Synkron I/O

```
try(FileInputStream inputStream = new
FileInputStream("foo.txt")) {
    Session IOUtils;
    String fileContent = IOUtils.toString(inputStream);
}
```

- What happens in the background? The main thread will be blocked until the file is read, which means that nothing else can be done in the meantime. To solve this problem and utilize your CPU better, you would have to manage threads manually.

- If you have more blocking operations, the event queue gets even worse

*(The **red bars** show when the process is waiting for an external resource's response and is blocked, the **black bars** show when your code is running, the **green bars** show the rest of the application)*

Asynchronous programming in Node.js

- To resolve this issue, Node.js introduced an asynchronous programming model.

- Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission has finished.

```
const fs = require('fs')
let content
try {
  content = fs.readFileSync('file.md', 'utf-8')
} catch (ex) {
  console.log(ex)
}
console.log(content)
```

Asynkron programmering

- Functions that can take other functions as arguments are called higher-order functions.

- In this example we pass in a function to the filter function. This way we can define the filtering logic.

- This is how callbacks were born: if you pass a function to another function as a parameter, you can call it within the function when you are finished with your job. No need to return values, only calling another function with the values.

```
const numbers = [2,4,1,5,4]

function isBiggerThanTwo (num) {
  return num > 2
}

numbers.filter(isBiggerThanTwo)
```

Asynkron programmering

```
const fs = require('fs')
fs.readFile('file.md', 'utf-8', function (err, content) {
  if (err) {
    return console.log(err)
  }

  console.log(content)
})
```

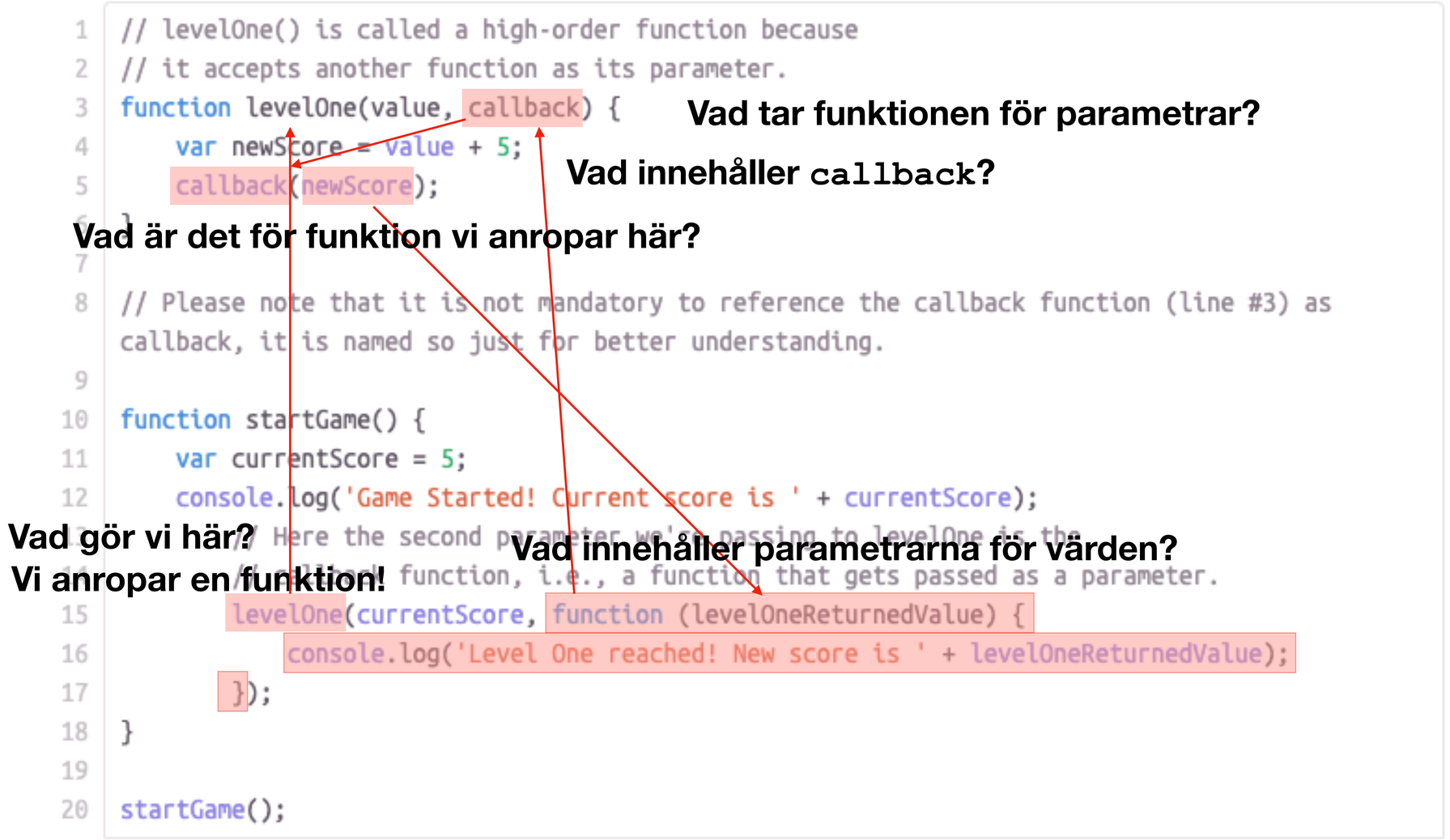
- Things to notice here:
  - error-handling: instead of a try-catch block you have to check for errors in the callback
  - no return value: async functions don't return values, but values will be passed to the callbacks

Callback functions

- ”A callback function in its simplest terms is a function that is passed to another function, as a parameter. The callback function then gets executed inside the function where it is passed and the final result is returned to the caller.”

