# TEHNICI WEB

# AJAX

*Claudia Chiriță . 2022/2023*

# OBIECTE

- un obiect este o colecție de perechi proprietate-valoare
- daca valoarea este o funcție, atunci proprietatea se numește metodă
- 

```
var ob = {prop1: val1, prop2: val2, ... , propn: valn};
```

- accesarea proprietăților:

```
ob.prop1; // val1
ob["prop1"]; // val1
```

# PROTOTIPURI

- prototipul unui obiect este desemnat prin

```
obiect.prototype
```

- orice obiect moștenește proprietățile obiectului prototip (prototypal inheritance)
- obiectele care au același prototip formează o clasă
- toate obiectele sunt descendenți ai obiectului generic *Object*

```
Object.getPrototypeOf() // prototipul obiectului specificat
```

# CREAREA OBIECTELOR

- prin object literal
- proprietățile, metodele, împreună cu valorile lor sunt enumerate între acolade
- se creează un singur obiect

```
var cat = {nume:"Tigger", culoare:"neagră", vârstă:14}
```

# CREAREA OBIECTELOR

- cu ajutorul obiectului generic
- se apelează constructorul new Object() și se adaugă apoi proprietățile și metodele
- se creează un singur obiect

```
var cat = new Object();
cat.nume = "Tigger";
cat.culoare = "neagră";
cat.vârstă = 14;
```

# CREAREA OBIECTELOR

- cu ajutorul unui constructor de obiecte
- se definește o funcție constructor(param) care apoi va fi apelată cu new constructor(param) pentru fiecare obiect care va fi creat

```
function cat(n, c, v) {
  this.nume = n;
  this.culoare = c;
  this.vârstă = v;
}

var c1 = new cat("Tigger", "neagră", 14);
var c2 = new cat("Fluff", "albă", 2);
```

# CREAREA OBIECTELOR

- cu metoda Object.create()
- ```
  Object.create(ob)
  ```

  creează un nou obiect, folosind un obiect existent *ob* ca prototip al obiectului nou creat
  ```

# EXEMPLU: INTERVAL

```
var interval = {
  mx: 2,
  my: 4,
  apartine: function(z) {
    return (z <= this.my) && (z >= this.mx);
  }
}; // clasa

var obi = Object.create(interval);
// obiect din clasa interval
obi.mx = 5;
obi.my = 7; // suprascriem proprietățile prototipului
```

# EXEMPLU: INTERVAL

```
var intervalD = Object.create(interval);
intervalD.apartine = function(z) {
  return (z < this.my) && (z > this.mx);
}; // subclasa

var obid = Object.create(intervalD);
obid.mx = 5;
obid.my = 10;
```

# EXEMPLU: INTERVAL

```
interval.valid = function() {
  return (this.my >= this.mx);
};

intervalD.vid = function() {
  return (this.mx == this.my);
};

alert(obid.valid()); // true
alert(obid.vid());  // false
```

# CUVÂNTUL-CHEIE THIS

- în interiorul unui constructor sau al unei metode asociate unui obiect, *this* se referă la obiectul curent
- într-o funcție folosită ca event handler, se referă la elementul pentru care este definit listenerul
- altfel, *this* se referă la obiectul *window*

# EXEMPLU: INTERVAL (2)

