Errors. Storages

UA Resource Development Unit

2020

AGENDA

ERRORS

RUNTIME ERROR

SYNTAX ERROR

SEMANTIC ERROR

description

Runtime errors result in new Error objects being created and thrown.

Error types

`EvalError`

`InternalError` ⚠

`RangeError`

`ReferenceError`

`SyntaxError`

`TypeError`

`URIError`

RangeError

Error that occurs when a numeric variable or parameter is outside of its valid range.

```
var pi = 3.14159;
pi.toFixed(100000);
```

ReferenceError

Error that occurs when a non-existent variable is referenced.

```
function foo() {
    bar++;
}
```

SyntaxError

Error that occurs when trying to interpret syntactically invalid code

```
var event = {
    title: " conference ",
    date: "today";
    var str = JSON.stringify(event);
    alert( str );
```

TypeError

Error that occurs when a variable or parameter is not of a valid type.

```
var foo = {};
foo.bar();
```

URIError

Error that occurs when a global URI handling function was used in a wrong way

```
decodeURIComponent("%");
```

Error

Error object can be used as a base object for user-defined exceptions.

Syntax:

```
new Error([message[, fileName[, lineNumber]]])
```

Example:

```
var error = new Error("error message");
```

Properties

`Error.prototype`

`Error.prototype.constructor`
Specifies the function that created an instance's prototype.

`Error.prototype.message`
Error message.

`Error.prototype.name`
Error name.

Vendor-specific extensions

Microsoft
`Error.prototype.description`
Error description. Similar to message.
`Error.prototype.number`
Error number.

Mozilla
`Error.prototype.fileName`
Path to file that raised this error.
`Error.prototype.lineNumber`
Line number in file that raised this error.
`Error.prototype.columnNumber`
Column number in line that raised this error.
`Error.prototype.stack`
Stack trace.

The “try…catch” syntax

The try statement consists of a try block, which contains one or more statements ({}must always be used, also for single statements), and at least one catch clause or a finally clause, or both.

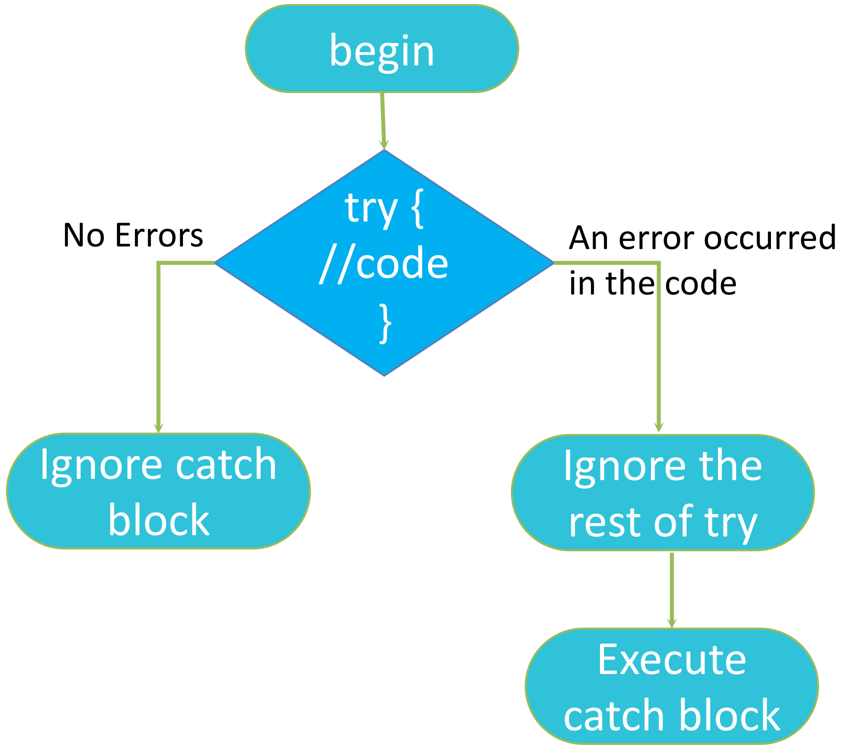
```
try {
    tryStatements
}
catch(exception){
    catchStatements
}
finally {
    finallyStatements
}
```

```
try...catch

try...finally

try...catch...finally
```

Error handling



the syntax construct try..catch allows to “catch” errors and, instead of dying, do something more reasonable