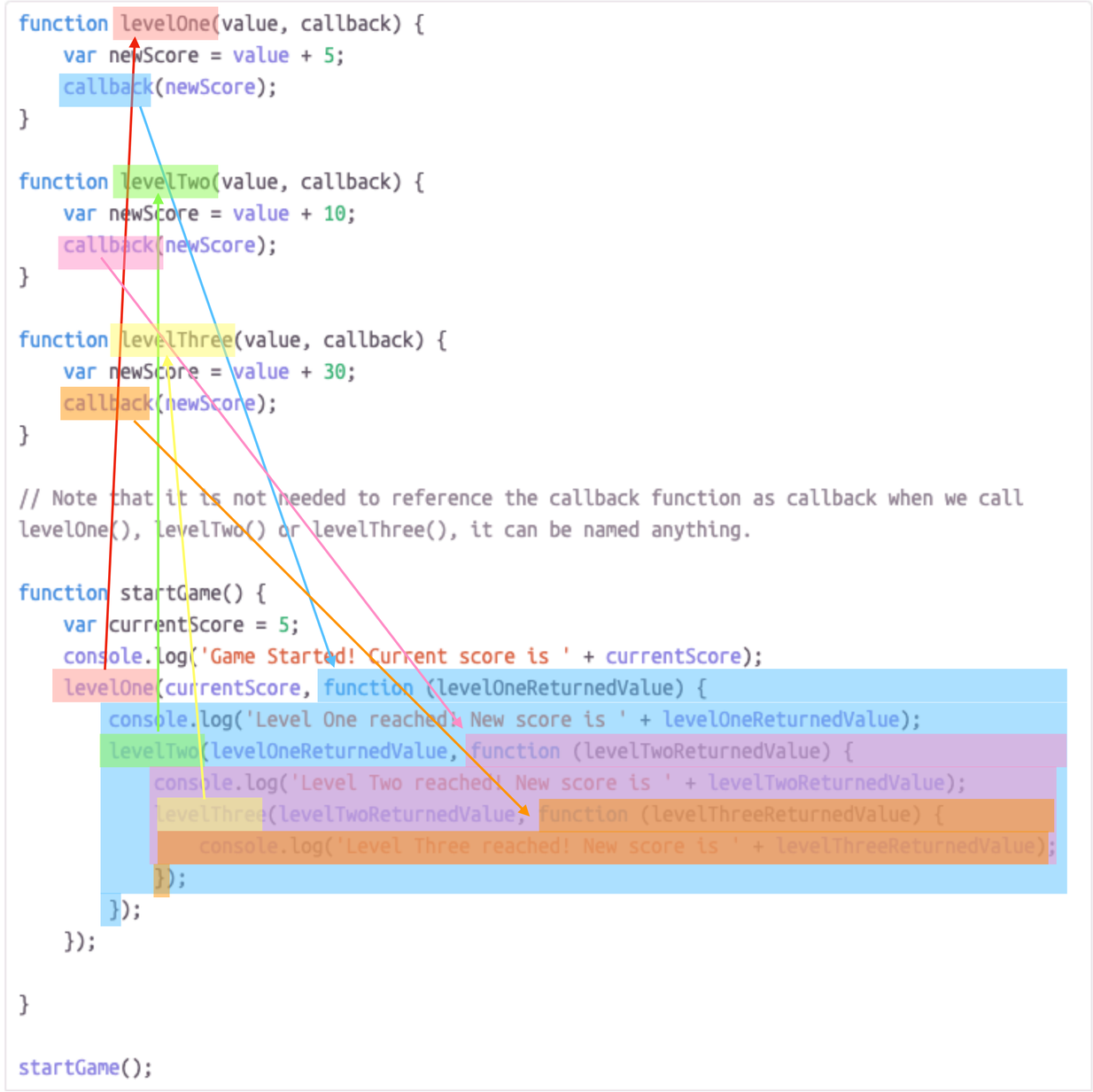
```
// I'm sure you've seen a JQuery code snippet like this at some point in your life!
// The parameter we're passing to the `click` method here is a callback function.

$("button").click(function() {
    alert('clicked on button`);
});
```