```
function Interval(x, y) {
  this.mx = x;
  this.my = y;
} // clasa

Interval.prototype.apartine = function(z) {
  return (z <= this.my) && (z >= this.mx);
} /* metodă adaugată în prototipul obiectelor
     create cu funcția constructor */
```

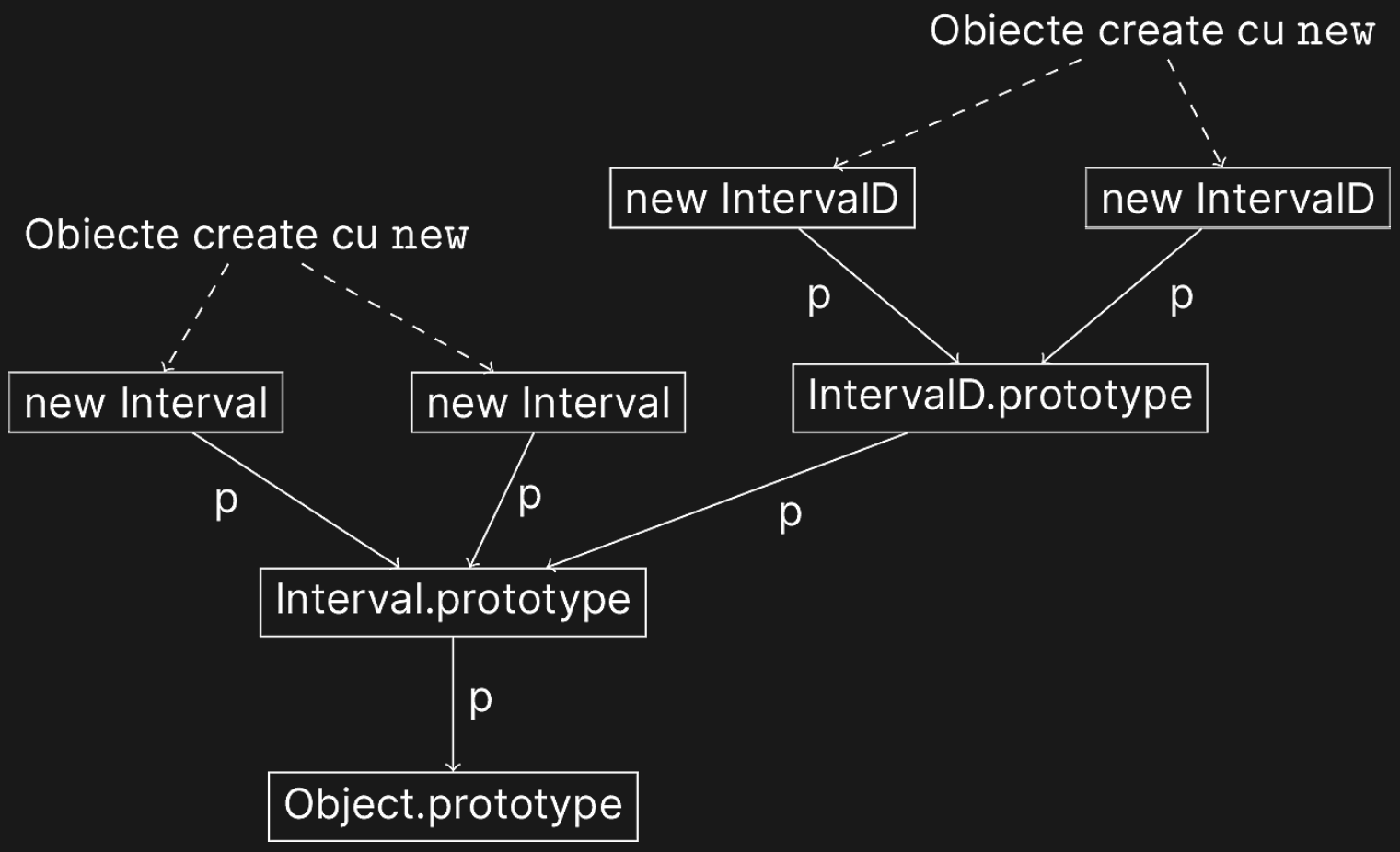
# EXEMPLU: INTERVAL (2)

```
var obi = new Interval(1, 4); // obiect din clasa Interval

Interval.prototype.valid = function() {
  return (this.my >= this.mx);
};

alert(obi.valid()); // true
```

# EXEMPLU: INTERVAL (2)



# EXEMPLU: INTERVAL (2)

## definirea subclaselor

```
function IntervalD(x, y) {
  Interval.call(this, x, y);
} // this este obiectul care se construiește

IntervalD.prototype = Object.create(Interval.prototype);
// schimbăm prototipul obiectelor create cu IntervalD
IntervalD.prototype.constructor = IntervalD;
// restaurăm proprietatea constructor

IntervalD.prototype.apartine = function(z) {
  return (z < this.my) && (z > this.mx);
};

var obid = new IntervalD(5, 10);
```

# EXEMPLU: INTERVAL (2)

## definirea subclaselor

```
function IntervalD(x, y) {
  Interval.call(this, x, y);
}
IntervalD.prototype = Object.create(Interval.prototype);
IntervalD.prototype.constructor = IntervalD;

IntervalD.prototype.apartine = function(z) {
  return (z < this.my) && (z > this.mx);
};
IntervalD.prototype.valid = function() {
  return (Interval.valid.call(this) && (this.mx != this.my));
}
```

# XML

Extensible Markup Language

# XML

- limbaj de marcare similar cu HTML
- nu conține taguri predefinite, utilizatorul își definește propriile taguri și structura documentului
- conceput pentru stocarea și transmiterea datelor
- independent de software și hardware
- utilizat pentru a crea limbaje de marcare precum XHTML, SVG, MathML

# STRUCTURA UNUI DOCUMENT XML

- Un document XML este format din:
  - elemente: taguri <nume_tag>
    case sensitive
  - date caracter: conținutul elementelor

# STRUCTURA UNUI DOCUMENT XML

- se definește versiunea XML și codificarea caracterelor (prima linie în document)

