokes the next upon the completion of a generic asynchronous operation.

We can generalize it to a the adjoining pattern.

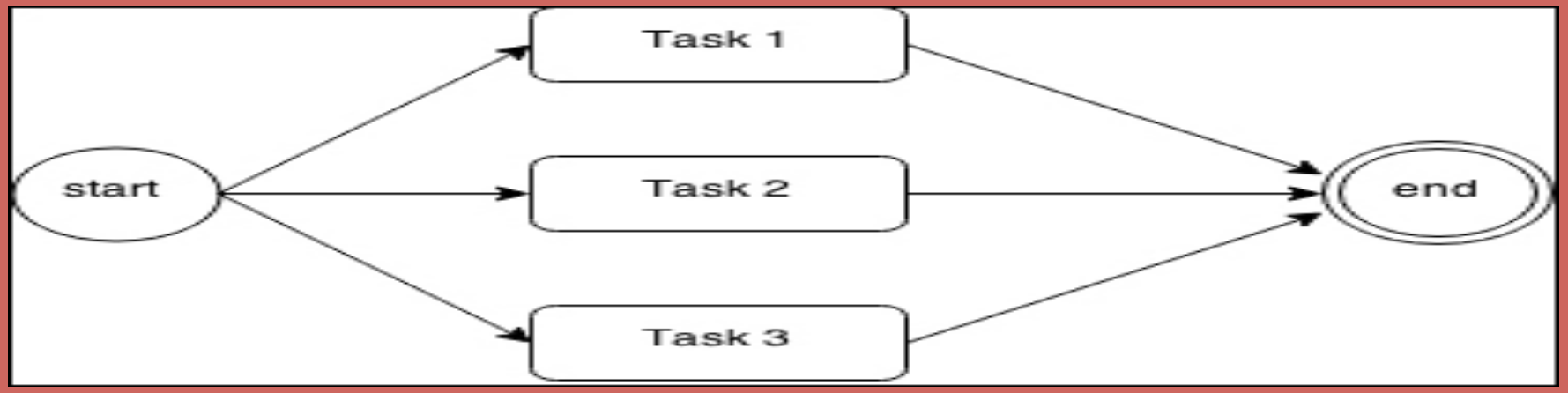
The pattern puts the emphasis on the modularization of tasks, showing how closures are not always necessary to handle asynchronous code.

Plain JS: Sequential Iteration

```
function iterate(index) {
  if(index === tasks.length)  {
    return finish();
  }
  var task = tasks[index];
  task(function() {
    iterate(index + 1);
  });
}

function finish() {
  //iteration completed
}
iterate(0);
```