Nästa exempel



Callbacks

- Callbacks kan vara ganska komplexa.
- Tänk om vi skulle behöva lägga till 10 nivåer till.
- Kallas ibland callback hell. Vad kan vi göra istället?



```
function levelOne(value, callback) {
    var newScore = value + 5;
    callback(newScore);
}

function levelTwo(value, callback) {
    var newScore = value + 10;
    callback(newScore);
}

function levelThree(value, callback) {
    var newScore = value + 30;
    callback(newScore);
}

// Note that it is not needed to reference the callback function as callback when we call
levelOne(), levelTwo() or levelThree(), it can be named anything.

function startGame() {
    var currentScore = 5;
    console.log('Game Started! Current score is ' + currentScore);
    levelOne(currentScore, function (levelOneReturnedValue) {
        console.log('Level One reached! New score is ' + levelOneReturnedValue);
        levelTwo(levelOneReturnedValue, function (levelTwoReturnedValue) {
            console.log('Level Two reached! New score is ' + levelTwoReturnedValue);
            levelThree(levelTwoReturnedValue, function (levelThreeReturnedValue) {
                console.log('Level Three reached! New score is ' + levelThreeReturnedValue);
            });
        });
    });

}

startGame();
```

Promises

- Javascript started supporting Promises from ES6. Promises are basically objects representing the eventual completion (or failure) of an asynchronous operation, and its resulting value.

```
// This is how a sample promise declaration looks like. The promise constructor
// takes one argument which is a callback with two parameters, `resolve` and
// `reject`. Do something within the callback, then call resolve if everything
// worked, otherwise call reject.

var promise = new Promise(function(resolve, reject) {
  // do a thing or twenty
  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

Promises

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

```
function successCallback(result) {
  console.log("Audio file ready at URL: " + result);
}

function failureCallback(error) {
  console.error("Error generating audio file: " + error);
}

createAudioFileAsync(audioSettings, successCallback, failureCallback);
```

```
function levelOne(value) {
    var promise, newScore = value + 5;
    return promise = new Promise(function(resolve) {
        resolve(newScore);
    });
}

function levelTwo(value) {
    var promise, newScore = value + 10;
    return promise = new Promise(function(resolve) {
        resolve(newScore);
    });
}

function levelThree(value) {
    var promise, newScore = value + 30;
    return promise = new Promise(function(resolve) {
        resolve(newScore);
    });
}

var startGame = new Promise(function (resolve, reject) {
    var currentScore = 5;
    console.log('Game Started! Current score is ' + currentScore);
    resolve(currentScore);
});

// The response from startGame is automatically passed on to the function inside the
subsequent then
startGame.then(levelOne)
.then(function (result) {
    // the value of result is the returned promise from levelOne function
    console.log('You have reached Level One! New score is ' + result);
    return result;
})
.then(levelTwo).then(function (result) {
    console.log('You have reached Level Two! New score is ' + result);
    return result;
})
.then(levelThree).then(function (result) {
    console.log('You have reached Level Three! New score is ' + result);
});
```

Promises

- Läs mer:
  - https://medium.com/quick-code/javascript-promises-in-twenty-minutes-3aac5b65b887

Await

- ”Async- await is being supported in javascript since ECMA2017. They allow you to write promise-based code as if it were synchronous code, but without blocking the main thread. They make your asynchronous code less "clever" and more readable.”

- ”If you use the `async` keyword before a function definition, you can then use `await` within the function. When you await a promise, the function is paused in a non-blocking way until the promise settles. If the promise fulfils, you get the value back. If the promise rejects, the rejected value is thrown.”

```
function levelOne(value) {
    var promise, newScore = value + 5;
    return promise = new Promise(function(resolve) {
        resolve(newScore);
    });
}

function levelTwo(value) {
    var promise, newScore = value + 10;
    return promise = new Promise(function(resolve) {
        resolve(newScore);
    });
}

function levelThree(value) {
    var promise, newScore = value + 30;
    return promise = new Promise(function(resolve) {
        resolve(newScore);
    });
}

// the async keyword tells the javascript engine that any function inside this function
having the keyword await, should be treated as asynchronous code and should continue
executing only once that function resolves or fails.
async function startGame() {
    var currentScore = 5;
    console.log('Game Started! Current score is ' + currentScore);
    currentScore = await levelOne(currentScore);
    console.log('You have reached Level One! New score is ' + currentScore);
    currentScore = await levelTwo(currentScore);
    console.log('You have reached Level Two! New score is ' + currentScore);
    currentScore = await levelThree(currentScore);
    console.log('You have reached Level Three! New score is ' + currentScore);
}

startGame();
```

Vad blir output?

```
const fs = require('fs')

console.log('start reading a file...')

fs.readFile('file.md', 'utf-8', function (err, content) {
  if (err) {
    console.log('error happened during reading the file')
    return console.log(err)
  }

  console.log(content)
})

console.log('end of the file')
```

```
start reading a file...
end of the file
error happened during reading the file
```

- As you can see once we started to read our file the execution continued, and the application printed end of the file. Our callback was only called once the file read was finished. How is it possible? Meet the event loop

The Event Loop

- The event loop is in the heart of Node.js / Javascript - it is responsible for scheduling asynchronous operations.

- Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.

- https://youtu.be/8aGhZQkoFbQ

Övningar

- https://github.com/stevekane/promise-it-wont-hurt
- https://github.com/bulkan/async-you
- https://www.youtube.com/watch?v=PoRJizFvM7s