```
<?xml version="1.0" encoding="UTF-8"?>
```

- document XML: structură arborescentă
- toate documentele XML trebuie să conțină un element rădăcină (root)

```
<root>
  <child>
    <subchild>...</subchild>
  </child>
</root>
```

# SINTAXA XML

- există un singur element root într-un document XML
- fiecare element trebuie să aibă tag de închidere
- elementele XML trebuie să fie corect imbricate
- atributele asociate elementelor nu pot conține mai multe valori

# ENTITĂȚI

- pentru a ne referi la caractere rezervate

| Entity | Character | Description |
|---|---|---|
| &lt; | < | Less than sign |
| &gt; | > | Greater than sign |
| &amp; | & | Ampersand |
| &quot; | " | One double-quotation mark |
| &apos; | ' | One apostrophe (or single-quotation mark) |

# EXEMPLU

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore> // elementul rădăcină
   <book category="fantasy"> // element copil
      <title lang="en">Piranesi</title>
      <author>Susanna Clarke</author>
      <year>2020</year>
      <price>30.00</price>
    </book>
    <book category="science-fiction">
       <title lang="en">Klara and the Sun</title>
       <author>Kazuo Ishiguro</author>
       <year>2021</year>
       <price>29.99</price>
    </book>
</bookstore>
```

# XML-PARSER

- browserele au un analizor XML încorporat pentru a accesa și manipula documente XML
- înainte ca un document XML să poată fi accesat, acesta trebuie convertit într-un obiect XML DOM

# XML-PARSER

- XML DOM defineste proprietăți și metode pentru accesarea și editarea XML
  - proprietăți: nodeName, nodeValue, parentNode, childNodes, attributes
  - metode: getElementsByTagName(), appendChild(), removeChild()

# XML-PARSER

```
text = "<bookstore><book>" +"<title>Piranesi</title>" +
"<author>Susanna Clarke</author>" +"<year>2020</year>" +
"</book></bookstore>"; // XML ca string

parser = new DOMParser(); // se creează un analizor XML DOM

xmlDoc = parser.parseFromString(text,"text/xml");
// se creează un obiect XML DOM din stringul text

xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValu
// extragem informația din nodurile XML DOM
```

# JSON

JavaScript Object Notation

# JSON

- oferă o modalitate de reprezentare a datelor, ca alternativă la XML
- bazat pe JavaScript; în prezent un format independent de limbaj
- multe limbaje pot prelucra date în format JSON
- folosit pentru schimbul de informații cu serverul

# JSON

## elemente de bază:

```
Object: {"cheie1":val1, "cheie2":val2}
Array: [val1, val2, val3]
Value: string, number, object, array, true, false, null
```

## date.json

```
[{"pers": {"nume":"Klara", "varsta":12} },
 {"pers": {"nume":"Josie", "varsta":14} } ]
```

# SINTAXA JSON

- câmpul **cheie** trebuie să fie scris cu ghilimele

```
"nume":"Klara"
```

- câmpul **valoare** poate fi:

```
string, number, obiect (JSON), array, boolean, null
```

nu poate fi function, date, undefined

- obiectele JSON sunt reprezentate între acolade

```
{"nume":"Klara", "varsta":12, "porecla":"Robo"}
```

# EXEMPLU JSON

```
JSON String:   { "nume":"Klara" }
JSON Number:   { "varsta":12 }
JSON Object:   { "pers": { "nume":"Josie", "varsta":14 } }
JSON Array:    { "copii": [ "Klara", "Josie", "Rick" ] }
JSON Boolean: { "robot": true }
JSON null:     { "porecla": null }
```

# OBIECTE JSON

```
myObj = { "cheie1":val1, "cheie2":val2, "cheie3":val3 }
```

- accesarea obiectelor:

```
myObj.cheie1

myObj["cheie1"]
```

# OBIECTE JSON

- iterarea proprietăților unui obiect:

```
var myObj = { "copil":"Klara", "varsta":12, "robot":t
for (x in myObj) {
   document.getElementById("prop").innerHTML += x + "
}
```

```
<p id="prop"></p>
```

paragraful va conține:
copil
varsta
robot

# OBIECTE JSON

- iterarea proprietăților unui obiect:

```
var myObj = { "copil":"Klara", "varsta":12, "robot":tru
for (x in myObj) {
   document.getElementById("val").innerHTML += myObj[x]
}
```

```
<p id="prop"></p>
```

paragraful va conține: Klara 12 true

# OBIECTE JSON ÎNCORPORATE

```
myObj = { "nume":"Klara",
          "varsta":12,
          "componente": {"baterie:"solara", "motoare":"10"} }
```

- accesarea obiectelor încorporate:

```
myObj.componente.motoare // 10

myObj.componente["motoare"] // 10
```

