

# **Events, callbacks and event loop**

# How events work

```
// Event handler
function sayHello(){
    document.getElementById("output").innerHTML = "Hello World";
}
```

```
// Target element to bind event
let elementObj = document.getElementById('myElement');
```

```
// Bind this way
elementObj.addEventListener("click", sayHello);
```

```
// Or better this way
elementObj.onclick = sayHello;
```

# Mouse Events

<b>onclick</b>	user presses/releases mouse button on the element
<b>ondblclick</b>	user presses/releases mouse button twice on the element
<b>onmousedown</b>	user presses down mouse button on the element
<b>onmouseup</b>	user releases mouse button on the element movement
<b>onmouseover</b>	mouse cursor enters the element's box
<b>onmouseout</b>	mouse cursor exits the element's box
<b>onmousemove</b>	mouse cursor moves around within the element's box

# The event object

Event handlers can accept an optional parameter to represent the event that is occurring. Event objects have the following properties and methods:

```
function handler(evt) {  
    // an event handler function ...  
}
```

**target**

The element on which the event handler was registered

**preventDefault()**

Prevents browser from performing its usual action in response to the event

**stopPropagation()**

Prevents the event from bubbling up further

**stopImmediatePropagation()**

Prevents the event from bubbling and prevents any other handlers from being executed

# Stopping an event's browser behavior

To abort a form, submit or another event's default browser behavior, call jQuery's **preventDefault()** method on the event

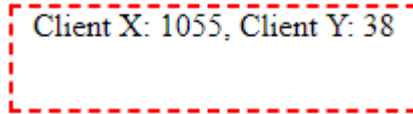
```
<form id="exampleform" action="">...</form>
```

```
exampleform.addEventListener('submit', validateData);
```

```
function validateData(event) {  
    if (!city.value || !state.value) {  
        alert("Error, invalid city/state."); // show error message  
        event.preventDefault();  
    }  
}
```

# Using event object

```
<div style="width: 200px; height: 50px;  
border: dashed red 2px;  
margin: auto; text-align: center" id="output">  
    Move over me!  
</div>
```



Client X: 1055, Client Y: 38

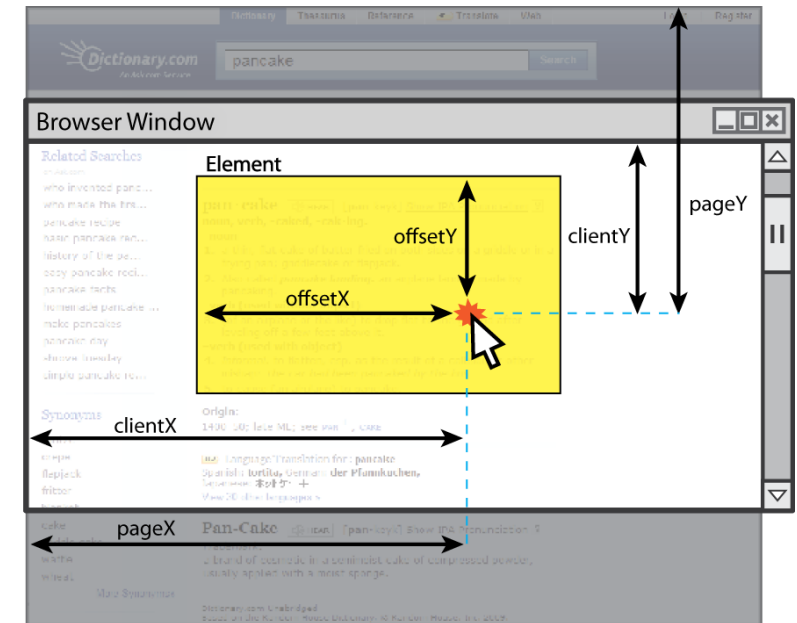
```
window.onload = function () {  
    document.getElementById("output").onmousemove = displayMouseCords;  
}  
  
function displayMouseCords(e) {  
    document.getElementById("output").innerHTML = "Client X: " + e.clientX +  
        ", Client Y: " + e.clientY;  
}
```

# Mouse Event Object

The **event** object is passed to a mouse handler has these properties:

**clientX, clientY**  
**screenX, screenY**  
**offsetX, offsetY**  
**pageX, pageY**  
**which**

coordinates in browser window  
coordinates in screen  
coordinates in element (non-standard)  
coordinates in entire web page  
which mouse button was clicked



# Page/window events

- onload, onunload** the browser loads/exits the page
- onresize** the browser window is resized
- onerror** an error occurs when loading a document or an image
- oncontextmenu** the user right-clicks to pop up a context menu

The above can be handled on the `window` object.