ReferenceError

Exception details

```
try {
    alort("call");
}
catch(e) {
    console.log (e);
    console.log ("Error Message: " + e.message);
    console.log ("Error Name: " + e.name);
    console.log ("Error lineNumber: " + e.lineNumber);
    console.log ("Error columnNumber: " + e.columnNumber);
    console.log ("Error fileName: " + e.fileName);
}
```

Conditional catch clauses

If you use an unconditional `catch` clause with one or more conditional `catch` clauses, the unconditional `catch` clause must be specified last. Otherwise, the unconditional `catch` clause will intercept all types of exception before they can reach the conditional ones.

```
try {
    myroutine(); // can throw three types of exceptions
} catch (e) {
    if (e instanceof TypeError) {
        // code for handling TypeError exceptions
    } else if (e instanceof RangeError) {
        // code for handling RangeError exceptions
    } else if (e instanceof EvalError) {
        // code for handling RangeError EvalError
    } else {
       // code to handle all other types of exceptions
       logMyErrors(e); // passing an exception to an exception handler
    }
}
```

try..catch works synchronously

```
try {
    setTimeout(function () {
        noSuchVariable; // script will die here
    }, 1000);
} catch (e) {
    alert("won't work");
}
```

try..catch actually wraps the setTimeout call that schedules the function. But the function itself is executed later, when the engine has already left the try..catch construct

```
setTimeout(function () {
    try {
        noSuchVariable; // try..catch handles the error!
    } catch (e) {
        alert("error is caught here!");
    }
}, 1000);
```

Nested try-blocks

Outer catch hadles the exception thrown in internal try block.

```
try {
    try {
        throw new Error('oops');
    }
    finally {
        console.log('finally');
    }
} catch (ex) {
    console.error('outer', ex.message);
}
```

throw

The throw statement throws a user-defined exception. Execution of the current function will stop (the statements after throw won't be executed), and control will be passed to the first catch block in the call stack. If no catch block exists among caller functions, the program will terminate.

```
throw 'Error242'; // generates an exception with a string value
throw 88; // generates an exception with the value 88
throw true; // generates an exception with the value true
```

Throw an object

You can specify an object when you throw an exception. You can then reference the object's properties in the catch block

```
function UserException(message) {
    this.message = message;
    this.name = 'UserException';
}
function getMonthName(mo) {
    mo = mo - 1; // Adjust month number for array index (1 = Jan, 12 = Dec)
    var months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
        'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
    if (months[mo] !== undefined) {
        return months[mo];
    } else {
        throw new UserException('InvalidMonthNo');
    }
}
try {
    // statements to try
    var myMonth = 15; // 15 is out of bound to raise the exception
    var monthName = getMonthName(myMonth);
} catch (e) {
    monthName = 'unknown';
    console.log(e.message, e.name); // pass exception object to err handler
}
```

Web Storage

The Web Storage API provides mechanisms by which browsers can securely store key/value pairs, in a much more intuitive fashion than using cookies.

Storage objects are simple key-value stores, similar to objects, but they stay intact through page loads. The keys and the values are always strings.

```
localStorage.colorSetting = '#a4509b';
localStorage['colorSetting'] = '#a4509b';
localStorage.setItem('colorSetting', '#a4509b');
```

Storage.getItem()
Storage.setItem()

Web Storage

The two mechanisms within Web Storage are as follows:

- **sessionStorage** maintains a separate storage area for each given origin that's available for the duration of the page session (as long as the browser is open, including page reloads and restores).
- **localStorage** does the same thing but persists even when the browser is closed and reopened.

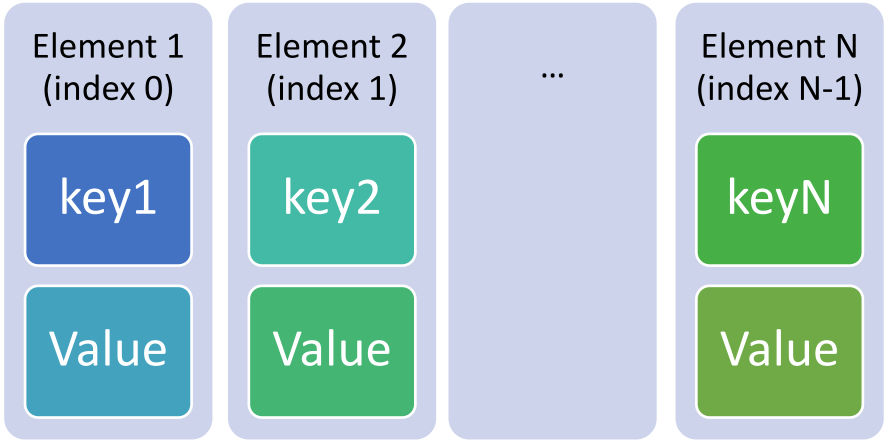
Storage Object Methods

Method	Description
key(*n*)	Returns the name of the *n*th key in the storage
getItem(*keyname*)	Returns the value of the specified key name
setItem(*keyname*, *value*)	Adds that key to the storage, or update that key's value if it already exists
removeItem(*keyname*)	Removes that key from the storage
clear()	Empty all key out of the storage