# OBIECTE JSON ÎNCORPORATE

```
myObj = { "nume":"Klara",
          "varsta":12,
          "componente": {"baterie":"solara", "motoare":"10"} }
```

- modificarea valorilor proprietăților:

```
myObj.componente.motoare="11";
```

- ștergerea proprietăților:

```
delete myObj.componente.baterie;
```

# JSON ARRAYS

```
[val1, val2, ...., valn]
```

val1,...,valn pot fi string, number, object, array, boolean, null

- Array în interiorul obiectelor JSON

```
myObj = { "nume":"Klara",
          "varsta":12,
          "componente": ["baterie", "motoare", "senzo
```

accesarea valorilor:

```
myObj.componente[0] // baterie
```

# JSON ARRAYS

- Array în interiorul obiectelor JSON

```
myObj = { "nume":"Klara",
          "varsta":12,
          "componente": ["baterie", "motoare", "senzo
```

iterarea valorilor în Array:

```
for (i in ob.componente){ x += ob.componente[i]; }

for (i = 0; i < ob.componente.length; i++) {
   x += ob.componente[i];
}
```

# OBIECTUL JSON ÎN JAVASCRIPT

```
JSON.stringify(valoare)
// transformă un obiect JavaScript într-un string JSON

JSON.parse(text)
 // transformă un string JSON într-un obiect JavaScript
```

poate fi folosit pentru memorare în *localStorage* și *sessionStorage*

# OBIECTUL JSON ÎN JAVASCRIPT

```
var o1 = {copil: {nume:"Klara", varsta:12}},
    o2 = {copil: {nume:"Josie", varsta:14}},
    o = [o1,o2];

var s = JSON.stringify(o);
// "[{"copil":{"nume":"Klara","varsta":12}},{"copil":{"nume":"

localStorage.setItem("myarray", s);
var st = localStorage.getItem("myarray");

var jo = JSON.parse(st);
```