# Form events

- onsubmit** form is being submitted
- onreset** form is being reset
- onchange** the text or state of a form control has changed



# Keyboard/text events

- onkeydown** user presses a key while this element has keyboard focus
- onkeyup** user releases a key while this element has keyboard focus
- onkeypress** user presses and releases a key while this element has keyboard focus
- onfocus** this element gains keyboard focus
- onblur** this element loses keyboard focus
- onselect** this element's text is selected or deselected)

## Keyboard event object properties

- which** ASCII integer value of key that was pressed (convert to char with `String.fromCharCode`)
- altKey, ctrlKey, shiftKey** true if Alt/Ctrl/Shift key is being held

# Event Bubbling

# Which element gets the event?

```
<body>
  <div>
    <p> Events are <em>crazy</em>! </p>
  </div>
</body>
<script>
  document.querySelectorAll("body, div, p, em").forEach(e => e.onclick = hello);

  function hello() {
    alert("You clicked on the " + this.nodeName);
  }
</script>
```

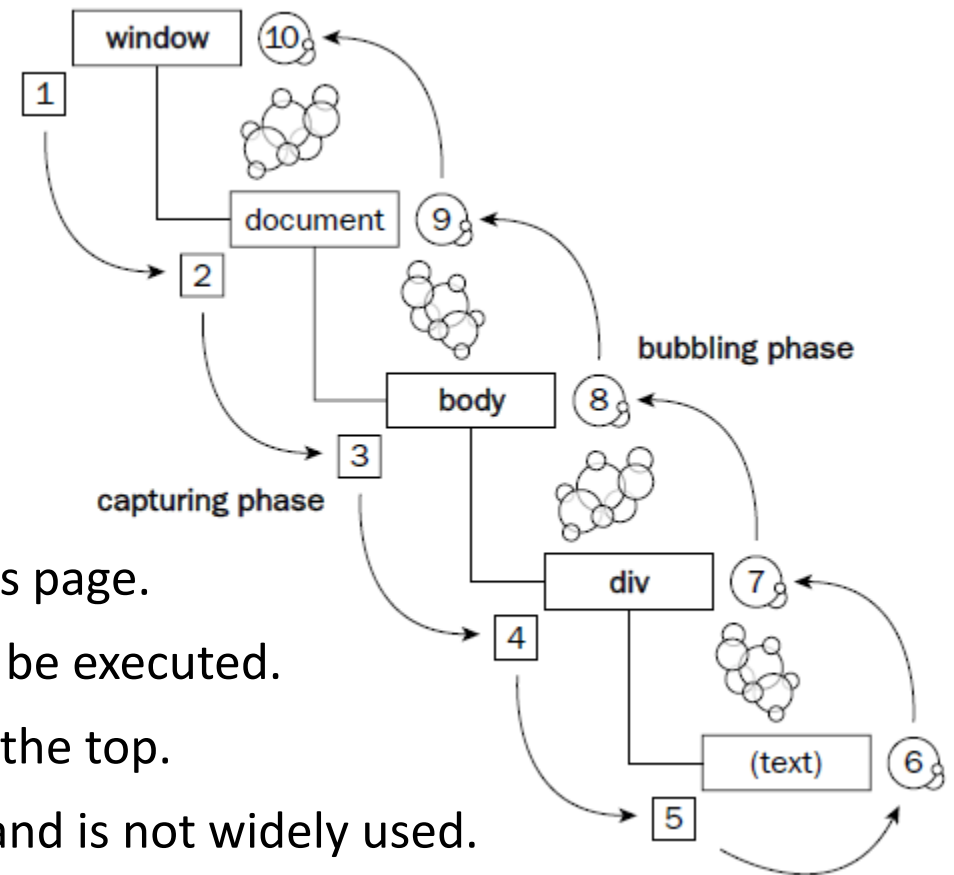
What happens when I click on the em? Which element will get the event?

Answer: All of them!

# Bubbling vs Capturing

```
<body>  
  <div>  
    <p> Events are <em>crazy</em>! </p>  
  </div>  
</body>
```

- Clicking the em is actually a click on every element in this page.
- Therefore, it was decided that all of the handlers should be executed.
- The events **bubble** from the bottom of the DOM tree to the top.
- The opposite model (top to bottom) is called **capturing** and is not widely used.



# Stopping an event from bubbling

Use the `stopPropagation()` method of the jQuery event to stop it from bubbling up.

```
<body>
  <div>
    <p> Events are <em>crazy</em>! </p>
  </div>
</body>
<script>
  document.querySelectorAll("body, div, p, em").forEach(e => e.onclick = hello);
  function hello(evt) {
    alert("You clicked on the " + this.nodeName);
    evt.stopPropagation();
  }
</script>
```

# Multiple handlers

```
<body>
  <div>
    <p> Events are <em>crazy</em>! </p>
  </div>
  <p>Another paragraph!</p>
</body>
<script>

  document.querySelectorAll("body, div, p, em").forEach(e => e.onclick = hello);
  document.querySelector("div > p").addEventListener("click", anotherHandler);

  function hello() { alert("You clicked on the " + this.nodeName); }
  function anotherHandler() { alert("You clicked on the inner P tag"); }
</script>
```

What happens when the first p tag is clicked?

# Stopping an event right now

- Use `stopImmediatePropagation()` to prevent any further handlers from being executed.
- Handlers of the same kind on the same element are otherwise executed in the order in which they were bound.

```
function hello(evt) {  
    alert("You clicked on the " + this.nodeName);  
    evt.stopImmediatePropagation();  
}
```

# Main Point

Event handlers take callback functions that are executed later when the event occurs.

**Science of Consciousness:** Callbacks are a form of memory for an action that is automatically executed when an event happens. When we act from deep levels of awareness we are more likely to activate appropriate memories and reactions (event handlers).