LOCAL STORAGE

The *read-only* localStorage property allows you to access a Storage object for the Document's origin; the stored data is saved across browser sessions. localStorage is similar to sessionStorage, except that while data stored in localStorage has no expiration time,

data stored in sessionStorage gets cleared when the page session ends — that is, when the page is closed.



EXAMPLE

```
// Save data to localStorage
localStorage.setItem('key', 'value');

// Get saved data from localStorage
localStorage.getItem('key'); // --> "value"

// Remove saved data from localStorage
localStorage.removeItem('key');

// Remove all saved data from localStorage
localStorage.clear();
```

LOCAL STORAGE EVENT

Also there is a special event for storages, it‘s called “storage”.

storage event is triggered each time the value in a storage is modified from another page or manually by user.

EXAMPLE

```
window.addEventListener('storage', ({ url, key, newValue: value }) => {
    console.log(`Key '${key}' changed value to '${value}' from '${url}'`);
});

// On the another page with the same domain...
localStorage.setItem('fired', true);
```

Working with data

Writing data to the repository

Reading data from the repository

Deleting data from the repository

length property

Getting values from storage

values can be retrieved from storage using Storage.getItem(). This takes the key of the data item as an argument and returns the data value.

```
localStorage.getItem(‘key’)
localStorage[“key"]
var key=localStorage.key(0)
```

```
function setStyles() {
    var currentColor = localStorage.getItem('bgcolor');
    var currentFont = localStorage.getItem('font');

    document.getElementById('bgcolor').value = currentColor;
    document.getElementById('font').value = currentFont;
    htmlElem.style.backgroundColor = '#' + currentColor;
    pElem.style.fontFamily = currentFont;
}
```

Setting values in storage

Storage.setItem() is used both to create new data items, and (if the data item already exists) update existing values. This takes two arguments — the key of the data item to create/modify, and the value to store in it.

localStorage.setItem(‘key', ‘value')
localStorage[“key"] = “value"

```
function populateStorage() {
    localStorage.setItem('bgcolor', document.getElementById('bgcolor').value);
    localStorage.setItem('font', document.getElementById('font').value);
    localStorage.setItem('image', document.getElementById('image').value);
}
```

Deleting data records

Web Storage also provides a couple of simple methods to remove data.

Storage.removeItem() takes a single argument — the key of the data item you want to remove — and removes it from the storage object for that domain.
Storage.clear() takes no arguments, and simply empties the entire storage object for that domain.

```
localStorage.removeItem(“key")
localStorage.clear()
delete localStorage[“key"]
```

Using localStorage

- Store the localStorage object in the container element
- The localStorage object stores data for an unlimited period of time
- The data stored in this object will be accessible even without an Internet connection.
- localStorage and sessionStorage restrict access to data by that domain, taking into account the protocol and port number in which this page is located

Storages

SESSION STORAGE

SESSION STORAGE

The *sessionStorage* property allows you to access a session Storage object for the current origin.

A page session lasts for as long as the browser is open and survives over page reloads and restores. **Opening a page in a new tab or window will cause a *new session* to be initiated**, which differs from how session cookies work.

Some browsers have ability to restore the session (and session storage with it). The API is the same as for *localStorage*.

support

```
if (window.sessionStorage && window.localStorage) {
    console.log(“sessionStorage and localStorage objects are support”);
}
else {
    console.log(“sessionStorage and localStorage objects are not support”);
}
```

What you can store and use for

- Application state
- User Settings
- Saving user data in case of unsuccessful attempt to send to the server.
- Recording data as they are entered

Available size

Unlike HTTP cookie, the web store provides a much larger size - 4Kb versus ~ 5Mb-10Mb in browser dependent

Checking storage space

```
try {
    localStorage.setItem('key', 'value');
} catch (e) {
    if (e == QUOTA_EXCEEDED_ERR) {
        alert(" ");
    }
}
```

Storages

COOKIES

COOKIES

Cookie is a *string* containing a semicolon-separated list of all cookies (key=value; key1=value1).