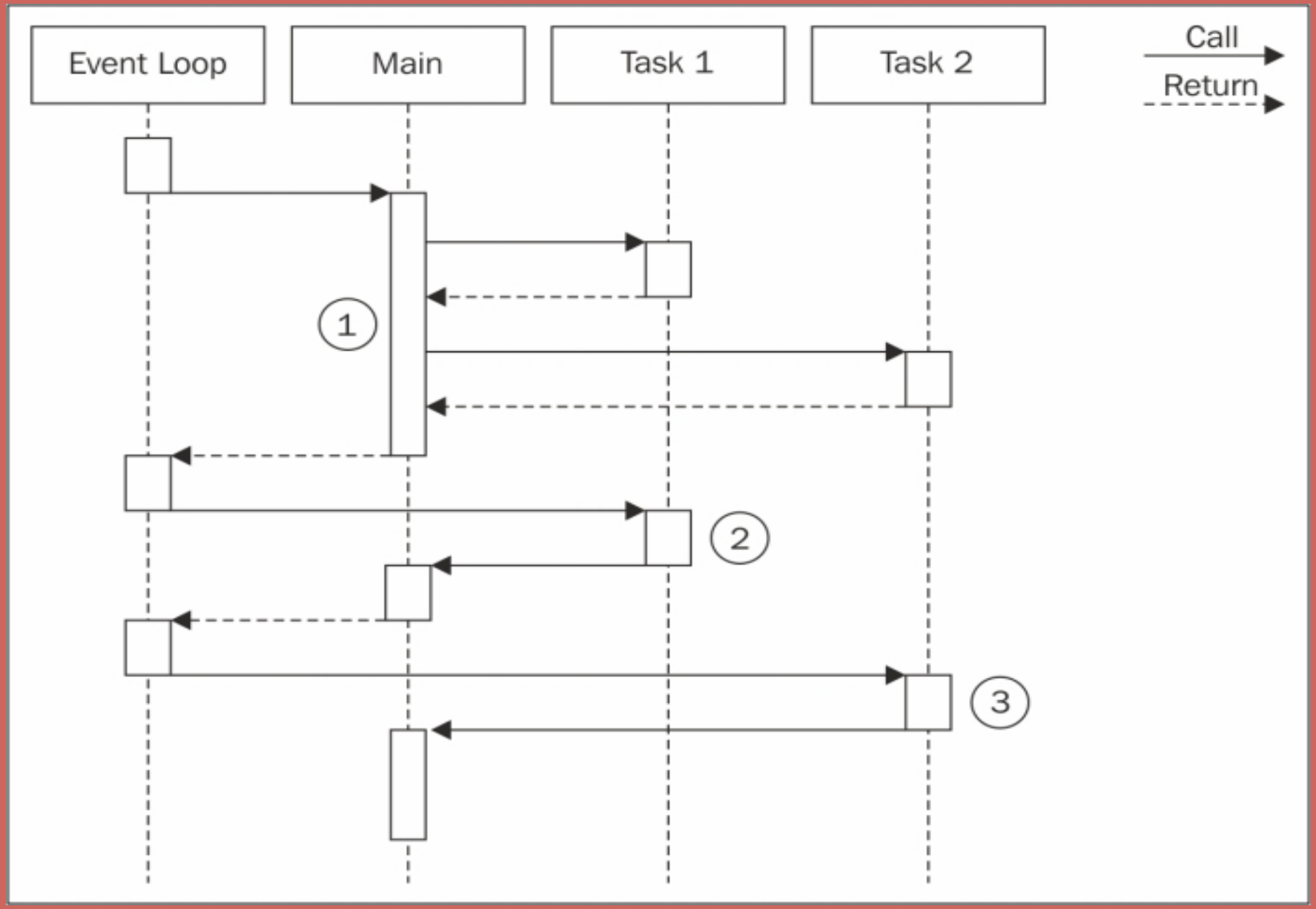
- Applies to scenarios wherein we don't know in advance how many asynchronous operations are to be performed.
- Common example is to perform an asynchronous task on a collection of items.
- To demonstrate we can modify the example that we have been building on to write the output to the result.txt only if it happens to be a directory.
- Let us first look at the problem while we try to perform sequential iteration:
  - ***NodeJS/Tutorial_3/Server_Basic/plain_js/plain_js_sequential_Iteration/readFilesSequential_problematic.js***
- Now let us look at the modified example:
  - ***NodeJS/Tutorial_3/Server_Basic/plain_js/plain_js_sequential_Iteration/readFilesSequential.js***

Parallel Execution:



- The order is not important.
- Needs a notification only at the end of all the tasks.
- But we said Node.js was single threaded, then where the heck did parallellism come!!!!!
- Next Slide please

Parallel Execution:



Pattern:

```
var tasks = [...];
var completed = 0;
tasks.forEach(function(task) {
task(function() {
    if(++completed === tasks.length) {
      finish();
    }
  });
});

function finish() {
  //all the tasks completed
}
```

1. Applies to scenarios wherein we don't care about the order of completion of the asynchronous tasks.
2. All we need is an acknowledgement that all the tasks are finished.
3. Common example is to read file concurrently and notifying when each of the files have been read.
4. To demonstrate we can modify the example that we have been building on to write the output to the result.txt but don't do it sequentially.

NodeJS/Tutorial_3/Server_Basic/plain_js/plainjs_parallel_execution/Node_Server_Parallel_Execution.js

Summary

- Define your challenges
  - Technological as well as personal
- Set realistic expectation
  - Mastery is not achieved overnight
- Keep your eye on the goal
  - Mentorship programs

Resources

- <Intranet site text here>
  <hyperlink here>

- <Additional reading material text here>
  <hyperlink here>

- This slide deck and related resources:
  <hyperlink here>

QUESTIONS?

APPENDIX