[Lesson 3]

Roi Yehoshua 2018

[DAN.IT EDUCATION]

What we learnt last time?

- Destructuring
- Closures
- Function declaration
- Named functional expressions
- Immediately invoked functional expressions
- Functions call and apply

[Our targets for today]

- Garbage collection
- Error Handling
- Using try/catch/final construction
- Throwing Errors

[DAN.IT EDUCATION]

[Garbage Collection]

→ A Lexical Environment object dies when it becomes unreachable: when no nested functions remain that reference it

→ In the code below, after **g** becomes unreachable, **value** is also cleaned from memory:

```
function f() {
    let value = 123;
    function g() { alert(value);

    return g;

}
let g = f(); // while g is alive its corresponding Lexical Environment lives

g = null; // ...and now the memory is cleaned up
```

→ JavaScript engines try to optimize that. They analyze variable usage and if it's easy to see that an outer variable is not used – it is removed.

[DAN.IT] EDUCATION

[Error Handling]

→ No matter how great we are at programming, sometimes our scripts have errors

→ They may occur because of our mistakes, an unexpected user input, an erroneous server response and for many other reasons

→ Usually, a script “dies” (immediately stops) in case of an error, printing it to console

→ But there’s a syntax construct try..catch that allows to “catch” errors and, instead of dying, do something more reasonable

[DAN.IT EDUCATION]

The try/catch Syntax

→ The try..catch construct has two main blocks: try, and then catch:

```
try {
    // code...

} catch (err) {
    // error handling
}
```

→ First, the code in try {...} is executed

→ If there were no errors, then catch(err) is ignored: the execution reaches the end of try and then jumps over catch

→ If an error occurs, then try execution is stopped, and the control flows to the beginning of catch(err)

→The err variable (can use any name for it) contains an error object with details about what's happened

[DAN.IT] EDUCATION

[try/catch Example]

→ An example for a runtime error that is caught in the catch block:

```
try {
    alert('Start of try runs');
    lalala; // error, variable is not defined!
    alert('End of try (never reached)');
} catch (err) {
    alert('Error has occurred!');
}

alert('...Then the execution continues');
```

→ try..catch only works for runtime errors or “exceptions”

→It won’t work if the code is syntactically wrong, e.g. it has unmatched curly braces:

```
try {
    {{{{{{{{{{{
} catch(e) {
    alert("The engine can't understand this code, it's invalid");
}
```

[DAN.IT] EDUCATION

[Error Object]

→ When an error occurs, JavaScript generates an object containing the details about it

→The object is then passed as an argument to catch:

```
try {
    // ...
} catch (err) { // <-- the "error object", could use another word instead of
err
    // ...
}
```

→The error object has the following properties:

→ **name** – the error name, e.g., “SyntaxError”, “ReferenceError”, “TypeError”

→ **message** – textual message about error details

→ **stack** – current call stack: a string with information about the sequence of nested calls that led to the error

→ Used for debugging purposes

[DAN.IT EDUCATION]

[Error Object Example]

```
try {
    lalala; // error, variable is not defined!
} catch (err) {
    alert(err.name); // ReferenceError
    alert(err.message); // lalala is not defined
    alert(err.stack); // ReferenceError: lalala is not defined at ...

    // Can also show an error as a whole
    // The error is converted to string as "name: message"
    alert(err); // ReferenceError: lalala is not defined
}
```

[DAN.IT] EDUCATION

Using try/catch

→ Let's explore a real-life use case of try..catch

→ As we already know, JavaScript supports the **JSON.parse**(str) method to read JSON-encoded values

→Usually it's used to decode data received over the network, from the server or another source

→ We receive it and call JSON.parse, like this:

```
let json = '{"name": "John", "age": 30}'; // data from the server

let user = JSON.parse(json); // convert the text representation to JS object

// now user is an object with properties from the string
alert(user.name); // John
alert(user.age); // 30
```

→ If json is malformed, JSON.parse generates an error, and the script "dies"

→This way, if something's wrong with the data, the visitor will never know that (unless he opens developer console)

[DAN.IT EDUCATION]

Using try/catch

→ Let's use try..catch to handle the error:

```
let json = "{ bad json }";

try {
    let user = JSON.parse(json); // <-- when an error occurs...
    alert(user.name); // doesn't work
} catch (e) {
    // ...the execution jumps here
    alert("Our apologies, the data has errors, we'll try to request it once more.");
    alert(e.name); // SyntaxError
    alert(e.message); // Unexpected token o in JSON at position 0
}
```

→ Here we use the catch block only to show an error message, but we can do much more: send a new network request, suggest an alternative to the visitor, send information about the error to a logging facility, etc.