# JS Timers (review)

```
// arranges to call given function after given delay in ms
```

```
setTimeout(function, delayMS);
```

```
// arranges to call function repeatedly every delayMS ms
```

```
setInterval(function, delayMS);
```

Both **setTimeout** and **setInterval** return an ID representing the timer, this ID can be passed to **clearTimeout(timerID)** and **clearInterval(timerID)** to stop the given timer.

**Note:** If **function** has parameters: **setTimeout(function, delayMS, param1, param2 ..etc);**

# Callbacks and Events Queue

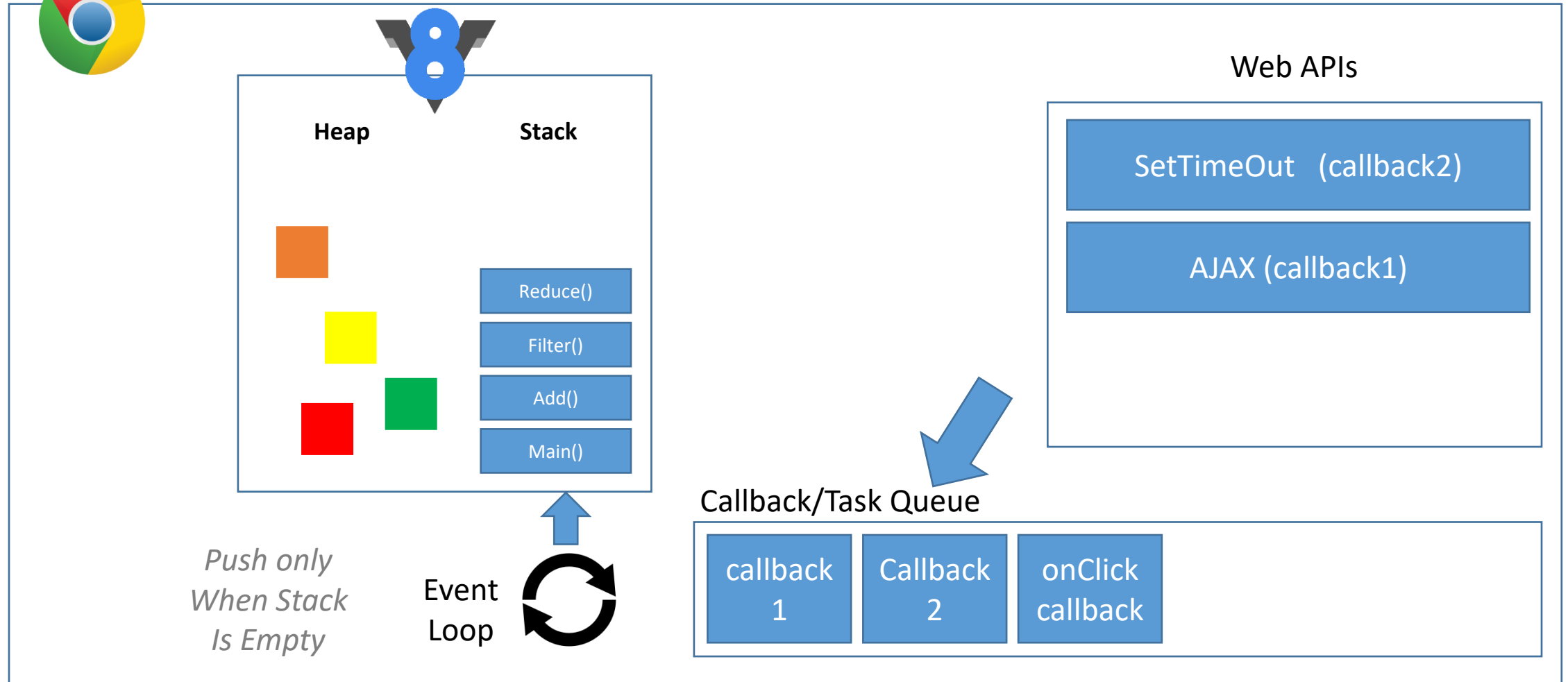
```
// In what order the results will be printed and why?  
  
function logger(f){  
  f();  
}  
  
console.log(1);  
setTimeout(function(){ console.log(2); }, 1000);  
logger(function(){console.log(3)});  
setTimeout(function(){ console.log(4); }, 0);  
console.log(5);
```

# Callbacks & Asynchronous Callbacks

A callback function is a function you give to another function, to be invoked later when the other function is finished (desired).

All Callback functions may not execute immediately, instead they may be registered in the browser and run later when desired from the browser **Event/Task Queue**.

# Chrome – Concurrency & the Event Loop



# Main Point

JavaScript is single threaded. It handles asynchronous events by storing them and cycling through them in an 'event loop'.

**Science of Consciousness:** The event loop gives the appearance of multitasking even though there is only ever a single task and thread of execution. The universe appears to be infinitely diverse even though there is only a single unified field.

# Callbacks and Pyramid of Doom

- Reading
  - [Introduction: callbacks \(javascript.info\)](http://javascript.info)

```
loadScript('1.js', function(error, script) {  
  if (error) {  
    handleError(error);  
  } else {  
    // ...  
    loadScript('2.js', function(error, script) {  
      if (error) {  
        handleError(error);  
      } else {  
        // ...  
        loadScript('3.js', function(error, script) {  
          if (error) {  
            handleError(error);  
          } else {  
            // ...  
          }  
        });  
      }  
    });  
  }  
});
```



# Promise (solution to callback hell)

- A Promise is an object representing the *eventual* completion or failure of an asynchronous operation.
- Most of the time in our application, we consume promises returned from calling some other APIs like `fetch()` or `json()`
  - But you can easily create and return Promise from your own APIs.

