

Asynchronous program flow

Synchronous program flow

- Traditionally programs execute the code from the start to the end – statement by statement (line by line)
- The program will wait for each statement to finish, before running the next statement
- For example, if a statement is going to load a file, the file must be loaded before the next statement is run
- Loading a file can take a long time. The program is therefore “blocked” – and can’t do other stuff - while the file is loaded, e.g.:

...some code...

...some code...

```
file = loadFile("myFile");
```

...do something with the file...

...some code...

...some code...

This code is blocked until the file is loaded

- To make program flow more effective, programs can run *asynchronous* (*async*)

Asynchronous program flow

- Instead of letting the load file statement block the program flow, we can continue with other stuff while the file is loading – and then come back and run some code when the loading is finished
- A common way to do this is to use *callback functions (callbacks)*. A callback is a function that is called when something has happened (e.g., the file has been loaded):

...some code...
...some code...

Callback

loadFile("myFile", finished);

```
function finished(file) {  
    ...do something with the file...  
}
```

...some code...
...some code...

This code can now run
while the file is loading

Implementing async prog. flow

- In JavaScript there are different ways to implement asynchronous program flow:
 1. Using the event system
 2. Using simple callbacks
 3. Using promises
 - I. Using *then(success, fail)*
 - II. Using *async* and *await*

The event system

- The event system in JavaScript is by definition asynchronous. E.g., if you register an event handler for mouseclick, the program will not wait until you click the mouse. Instead, it can do other stuff in the meantime, e.g., draw something in the canvas, do something if you press a key etc.

```
myButton.addEventListener("click", startMovie);
```

```
function startMovie(evt) {  
    //code...  
}
```

- You can also create your own events and send (dispatch) them so that event handlers for your (self-made) event is called.

Dispatching (sending) events

- To send an event, we must first create an event object and then dispatch the event from an object (e.g. a list element). If we want, we can also put in some extra data inside the object, e.g.:

```
var myEvent = new Event("armageddon");  
myEvent.someData = "don't panic!";  
myEvent.someMoreData = "Trump will save you!";  
myList.dispatchEvent(myEvent);
```

← Name of the event

Note! We can only dispatch events from elements on the webpage (DOM-objects).

- We can then register an event handler for the event (listen to events from the button):

```
myList.addEventListener("armageddon", endOfWorld);
```

```
function endOfWorld(evt) {  
    console.log(evt.someData); //don't panic  
    console.log(evt.someMoreData); // Trump will save you!;  
}
```

Callbacks

- The second way to implement async program flow is to use callbacks. An example of a pre-made function that uses a callback is *setTimeout*:

```
setTimeout(timeout, 5000);
```

```
function timeout(){  
    //code...  
}
```

Here *timeout* is the callback – which is called after five secons.

- If the callback is referred to from only one place in the code, it is common to include it in the call as an anonymous function, e.g.:

```
setTimeout(function(){  
    //code...  
}, 5000);
```

Promises

- The event system is good on handling things that can happen multiple times on the same object, e.g., mouse clicks, keypress etc.
- Callbacks is often used on things that happens once (or once in a while), and that can result in either success or failure, e.g., loading a file
- In ES6 (and several other programming languages) we have a more robust callback system that uses something called *promises*
- A promise is a type of object that will receive a certain value in the future – either *resolved* or *rejected* based on some process or operation – e.g., downloading data from a server. Until the promise is resolved or rejected, it has the value of *pending*.
- By using promises it becomes easier to handle several asynchronous tasks at once – e.g., do something when a set of multiple files have finished loading – or controlling the order of async tasks. By using promises it is also easier to handle errors that can happen during async tasks.

Promise states

- A promise can be in one of three states:
 - *Pending* – the outcome isn't yet determined (the async task has not completed yet) `▶ Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}`
 - *Resolved* – the async task has completed successfully with a result value `▶ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: "Hello!"}`
 - *Rejected* – the async task has failed and the promise will not be fulfilled. The promise will contain a result value for the failure `▶ Promise {[[PromiseStatus]]: "rejected", [[PromiseValue]]: "Something went wrong!"}`
- The *pending* state can transition to either *resolved* or *rejected*. The resolved and rejected states can't transition to other states – and their value can't change

Using promises

- Let's say we are calling a function named *sayHello* that returns a message string:

```
var msg = sayHello();  
console.log(msg); //Hello!
```

- If *sayHello* is doing an async task, e.g., loading the message from a server on the internet, it may return a promise – instead of the message string:

```
var msgPromise = sayHello();
```

Using promises cont.