[DAN.IT] EDUCATION

Throwing Errors

→ We can throw our own errors

→ The throw operator generates an error

→ The syntax is:

```
throw <error object>
```

→ We can throw anything as an error object

→ That may be even a primitive, like a number or a string, but it's better to use objects, preferrably with name and message properties

→ We can also throw one of JavaScript built-in error objects

→ Besides the generic Error constructor, there are seven other core error constructors: SyntaxError, TypeError, EvalError, InternalError, RangeError, ReferenceError, URIError

→ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

[DAN.IT EDUCATION]

Throwing Errors Example

→ Let's say that we get a json object that is syntactically correct, but doesn't have a required name property

→ We can treat the absence of name as a syntax error

→ So let's throw a SyntaxError exception:

```
let json = '{ "age": 30 }'; // incomplete data

try {
     let user = JSON.parse(json); // <-- no errors

     if (!user.name) {
          throw new SyntaxError("Incomplete data: no name"); // passing the message to the constructor
     }
     alert(user.name);
} catch (e) {
     alert("JSON Error: " + e.message); // JSON Error: Incomplete data: no name
}
```

[DAN.IT] EDUCATION

[Rethrowing Errors]

→In the example above we use try..catch to handle incorrect data

→But is it possible that *another unexpected error* occurs within the try {...} block

→**Catch should only process errors that it knows and “rethrow” all others**

→The “rethrowing” technique can be explained in more detail as:

- →Catch gets all errors
- →In catch(err) {...} block we analyze the error object err
- →If we don’t know how to handle it, then we do throw err

[DAN.IT EDUCATION]

Rethrowing Errors Example

```
let json = '{ "age": 30 }'; // incomplete data
try {
     let user = JSON.parse(json);
     if (!user.name) {
          throw new SyntaxError("Incomplete data: no name");
     }
     blabla(); // unexpected error
     alert(user.name);
} catch (e) {
     if (e instanceof SyntaxError) {
          alert("JSON Error: " + e.message);
     } else {
          throw e; // rethrow the error (*)
     }
}
```

→ The error throwing on line (*) from inside the catch block “falls out” of try..catch and can be either caught by an outer try..catch construct (if it exists), or it kills the script

[DAN.IT] EDUCATION

[try/catch/finally]

→ The try..catch construct may have one more code clause: finally

→ If it exists, it runs in all cases:

→ after try, if there were no errors

→ after catch, if there were errors

→ The extended syntax looks like this:

```
try {
   ... try to execute the code ...
} catch(e) {
   ... handle errors ...
} finally {
   ... execute always ...
}
```

→ The finally clause is often used when we start doing something before try..catch and want to finalize it in any case of outcome

[DAN.IT EDUCATION]

[try/catch/finally Example]

→ For instance, let's say we want to measure the time that a Fibonacci numbers function **fib**(n) takes

→ We can start measuring before it runs and finish afterwards

→ But what if there's an error during the function call? e.g., if the function receives negative or non-integer numbers

→ The finally clause is a great place to finish the measurements no matter what

→ In the code on the next slide **finally** guarantees that the time will be measured correctly in both situations – in case of a successful execution of **fib** and in case of an error in it

[DAN.IT EDUCATION]

try/catch/finally Example

```
let num = +prompt("Enter a positive integer number?", 35);
let diff, result;

function  fib(n) {
    if (n < 0 || Math.trunc(n) != n) {
        throw new Error("Must not be negative, and also an integer.");
    }
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
let start =

Date.now();  try {

    result = fib(num);
} catch (e) {
    result = 0;
} finally {
    diff = Date.now() - start;
}

alert(result || "error occured");
alert(`execution took ${diff}ms`);
```

[finally and return]

→The finally clause works for *any* exit from try..catch

→That includes an explicit return

→In the example below, there's a return in try

→In this case, finally is executed just before the control returns to the outer code

```
function func(){
    try {
        return 1;
    } catch  (e) {
        /* ... */
    } finally {
        alert('finally');
    }
}

alert(func()); // first works alert from finally, and then this one
```

[DAN.IT EDUCATION]

[Global Catch]

→Let's imagine we've got a fatal error outside of try..catch, and the script died

→Is there a way to react on such occurrences? We may want to log the error, show something to the user, etc.