# Demo: Promise Constructor

```
let promise = new Promise(function (resolve, reject) {
  let r = Math.ceil(Math.random() * 10);
  if (r % 2 === 0) {
    setTimeout(function () {
      resolve('even');
    }, 300);
  } else {
    setTimeout(function () {
      reject('odd');
    }, 300);
  }
});

promise
  .then(function (value) {
    console.log("Success: "+value);
  })
  .catch(function (err) {
    console.log("Error: "+ err);
  });

console.log(promise1); // this line will execute before promise resolves or rejects
```



# Promise Constructor

- Syntax

```
new Promise(executor);
```

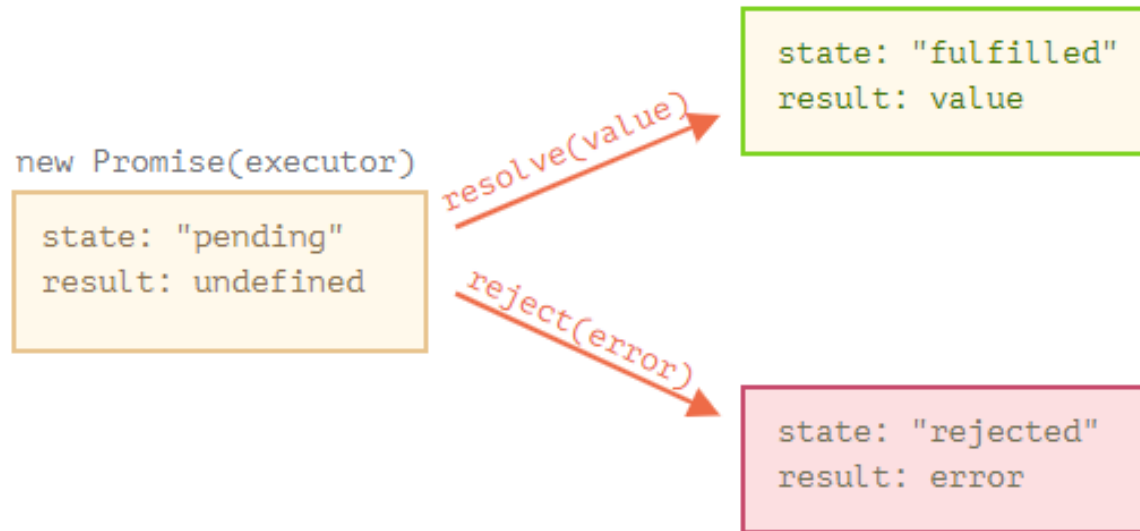
- Executer is a function that is passed with the arguments `resolve` and `reject`.
- The `executer` function is executed immediately by the `Promise` implementation, passing `resolve` and `reject` functions.
- The `resolve` and `reject` functions, when called, `resolve` or `reject` the promise, respectively.
- The `executer` normally initiates some asynchronous work, and then, once that completes, either call the `resolve` function to resolve the promise or else rejects it if an error occurred.

# Description

- A Promise is a proxy for a value not necessarily known when the promise is created.
  - It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
  - This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to *supply* the value at some point in future.

# Promise states

- A promise is in one of these states:
  - *pending* initial state, neither fulfilled nor rejected.
  - *fulfilled*: meaning that the operation completed successfully.
  - *rejected*: meaning that the operation failed.