We can set an expiration date for the cookies. After expiration date is reached, cookies won’t be sent to the server with a request until the expiration date is renewed.

However every browser can behave differently from each other and some browsers can delete cookies after expiration date, while the others can store them for a long period of time after expiration.

Also we can set property ‘secure’ to *true* in cookies, so they can be sent only through HTTPS or SSL.

Setting ‘path’ property will allow access to the cookie only from the specific path.

Cookies can be HttpOnly, but you can’t set it with JS.

Reading from document.cookie

```
alert( document.cookie ); // cookie1=value1; cookie2=value2;...
```

The value of document.cookie consists of name=value pairs, delimited by ”;”
Each one is a separate cookie.

To find a particular cookie, we can split document.cookie by ”;” and then find the right name. We can use either a regular expression or array functions to do that.

We leave it as an exercise for the reader. Also, at the end of the chapter you’ll find helper functions to manipulate cookies.

Writing to document.cookie

We can write to document.cookie. But it’s not a data property, it’s an accessor (getter/setter). An assignment to it is treated specially.

A write operation to document.cookie updates only cookies mentioned in it but doesn’t touch other cookies.

```
document.cookie = "user=John"; // update only cookie named 'user'
alert(document.cookie); // show all cookies
```

If you run it, then probably you’ll see multiple cookies. That’s because document.cookie= operation does not overwrite all cookies. It only sets the mentioned cookie user.

COOKIES: Examples

EXAMPLE #1

```
document.cookie = 'cookie_key=a32; path=/our/location; domain=.site.com; secure=true';
// Set cookie named "cookie_key" with value "a32"
// That is accessible only from domain ".site.com"
// And only on path "/our/location" and everything after (like "/our/location/test")
```

EXAMPLE #2

```
const expiration = new Date();

// User logged in, his session expires after 1 day
expiration.setDate(expiration.getDate() + 1);
document.cookie = `user_token=unique123token; secure=true; expires=${expiration.toGMTString()}`;

// User logged out, remove cookie
document.cookie = `user_token=; expires=${(new Date(0)).toGMTString()}`;
```

DIFFERENCE

localStorage	sessionStorage	cookie
Can store from 2 to 10 MB	Can store from 2 to 10 MB	Can store 4 kB
Data lives until you delete it	Data lives until the end of session or until you delete it	Data lives until the end of expiration date* or until you delete it
Has an associated event	Has an associated event	Doesn’t have any associated event
Has special methods for writing and reading	Has special methods for writing and reading	Doesn’t have any method for writing and reading

JSON

The JSON object contains methods for parsing JavaScript Object Notation (JSON) and converting values to JSON.

JSON is a syntax for serializing objects, arrays, numbers, strings, booleans, and null. It is based upon JavaScript syntax but is distinct from it: some JavaScript is not JSON, and some JSON is not JavaScript.

JSON Syntax

Objects

Arrays

String

Number

Boolean value

The value is null.

```
{
    name1: "string",
    name2: 13,
    name3: true,
    name4: false,
    name5: null
}
```

JavaScript and JSON differences

JavaScript type	JSON differences
Objects and Arrays	Property names must be double-quoted strings; trailing commas are forbidden.
Numbers	Leading zeros are prohibited (in JSON.stringify zeros will be ignored, but in JSON.parse it will throw SyntaxError); a decimal point must be followed by at least one digit.

Strings	Only a limited set of characters may be escaped; certain control characters are prohibited
	`var code = '"\u2028\u2029"';`
	`JSON.parse(code); // works fine`
	`eval(code); // fails`

Methods

`JSON.parse()`

Parse a string as JSON, optionally transform the produced value and its properties, and return the value.

`JSON.stringify()`

Return a JSON string corresponding to the specified value, optionally including only certain properties or replacing property values in a user-defined manner.

example

```
let data = {};
if (localStorage.getItem(“key")) {
  data = JSON.parse(localStorage.getItem(“key"));
}
```

```
let obj = {
    id: 1,
    model: [123, "two", 3.0],
    description: "hello"
};
let serialObj = JSON.stringify(obj);
localStorage.setItem("myKey", serialObj);
```

Q&A

<epam>

DRIVEN CANDID CREATIVE ORIGINAL INTELLIGENT EXPERT

UA Frontend Online LAB