# AJAX

Asynchronous JavaScript and XML

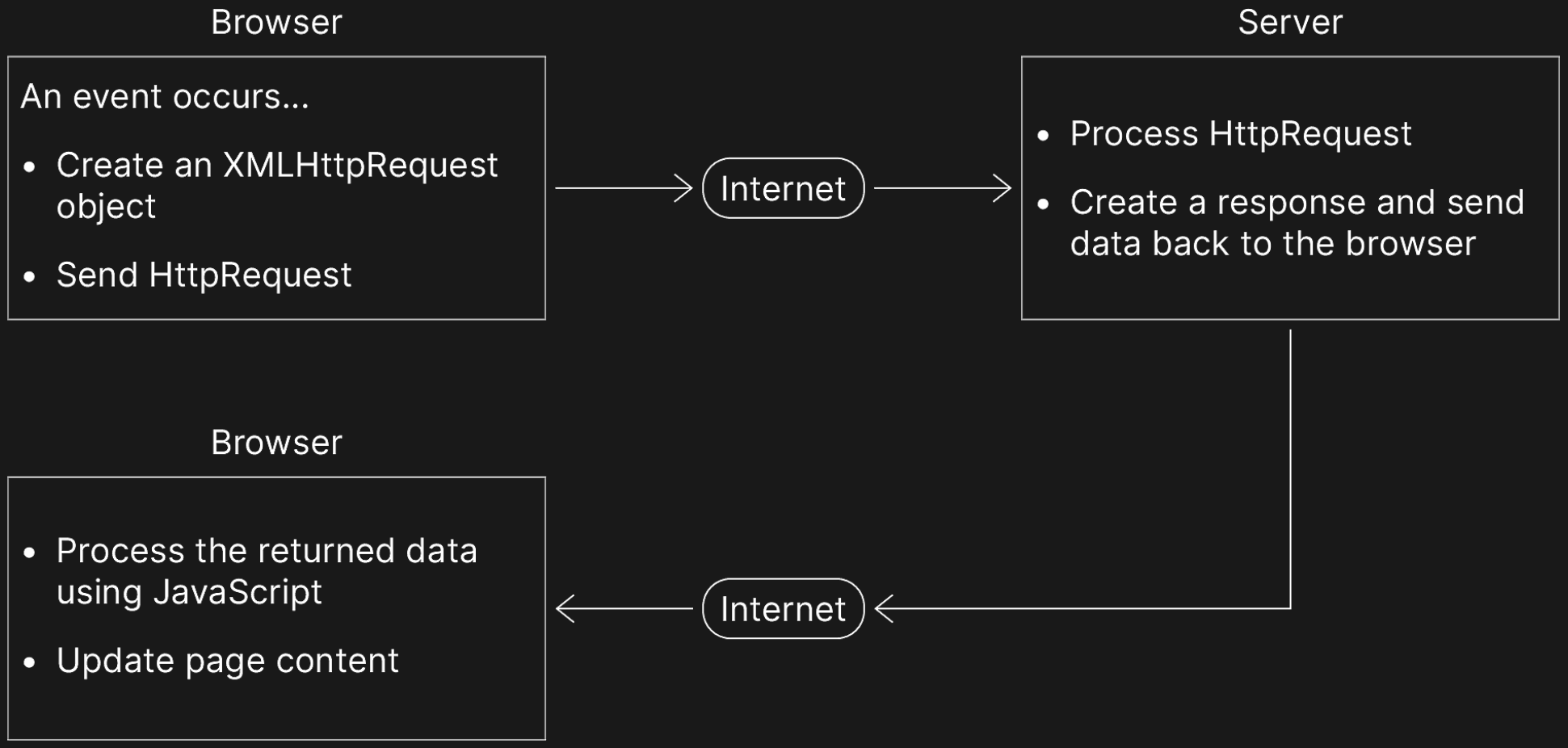
# AJAX

- termenul AJAX a fost introdus de Jesse James Garrett în 2005
- azi se referă la un grup de tehnologii folosite pentru procesele client ce descarcă informație de la server fără reîncărcarea completă a unei pagini web
- inițial se foloseau documente XML pentru request-uri și response-uri
- acum formatul JSON e mult mai uzual

# AJAX

- permite actualizarea unor părți ale unei pagini web fără reîncărcarea completă a paginii
- trimite cereri către un server web și citește datele primite de la server
- nucleul său îl reprezintă obiectul *XMLHttpRequest* care este folosit pentru a schimba date asincron cu serverul web

# AJAX


Browser
An event occurs...
Create an XMLHttpRequest object
Send HttpRequest
Internet
Server
Process HttpRequest
Create a response and send data back to the browser
Browser
Process the returned data using JavaScript
Update page content
Internet

# XMLHTTPREQUEST

- obiect JavaScript ce permite trimiterea de cereri către un server și primirea rezultatului în script
- pot fi procesate în paralel mai multe conexiuni cu serverul, fără blocarea browser-ului până la primirea răspunsului
- înainte de utilizarea XMLHttpRequest, trebuie creată o instanță

```
var xhr = new XMLHttpRequest()
```

# XMLHTTPREQUEST

- pune la dispoziție mai multe proprietăți și metode pentru comunicarea client/server

```
open() // creează cererea
send() // trimite cererea

readyState // starea cererii (0,1,2,3,4)
onreadystatechange
// funcția care se execută la schimbarea stării cererii

status // codul de stare HTTP (200 OK)
responseText
// răspunsul primit de la server în format text
responseXML
// răspunsul primit de la server în format XML
```

# CERERI HTTP

```
open( method, url, async ) // specifică tipul de cerere
/* method: poate fi GET sau POST
   url: adresa serverului
   async: true (asynchronous) sau false (synchronous) */
```

```
send() // trimite cererea către server ( GET )
send(date) // trimite cererea către server ( POST )
```

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://localhost:3000/test.txt", true);
xhr.send();
```

# GESTIONAREA RĂSPUNSULUI

- proprietatea *readyState* reprezintă starea cererii:
  0. cererea este neinițializată
  1. conexiune stabilită cu serverul
  2. cererea a fost primită
  3. se procesează cererea
  4. cererea este finalizată și răspunsul este gata
- proprietatea *onreadystatechange* definește o funcție care trebuie executată când se schimbă *readyState*

```
xhr.onreadystatechange = nume-functie;
```

# GESTIONAREA RĂSPUNSULUI

- proprietatea *status*: codul de stare HTTP al răspunsului de la server, în format numeric (200 "OK", 403 "Forbidden", 404 "Not found")
- proprietatea *statusText:* statusul în format text ("OK", "Not Found")

# GESTIONAREA RĂSPUNSULUI

- proprietatea *responseText*: returnează răspunsul primit de la server, în format text (string)
- proprietatea *responseXML:* returnează răspunsul primit de la server, în format XML

# EXEMPLU - TEXT

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
    <script>window.onload=function() {
   var httpRequest;

document.getElementById("ajaxButton").addEven
tListener('click', makeRequest);
   function makeRequest() {
      httpRequest = new XMLHttpRequest(); //
creează un obiect XMLHttpRequest
      if (!httpRequest) {
         alert('Giving up :( Cannot create an
XMLHTTP instance');
         return false;
      }
   httpRequest.onreadystatechange
=alertContents;
   httpRequest.open('GET', 'test.html');
   httpRequest.send();
   }
   function alertContents() {
      if (httpRequest.readyState == 4) {
         if (httpRequest.status == 200) {
```