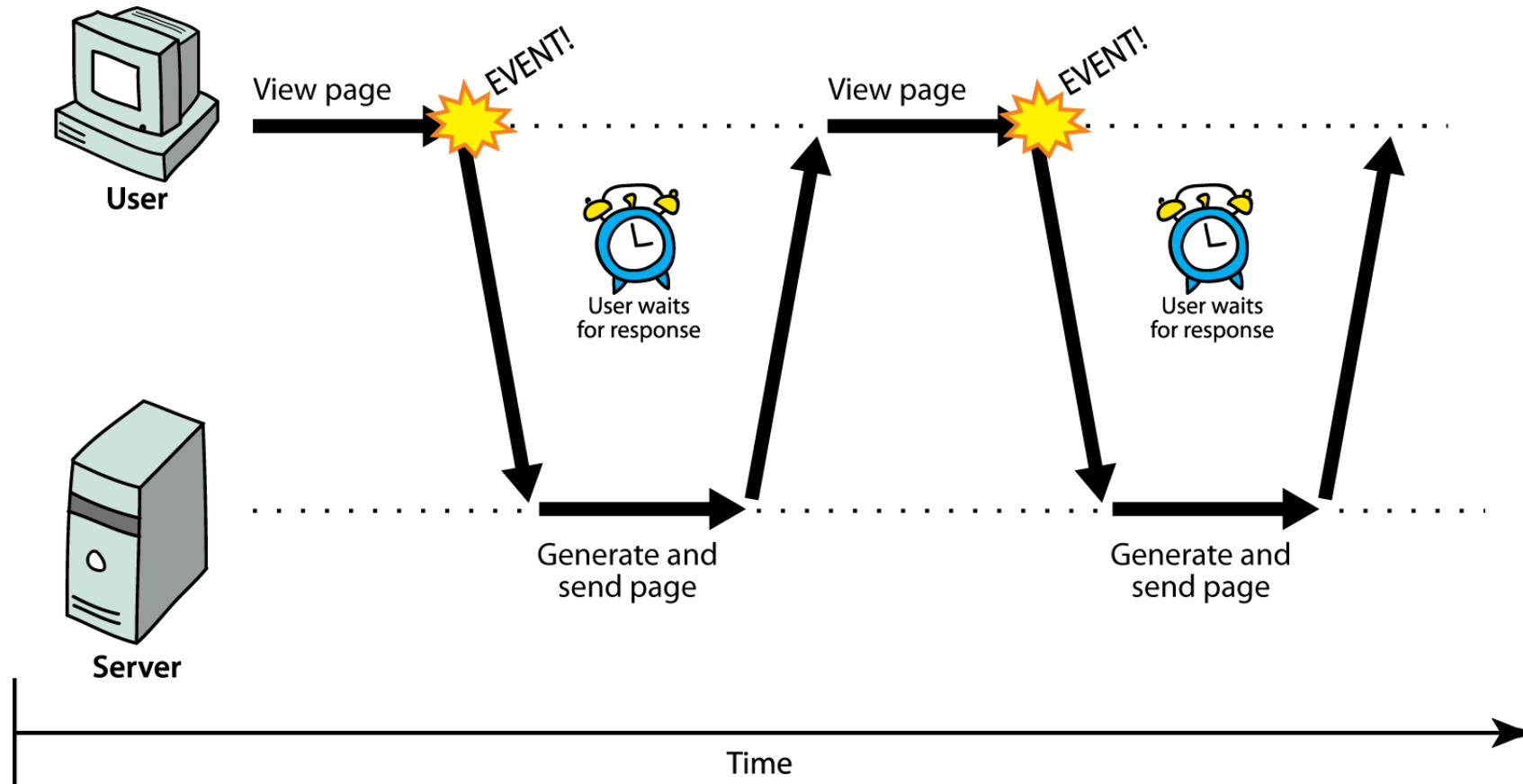
# Async/await

- There's a special syntax to work with promises in a more comfortable fashion, called "async/await".
- The keyword `async` keyword can be placed before a function
  - The word "async" before a function means one simple thing: a function always returns a promise.
  - The keyword `await` makes JavaScript wait until that promise settles and returns its result.
- `await` only works inside an `async` function.
- See example