→If we run the JavaScript code in a browser, we can assign a function to a special **window.onerror** property, that will run in case of an uncaught error

```
window.onerror = function(message, url, line, col, error) {
    // ...
};
```

→ There are also web-services that provide error-logging for such cases, like https://errorception.com or http://www.muscula.com

→You register at the service and get a piece of JS (or a script URL) from them to insert on pages

→That JS script has a custom window.onerror function

→When an error occurs, it sends a network request about it to the service

[DAN.IT EDUCATION]

[Error Handling – Summary]

A try catch finally statement

```
try {
  let x = parseInt(prompt("Enter a number", ""));
  if (isNaN(x)) {
    let e = new Error();
    e.message = "That wasn't a
    number";  throw e;
  }
}
catch (err) {
  alert("Something went wrong:" + err.message);
}
finally {
  . . .
}
```

Try to execute the code

Create a new error object

Check the input

Throw the error

Catch the thrown error

Handle the error

Finally executes on success or failure

[DAN.IT EDUCATION]

[Custom Errors]

→ When we develop an application, we often need our own error classes to reflect specific problems that may occur in our tasks

→For errors in network operations we may need HttpError, for database operations DbError, etc.

→Our errors should support basic error properties like message, name and stack

→But they also may have other properties of their own

→e.g. HttpError objects may have statusCode property with a value like 404 or 500

→ JavaScript allows to use **throw** with any argument, so technically our custom error classes don't need to inherit from Error

→ But if we inherit from Error, then it becomes possible to use obj instanceof Error to identify error objects. So it's better to inherit from it.

→ As we build our application, our own errors naturally form a hierarchy, for instance, HttpTimeoutError may inherit from HttpError, and so on1

[DAN.IT EDUCATION]

[Extending Error]

→ As an example, let's consider a function **readUser(json)** that should read JSON with user data

→Internally, it will use JSON.parse

→If it receives malformed json, then it throws SyntaxError

→But even if json is syntactically correct, that doesn't mean that it's a valid user

→For instance, it may not have some required properties, such as name and age

→Our function readUser(json) will not only read JSON, but check ("validate") the data

→If there are no required fields, it will throw a ValidationError, which will carry the information about the offending field

[DAN.IT EDUCATION]

[Extending Error]

→ Our ValidationError class should inherit from the built-in Error class

→ The Error class's code looks something like this:

```
// The "pseudocode" for the built-in Error class defined by JavaScript itself
class Error {
    constructor(message) {
        this.message = message;
        this.name = "Error"; // different names for different built-in error classes
        this.stack = <nested calls>; // non-standard, but most environments support it
    }
}
```

→ Now we will inherit ValidationError from it:

```
class ValidationError extends Error {
    constructor(message) {
        super(message); // the parent constructor sets the message property
        this.name = "ValidationError"; // reset the name property to its right
        value
    }
}
```

[DAN.IT] EDUCATION

Extending Error

→ Let's try to use it in readUser(json):

```
// Usage
function readUser(json) {
    let user = JSON.parse(json);
    if (!user.age) {
        throw new ValidationError("No field: age");
    }
    if (!user.name) {
        throw new ValidationError("No field: name");
    }
    return user;
}
```

```
// Working example with try..catch
try {
    let user = readUser('{ "age": 25 }');
} catch (err) {
    if (err instanceof ValidationError) {
        alert("Invalid data: " + err.message); //
Invalid data: No field: name
    } else if (err instanceof SyntaxError) {
        alert("JSON Syntax Error: " + err.message);
    } else {
        throw err; // unknown error, rethrow it
    }
}
```

→ The try..catch block in the code above handles both our ValidationError and the built-in SyntaxError from JSON.parse()

→ If it meets an unknown error, then it rethrows it, since the catch only knows how to handle validation and syntax errors, other kinds should fall through

[DAN.IT] EDUCATION

Exercise (1)

→The ValidationError class is very generic. Many things may go wrong.