# GESTIONAREA RĂSPUNSULUI

- Obiectul XML DOM în JavaScript poate fi parcurs cu metode asemănătoare celor din DOM

```
<?xml version="1.0" encoding="UTF-8"?>
<persoane>
  <pers nume="Klara" varsta="12"></pers>
  <pers nume="Josie" varsta="14"></pers>
 </persoane>
```

```
var xml = httpRequest.responseXML;
var vpers= xml.getElementsByTagName('pers');
alert(vpers[0].getAttribute('nume'));
```

# EXEMPLU - XML

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
    <script>window.onload=function() {
   var httpRequest;

document.getElementById("ajaxButton").addEven
tListener('click', makeRequest);
   function makeRequest() {
      httpRequest = new XMLHttpRequest(); //
creează un obiect XMLHttpRequest
      if (!httpRequest) {
         alert('Giving up :( Cannot create an
XMLHTTP instance');
         return false;
      }
   httpRequest.onreadystatechange =
alertContents;
   httpRequest.open('GET', 'test.xml');
   httpRequest.send();
   }
   function alertContents() {
      if (httpRequest.readyState == 4) {
         if (httpRequest.status == 200) {
```

# EXEMPLU - JSON

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
    <script>window.onload=function() {
   var httpRequest;

document.getElementById("ajaxButton").addEven
tListener('click', makeRequest);
   function makeRequest() {
      httpRequest = new XMLHttpRequest(); //
creează un obiect XMLHttpRequest
      if (!httpRequest) {
         alert('Giving up :( Cannot create an
XMLHTTP instance');
         return false;
      }
   httpRequest.onreadystatechange =
alertContents;
   httpRequest.open('GET',
'https://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-
store/products.json');
   httpRequest.send();
   }
```

# TRIMITEREA FORMULARELOR

```
<!DOCTYPE html>
<html>


  <head>
     <link rel="stylesheet" href="demo.css">
  </head>
  <body>
    <form
action="https://www.w3schools.com/action_page
.php">
       <label for="fname">Prenume:</label>
<br>
       <input type="text" id="fname"
name="fname" value="Klara"><br>
       <label for="lname">Nume:</label><br>
       <input type="text" id="lname"
name="lname" value="Sun"><br><br>
       <input type="submit" value="Trimite">
    </form>
```

❯

# TRIMITEREA FORMULARELOR VIA JS

## cu JSON?

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
     <script>window.onload=function() {
   var httpRequest;
   var myform =
document.getElementById("testform");
   myform.onsubmit = function (event){
      event.preventDefault();

      var data =
{prenume:document.getElementById("fname").val
ue,
nume:document.getElementById("lname").value};

      httpRequest = new XMLHttpRequest(); //
creează un obiect XMLHttpRequest
      if (!httpRequest) {
```

# TRIMITEREA FORMULARELOR VIA JS

## cu FormData

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
<script>window.onload=function() {
   var httpRequest;
   var myform =
document.getElementById("testform");
   var FD;
   myform.onsubmit = function (event){
      event.preventDefault();
      FD = new FormData(myform);
   httpRequest = new XMLHttpRequest(); //
creează un obiect XMLHttpRequest
   if (!httpRequest) {
      alert('Giving up :( Cannot create an
XMLHTTP instance');
      return false;
   }
```

# ASYNC JAVASCRIPT

# ASYNC JAVASCRIPT

- sync vs. async programs
- event handlers sunt un tip de programare asincronă: apelăm o funcție (handlerul) nu imediat, ci când se declanșează un eveniment
  event -> încheierea operației asincrone
- XMLHttpRequest - API asincron bazat pe evenimente: adăugăm event listenere obiectului XMLHttpRequest

# ASYNC JAVASCRIPT

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
     <script>window.onload=function() {
   var httpRequest;

document.getElementById("ajaxButton").addEven
tListener('click', makeRequest);
   function makeRequest() {
      httpRequest = new XMLHttpRequest(); //
creează un obiect XMLHttpRequest

   httpRequest.addEventListener('loadend', ()
=> {
     alert('Finished with status:' +
httpRequest.status);});

   httpRequest.open('GET', 'test.html');
   httpRequest.send();
   }
 }
    </script>
  </head>
  <body>
```

# CALLBACKS

- un event handler e un tip particular de callback
- callback = funcție transmisă unei alte funcții ca parametru, în vederea apelării ei la momentul potrivit
- callbacks în callbacks = *callback hell* sau *pyramid of doom*
- API-urile asincrone moderne nu folosesc callbacks

# PROMISES

- cod producător: de obicei necesită mult timp și întoarce un rezultat (ex: ia date dintr-o bază de date)
- cod consumator: dorește rezultatul *codului producător* de îndată ce este gata
- promise: obiect JavaScript care leagă *codul producător* și *codul consumator*

# PROMISES

- promise: obiect returnat de o funcție asincronă, ce reprezintă starea curentă a operației executate
- la momentul returnării promisiunii în funcția caller, de regulă operația nu s-a terminat; obiectul oferă însă metode pentru gestiunea succesului sau eșecului operației

# PROMISES

## crearea unei promisiuni