**AJAX**

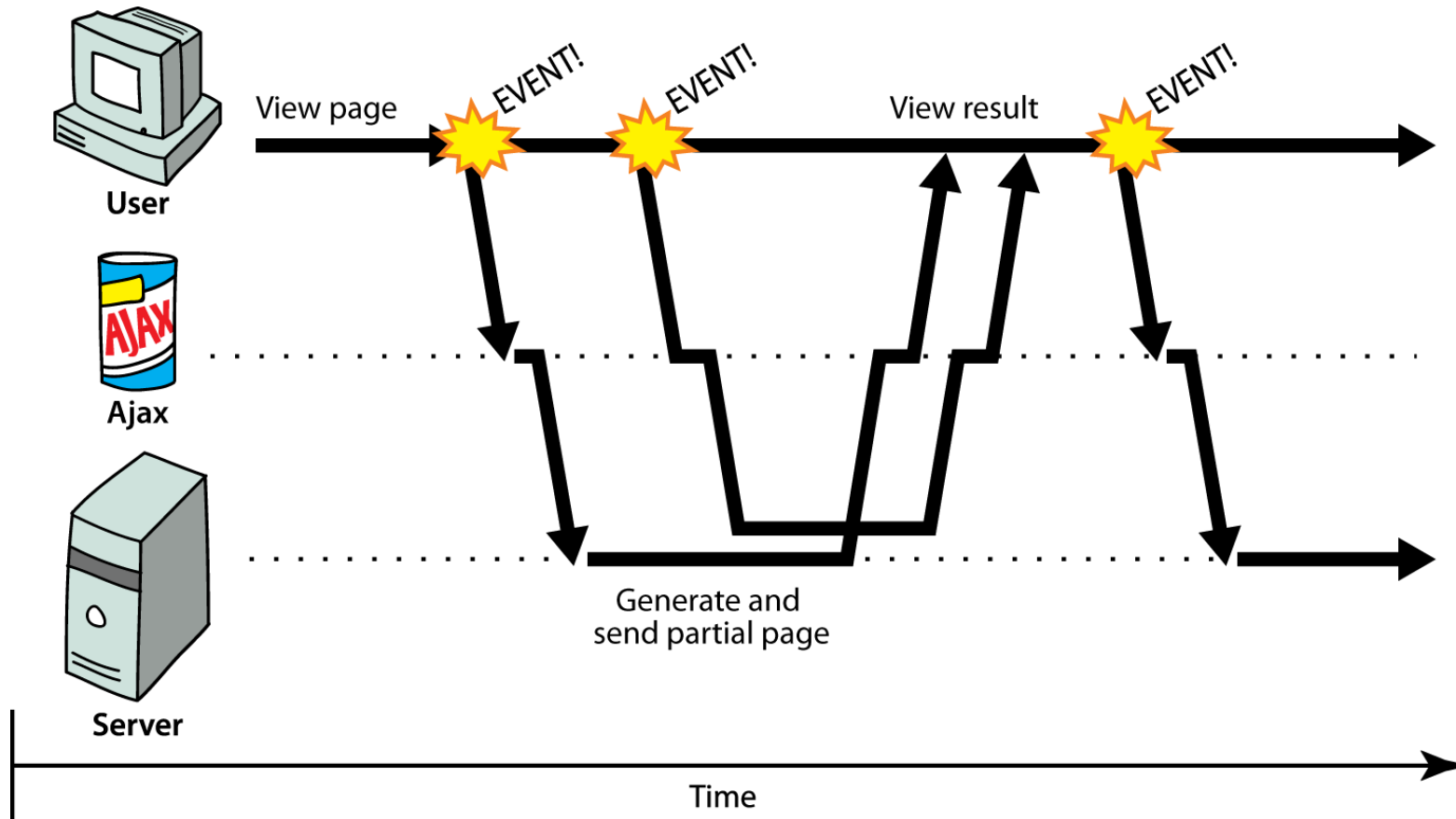
# Synchronous web communication

**Synchronous:** user must wait while new pages load (click, wait, refresh)



# Asynchronous web communication

**Asynchronous:** user can keep interacting with page while data loads



# Web applications and Ajax

- **Web Application:** a dynamic web site that mimics the feel of a desktop app
  - Presents a continuous user experience rather than disjoint pages
  - Examples: [Gmail](#), [Google Maps](#), [Google Docs and Spreadsheets](#)
- **Ajax:** Asynchronous JavaScript and XML
  - Not a programming language; a particular way of using JavaScript
  - Downloads data from a server in the background
  - Allows dynamically updating a page without making the user wait
  - Avoids the "click-wait-refresh" pattern
  - Example: [Google Suggest](#)



# XMLHttpRequest

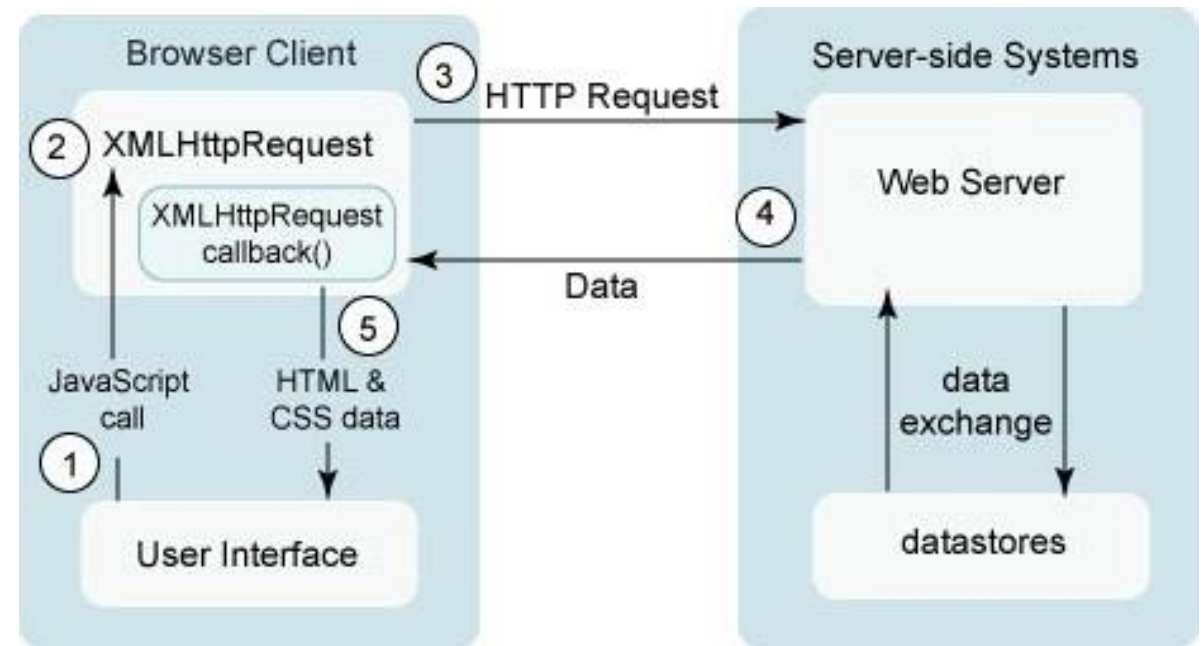
- JavaScript includes an **XMLHttpRequest** object that can make requests to a web server
- Supported in IE5+, Safari, Firefox, Opera, Chrome, etc. (with minor incompatibilities)
- It can do this asynchronously (in the background, transparent to user)
- The contents of the fetched file can be put into current web page using the DOM
- Sounds great!...

# XMLHttpRequest *(and why we won't use it)*

- New Fetch API provides a more powerful and flexible feature set.
- In modern web-development `XMLHttpRequest` is used for three reasons:
  - Historical reasons: we need to support existing scripts with `XMLHttpRequest`.
  - We need to support old browsers, and don't want polyfills (e.g., to keep scripts tiny).
  - We need something that fetch can't do yet, e.g., to track upload progress.