→The property may be absent or it may be in a wrong format (like a string value for age)

→Create a more concrete class PropertyRequiredError, exactly for absent properties

→It should inherit from ValidationError and add the property “missingProperty” to it

→Set its error message to be “No property: [name of the missing property]”

```
class PropertyRequiredError extends ValidationError {
    // Your code here
}

function readUser(json) {
    let user = JSON.parse(json);
    if (!user.age) {
        throw new PropertyRequiredError("age");
    }
    if (!user.name) {
        throw new PropertyRequiredError("name");
    }
    return user;
}
```

```
try {
    let user = readUser('{ "age": 25 }');
} catch (err) {
    if (err instanceof ValidationError) {
        alert("Invalid data: " + err.message); // Invalid
data: No property: name
        alert(err.name); // PropertyRequiredError
        alert(err.missingProperty); // name
    } else if (err instanceof SyntaxError) {
        alert("JSON Syntax Error: " + err.message);
    } else {
        throw err; // unknown error, rethrow it
    }
}
```

[DAN.IT] EDUCATION

[Drag and Drop]

- Drag and drop is a very common feature: it allows you to take an object, drag it and drop it in another location
- This provides a simple way to do many things, from copying and moving files to ordering (drop into cart)
- The basic Drag'n'Drop algorithm looks like this:
  - Catch mousedown on a draggable element
  - Prepare the element to moving (maybe create a copy of it or whatever)
  - Then on mousemove move it by changing left/top and position:absolute
  - On mouseup (button release) – perform all actions related to a finished Drag'n'Drop

[DAN.IT EDUCATION]

Drag and Drop Example

```
<p>Drag the ball.</p>
<img id="ball" src="https://js.cx/clipart/ball.svg" style="cursor: pointer"/>

<script>
    let ball = document.getElementById("ball");

    Ball.onmousedown = function (event) {  // start the process
        // prepare to moving: make absolute and on top by z-index
        ball.style.position = "absolute";
        ball.style.zIndex = 1000;

        // move the ball out of any current parents directly into body
        // to make it positioned relative to the body
        document.body.append(ball);

        // put the absolute positioned ball under the cursor
        moveAt(event.pageX, event.pageY);

        // move the ball onmousemove
        // we track mousemove on document, not on ball, because mousemove doesn't trigger
        // for every pixel, thus the cursor may leave the ball's area on a swift move
        document.addEventListener("mousemove", onMouseMove);
```

[DAN.IT] EDUCATION

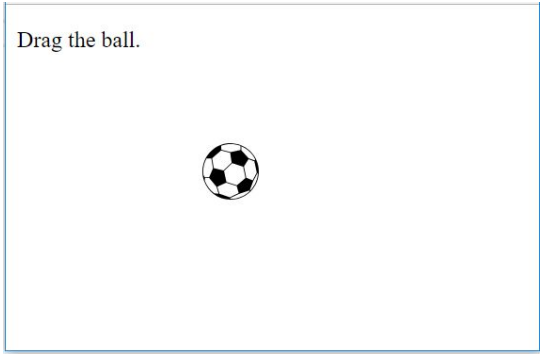
Drag and Drop Example

```
        // drop the ball, remove unneeded handlers
        ball.onmouseup = function (event) {
            document.removeEventListener("mousemove", onMouseMove);
            ball.onmouseup = null;
        }

        function onMouseMove(event) {
            moveAt(event.pageX, event.pageY);
        }

        // center the ball at (pageX, pageY) coordinates
        function moveAt(pageX, pageY) {
            ball.style.left = pageX - ball.offsetWidth / 2 + "px";
            ball.style.top = pageY - ball.offsetHeight / 2 + "px";
        }
    }

    // Disable the browser's default behavior for drag'n'drop
    ball.ondragstart = function (event) {
        event.preventDefault();
    }
</script>
```



[Exercise (2)]

- Create a slider:

- Drag the blue thumb with the mouse and move it
- Important details:
  - When the mouse button is pressed, during the dragging the mouse may go over or below the slider. The slider will still work (convenient for the user).
  - If the mouse moves very fast to the left or to the right, the thumb should stop exactly at the edge.
- Start with the HTML on the next slide

[DAN.IT] EDUCATION

[Exercise (3)]

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <style>
        .slider {
            border-radius: 5px;
            background: #E0E0E0;
            background: linear-gradient(left top, #E0E0E0, #EEEEEE);
            width: 310px;
            height: 15px;
            margin: 5px;
        }

        .thumb {
            width: 10px;
            height: 25px;
            border-radius: 3px;
            position: relative;
            left: 10px;
            top: -5px;
            background: blue;
            cursor: pointer;
        }
    </style>
</head>
```

```
<body>
    <div id="slider" class="slider">
        <div class="thumb"></div>
    </div>

    <script>
            // ...your code...
    </script>
</body>
</html>
```

[DAN.IT] EDUCATION

Control questions

1. What is garbage collector?

1. Which construct can we use to catch and process errors?
2. How can we throw an error event ourselves?
3. How can we catch a global error without special constructs?
4. How can we create Drag and Drop functionality?