```
promise = new Promise( function(resolve, reject) {
  /* cod producător care se execută asincron:
     setInterval, setTimeout, cereri Ajax, evenimente */
  resolve(value) // pentru succes
  reject(error) // pentru eroare
} );
```

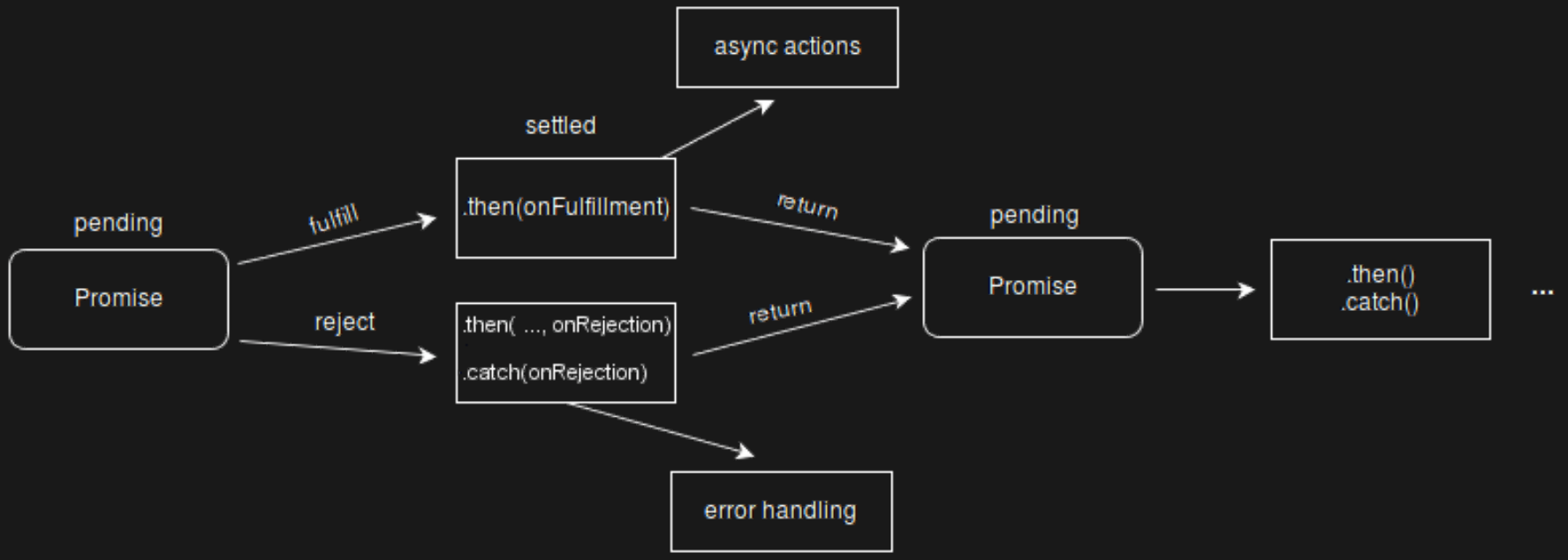
# PROMISES

- la crearea unei promisiuni, se apelează automat funcția transmisă ca argument promisiunii (numită funcție *executor*)
- *executorul* are ca parametri două funcții predefinite (resolve & reject) și conține *codul producător* care ar trebui să producă în cele din urmă rezultatul dorit

# PROMISES

- în funcție de rezultatul obținut (codul asincron va decide dacă e cu succes sau eșec) se va apela una din cele două funcții (functia de succes - resolve sau funcția de eșec - reject)
- un obiect promise se poate afla într-una din stările: pending (în așteptare), fulfilled (îndeplinită), rejected (respinsă)

# PROMISES


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
.then()
.catch()
...
reject
.then( ..., onRejection)
.catch(onRejection)
return
error handling

# PROMISES