# A typical Ajax requests

1. User clicks, invoking an event handler
2. Handler's code creates an XMLHttpRequest object
3. XMLHttpRequest object makes partial request to the server and wait.
4. Server responds (success or fail)
5. XMLHttpRequest fires callback associated with the event.
6. Your callback (event handler function) processes the data or error and partially refreshes the view.



# Main Point

The key component that a browser provides to enable Ajax is the XMLHttpRequest object, which is supported by all modern browsers. This object opens a connection with a server, sends a message, waits for the response, and then activates a given callback method. **Science of Consciousness:** The TM Technique is supported by any human nervous system. It allows us to connect with the source of thought, experience restful alertness, and then return to activity with that influence of calm alertness.

# JavaScript Object Notation (JSON)

**JavaScript Object Notation (JSON):** Data format that represents data as a set of JavaScript objects

- Natively supported by all modern browsers (and libraries to support it in old ones)
- Becoming more popular than XML due to its simplicity and ease of use

# XML vs JSON

```
<?xml version="1.0" encoding="UTF-8"?>
<note private="true">
  <from>Alice Smith (alice@example.com)</from>
  <to>Robert Jones (roberto@example.com)</to>
  <to>Charles Dodd (cdodd@example.com)</to>
  <subject>Tomorrow's "Birthday Bash" event!</subject>
  <message language="english">
    Hey guys, don't forget to call me this weekend!
  </message>
</note>
```

```
{
  "private": "true",
  "from": "Alice Smith (alice@example.com)",
  "to": [
    "Robert Jones (roberto@example.com)",
    "Charles Dodd (cdodd@example.com)"
  ],
  "subject": "Tomorrow's \"Birthday Bash\" event!",
  "message": {
    "language": "english",
    "text": "Hey guys, don't forget to call me this weekend!"
  }
}
```

# JavaScript Object Notation (JSON)

JSON is a syntax for storing and exchanging data and an efficient alternative to XML

```
{  
  "employees": [  
    {  
      "firstName": "John", "lastName": "Doe"},  
    {  
      "firstName": "Anna", "lastName": "Smith"},  
    {  
      "firstName": "Peter", "lastName": "Jones"}  
  ]  
}
```

A name/value pair consists of a field name (**in double quotes**), followed by a colon, followed by a value.

JSON values can be:

- A number (integer or floating point)
- A string (in double quotes)
- A Boolean (true or false)
- An array (in square brackets)
- An object (in curly braces)
- null

# Browser JSON methods

- You can use Ajax to fetch data that is in JSON format
- Then call `JSON.parse` on it to convert it into an object
- Then interact with that object as you would with any other JavaScript object

method	description
<code>JSON.parse(<i>string</i>)</code>	converts the given string of JSON data into an equivalent JavaScript object and returns it
<code>JSON.stringify(<i>object</i>)</code>	converts the given object into a string of JSON data (the opposite of <code>JSON.parse</code> )



# JSON expressions exercise

Given the JSON data at right, what expressions would produce:

- The window's title?  
`var title = data.window.title;`
- The image's third coordinate?  
`var coord = data.image.coords[2];`
- The number of messages?  
`var len = data.messages.length;`
- The y-offset of the last message?  
`var y = data.messages[len - 1].offset[1];`

```
var jsonString = '{
  "window": {
    "title": "Sample Widget",
    "width": 500,
    "height": 500
  },
  "image": {
    "src": "images/logo.png",
    "coords": [250, 150, 350, 400],
    "alignment": "center"
  },
  "messages": [
    {"text": "Save", "offset": [10, 30]},
    {"text": "Help", "offset": [ 0, 50]},
    {"text": "Quit", "offset": [30, 10]},
  ],
  "debug": "true"
}';
var data = JSON.parse(jsonString);
```

# Main Point

JSON has become more widely used for Ajax data representations than XML because JSON is easier to write and read and is almost identical to JavaScript object literal syntax. **Science of Consciousness:** We always prefer to do less and accomplish more. Actions arising from deep levels of consciousness are more efficient and effective.

# Fetch

- The basic syntax is:

```
let promise = fetch(url, [options])
```

- `url` – the URL to access
  - `options` – optional parameters: method, headers etc.
- Without `options`, this is a simple GET request, downloading the contents of the url.

# Fetch example

```
let response = await fetch(url);

if (response.ok) { // if HTTP-status is 200-299
  // get the response body (the method explained below)
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

- First, the promise, is returned by fetch, resolves as soon as the server responds with headers.
- Second, to get the response body, we need to use an additional method call.
  - **Response** provides multiple promise-based methods to access the body in various formats: text, **JSON**, FormData etc.

# Response headers

- The response headers are available in a Map-like headers object in `response.headers`.