- We must then handle the promise by defining two callbacks that are called if the promise is successful or fails (the fail-function is optional):

```
msgPromise.then(success, fail);
```

```
function success(result) {  
    console.log(result); //Hello!  
}
```

```
function fail(err) {  
    console.log(err); //Something went wrong!  
}
```

Using promises cont.

- The code in the previous slide is often shortened by using anonymous functions:

```
myPromise.then(function(result){...code}, function(err){...code});
```

For example:

```
msgPromise.then(function(result) {  
    console.log(result); //Hello!  
},  
function(err) {  
    console.log(err); //Something went wrong!  
});
```

Handling a set of async tasks in sequence

- Promises are great for handling async tasks in a specific order. For example, first you want to download something. When the download is finished, you want to download something else, and so on... (Note! The methods mentioned here can also be implemented (using less code) with *async* and *await* – see slides later...)
- Let's say you have three functions, named *sayHello*, *sayCurse* and *sayGrunt*, that are returning promises. We create a sequence by letting the callbacks return promises from the other functions and chaining them together (Note. In the example below we don't handle errors):

```
var msgPromise = sayHello();  
msgPromise.then(success1).then(success2).then(success3);
```

```
function success1(result) {  
    console.log(result); //Hello!  
    return sayCurse();  
}
```

```
function success2(result) {  
    console.log(result); //Darn!  
    return sayGrunt();  
}
```

```
function success3(result) {  
    console.log(result); //Arghh!  
}
```

Handling a set of async tasks in sequence cont.

- Here is the previous example using anonymous functions:

```
var msgPromise = sayHello();  
msgPromise.then(function (result) {  
    console.log(result); //Hello!  
    return sayCurse();  
}).then(function (result) {  
    console.log(result); //Darn!  
    return sayGrunt();  
}).then(function (result) {  
    console.log(result); //Arghh!  
});
```

Handling errors with *catch*

- You can add a *catch* at the end of the chain. The callback in *catch* will be called for any unhandled error in the chain.

```
var msgPromise = sayHello();  
msgPromise.then(success1).then(success2).then(success3).catch(fail);
```

```
function success1(result) {  
    console.log(result); //Hello!  
    return sayCurse();  
}
```

```
function success2(result) {  
    console.log(result); //Darn!  
    return sayGrunt();  
}
```

```
function success3(result) {  
    console.log(result); //Arghh!  
}
```

```
function fail(err) {  
    console.log(err); //Something bad happened  
}
```

Handling separate errors in the chain

- If you want to handle errors for each link in the chain, you can include fail-callbacks inside *then* – and let the callbacks return the same promise as the corresponding success callbacks:

```
var msgPromise = sayHello();
msgPromise.then(success1, error1)
             .then(success2, error2)
             .then(success3, error3);

function success1(result) {
  console.log(result); //Hello!
  return sayCurse();
}
function error1(err) {
  console.log(err); //Can't do Hello!
  return sayCurse();
}

function success2(result) {
  console.log(result); //Darn!
  return sayGrunt();
}
function error2(err) {
  console.log(err); //Can't do darn!
  return sayGrunt();
}

function success3(result) {
  console.log(result); //Arghh!
}
function error3(err) {
  console.log(err); //Can't do Arghh!
}
```


Handling separate errors with *catch*

- You can use *catch* in the chain almost in the same way as using fail-callbacks inside *then*:

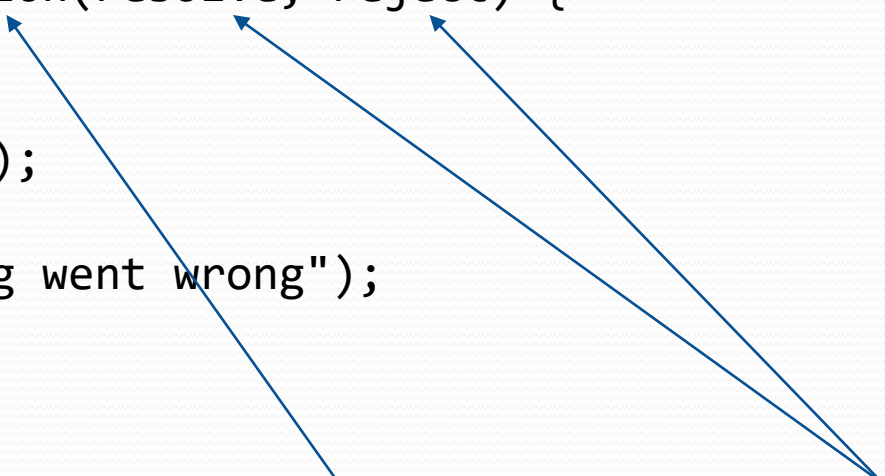
```
var msgPromise = sayHello();  
msgPromise.then(success1).catch(error1)  
            .then(success2).catch(error2)  
            .then(success3).catch(error3);
```

- The implementation of the callbacks is the same as in previous slide
- Also note that you can throw an error, e.g.: **throw new Error("bad stuff!")** from any callback. The subsequent error callback will then be called
- There is a subtle difference between the two methods when you throw an error from inside a success callback. If you use *catch*, the fail-callback in the same promise will be called. If you use fail-callbacks inside *then*, the error callback in the next promise in the chain will be called. E.g. if you throw an error from *success1* above, the following sequence will occur: *success1* -> *error1* -> *success2* -> *success3*. If you don't use *catch*, but handles error from inside *then*, the following sequence will occur: *success1* -> *error2* -> *success3*

Creating promises

- The previous slides dealt with how to use functions that returned a promise (instead of a concrete value)
- We can also create promises (promise objects) – much the way we can create events (event objects) in an event system – and make a functions return them:

```
function sayHello() {  
    return new Promise(function(resolve, reject) {  
        //if things went ok  
        resolve("Hello!");  
        //else  
        reject("Something went wrong");  
    });  
}
```



Note that we crate a promise object by passing a function which again has the two callbacks as parameters. It is common to name the parameters *resolve* and *reject* for the corresponding success and failure callbacks.

Async and await

- An alternative way to handle promises is to use *async* functions.
- Async functions are functions that can be halted (i.e. waiting for something to finish), while other code in the program can still run.
- We use the codeword *async* to make a function asynchronous.
- We use the codeword *await* to make the function wait for a promise to be resolved.
- If an async function returns something, it will always be a promise. I.e. if you return something that is not a promise, it will be automatically “wrapped” in a promise.
- On the next slide we see the same code on promises as we have seen earlier.
- On the slide after, we see how we can do the same thing by using *async/await*.

Using *then*

```
var msgPromise = sayHello();
msgPromise.then(success1).then(success2).then(success3).catch(fail);

function success1(result) {
    console.log(result); //Hello!
    return sayCurse();
}

function success2(result) {
    console.log(result); //Darn!
    return sayGrunt();
}

function success3(result) {
    console.log(result); //Arghh!
}

function fail(err) {
    console.log(err); //Something bad happened
}
```

Using *async/await*

```
async function loadMsg() {  
  
    let promise1 = await sayHello();  
    let promise2 = await promise1.sayCurse();  
    let promise3 = await promise2.sayGrunt();  
  
    //          Hello!      Darn!      Arghh!  
    console.log(promise1, promise2, promise3);  
}  
  
loadMsg();
```

Using *async/await* and catching errors

- We can implement error handling by using try – catch:

```
async function loadMsg() {  
  
    try {  
        let promise1 = await sayHello();  
        let promise2 = await promise1.sayCurse();  
        let promise3 = await promise2.sayGrunt();  
  
        //          Hello!      Darn!      Arghh!  
        console.log(promise1, promise2, promise3);  
    }  
    catch(err) {  
        console.log(err);  
    }  
}  
  
loadMsg();
```