```
// cod consumator care așteaptă rezultatul
promise.then(function (value) { /* cod pentru succes */ },
             function(error)  { /* cod pentru eroare */ })
       .catch(function(error) { /* tratarea erorii */ })
```

- *funcțiile consumatoare* pot fi înregistrate folosind metodele .then, .catch și .final

# PROMISES

- primul argument al lui .then este o funcție care rulează atunci când promisiunea este rezolvată și primește rezultatul așteptat
- al doilea argument al lui .then este o funcție care rulează atunci când promisiunea este respinsă și primește eroarea

# PROMISES

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
     <script>window.onload=function() {
    /* let p = new Promise(function(resolve,
reject) { // se creează o promisiune
       setTimeout(function(){
          var nota = Math.floor(Math.random()
* 10) + 1;
          if(nota >= 5)
            resolve("Bravo, ai promovat cu
nota " + nota+ "!"); // apelăm funcția pentru
succes
          else
            reject(nota); // apelăm funcția
pentru eșec
          },2000)});
   console.log("Aștept răspunsul...");
   p.then(succes, esec).catch(function(err)
{console.log(err)}); // dacă primim o eroare,
o afișăm
}
  function succes(rez){ // funcția pentru
succes
```

# FETCH

- API pentru efectuarea unei cereri Ajax în JavaScript bazat pe promisiuni
- alternativă modernă la XMLHttpRequest
- metoda fetch():

```
const promise = fetch(url, [options]);
/* întoarce o promisiune folosită pentru a obține
   răspunsul de la server

   url: adresa url a severului
   options: method, headears, body (opționale) */
```

# FETCH

```
const promise = fetch(url, [options]);
promise.then(function(response) {
                return response.text();
              });
```

- promisiunea returnată de fetch() este rezolvată cu un obiect din clasa Response imediat ce serverul răspunde cu un header
- promisiunea este respinsă dacă cererea HTTP nu poate fi trimisă (există probleme de conexiune) sau nu se primește răspunsul

# FETCH

## cerere Ajax cu XMLHttpRequest

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
    <script>window.onload = function() {
   /* var data;
   var httpRequest = new XMLHttpRequest();;
   httpRequest.open('GET',
'https://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-
store/products.json');
   httpRequest.onreadystatechange =
function() {
      if (httpRequest.readyState == 4) {
         if (httpRequest.status == 200) {
            alert(httpRequest.status + "
OK");
         } else {
            alert('There was a problem with
the request.');
         }
      }
```

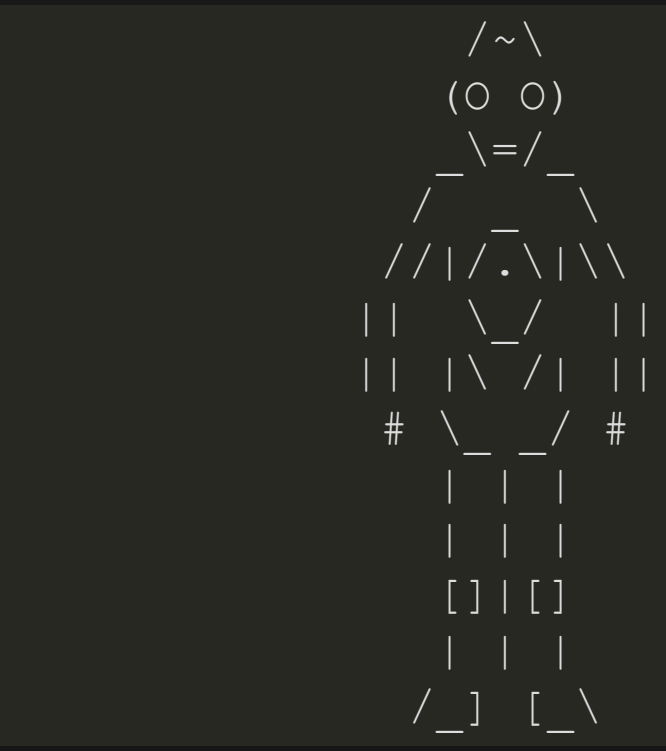
# FETCH

## cerere Ajax cu Fetch

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
    <script>window.onload = function() {
   /* var data;
   var promiseFetch =
fetch('https://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-
store/products.json');

   promiseFetch.then(function(response){
       if(response.status == '200')
         return response.status;
       else
         throw "eroare";})
       .then(function(status) {
         alert(status);})
       .catch(function(err){
         alert(err);});   */
}
   </script>
```

# TRIMITEREA FORMULARELOR VIA FETCH

```
<!DOCTYPE html>
<html>
  <head>
     <link rel="stylesheet" href="demo.css">
     <script>window.onload=function() {
   var httpRequest;
   var myform =
document.getElementById("testform");
   myform.onsubmit = function (event){
      event.preventDefault();

      var data =
{prenume:document.getElementById("fname").val
ue,
nume:document.getElementById("lname").value};

     fetch(myform.action,
           { method:"post",
             headers: {'Content-Type':
'application/json'}
```



întrebări?