```
let response = await fetch('url');

// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// iterate over all headers
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

# Request headers

- To set a request header in `fetch`, we can use the `headers` option. It has an object with outgoing headers, like this:

```
let response = fetch(protectedUrl, {  
  headers: {  
    'Content-Type': 'application/json;charset=utf-8'  
  }  
});
```

# POST requests

- To make a **POST** request, or a request with another method, we need to use fetch options:
  - **method** – HTTP-method, e.g., **POST**,
  - **body** – the request body

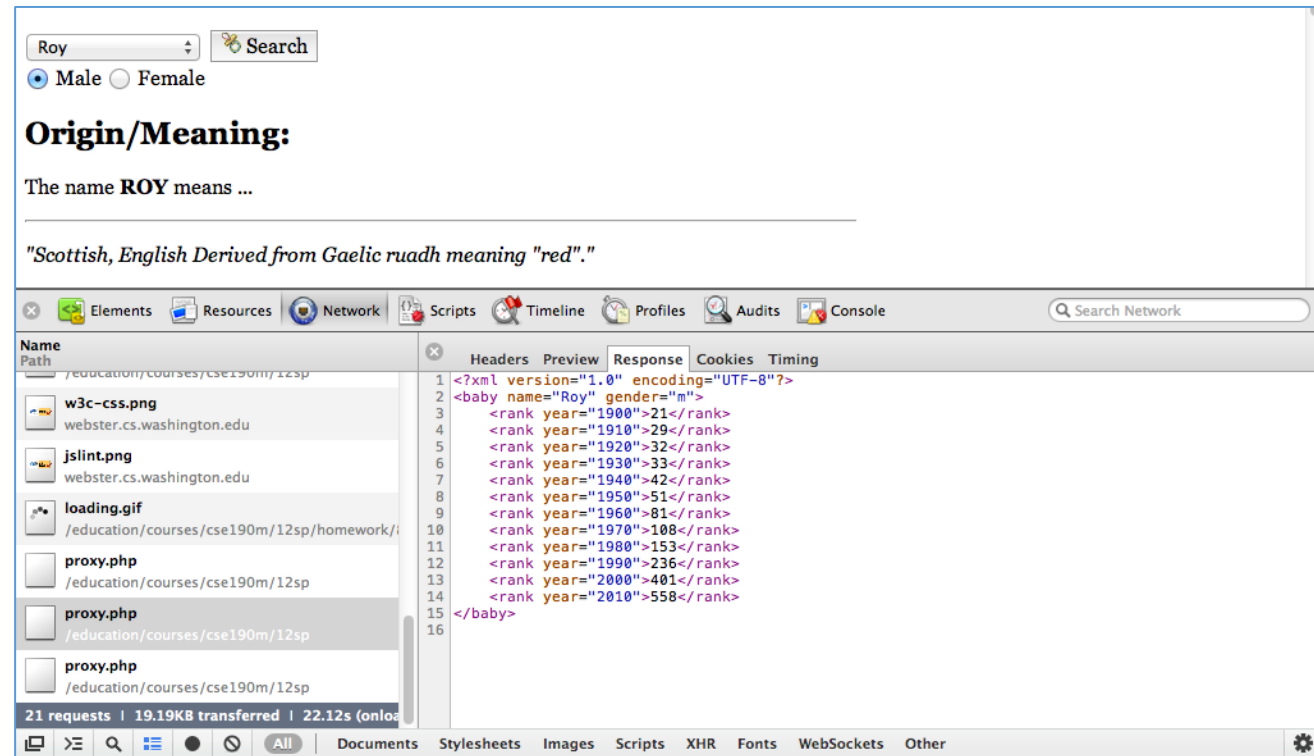
```
let user = { name: 'John', surname: 'Smith' };

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

# Debugging AJAX code

- Chrome DevTool's **Network** tab shows each request, parameters, response, errors
- expand a request by clicking on it and look at **Response** tab to see Ajax result
- check the **Console** tab for any errors that are thrown by requests





# Main Points

- The browser console window is critical to debugging Ajax code since it allows the developer to easily see the request that was sent and the response that was returned. **Science of Consciousness:** TM Checking is critical to ensure that we are practicing our TM Technique easily and effortlessly.

# XMLHttpRequest security restrictions

- The same-origin policy restricts how a document or script loaded from one origin can interact with the resource from another origin.
- It is a critical security mechanism for isolating potentially malicious documents.
  - Origin: combination of protocol, host and port



# Main Points

The same origin policy is a security constraint on browsers that restricts scripts to only contact a site with the same domain name, application protocol, and port. This means that browsers only allow Ajax calls to the same web server from which the page originated. **Science of Consciousness:** If we are calm and alert then we are more secure from disruptions by external distractions or deceptions.

# AJAX summary

- So, what makes AJAX, well... AJAX?
  - Asynchronous request.
  - Partial page rendering.

# Exercise: Parsing JSON

Suppose we have a service <http://jsonplaceholder.typicode.com> about blogs.

- Write a page that processes this JSON blog data.
  - To display user information </users/1>
  - Display all posts from selected user </posts?userId=1>
  - Display all comments from selected post </comments?postId=1>
- Create one page with input form to take userId from the browser
- Display user name and email and address and all posts belongs to this userId
- For every post, you need to show a button (show comments) once clicked you need to show all comments for the specific post.

# CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

## *Ajax and Advanced techniques*

1. Client side programming with JavaScript is useful for making web applications highly responsive.
  2. Ajax allows JavaScript to access the server in a very efficient manner using asynchronous messaging and partial page refreshing.
- 

3. **Transcendental consciousness** is the experience of the home of all the laws of nature where all information is available at every point.
4. **Impulses within the transcendental field:** Communication at this level is instantaneous and effortless.
5. **Wholeness moving within itself:** In unity consciousness daily life is experienced in terms of this frictionless and effortless flow of information.



# References

- [https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)
- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using\\_promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises)
- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)