JAVASCRIPT BASICS

Jakub Baierl & Šimon Lomič

JAVASCRIPT DEVELOPERS @ ACKEE

ackee

Paradox of JS:

Achilles' Heel of the language:

“Because JavaScript can be used without understanding, the understanding of the language is often never attained.”

SCOPE

SCOPE

```
function foo(a) {
    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

1: **foo**

2: **a**
b
bar

3: **c**

SCOPE

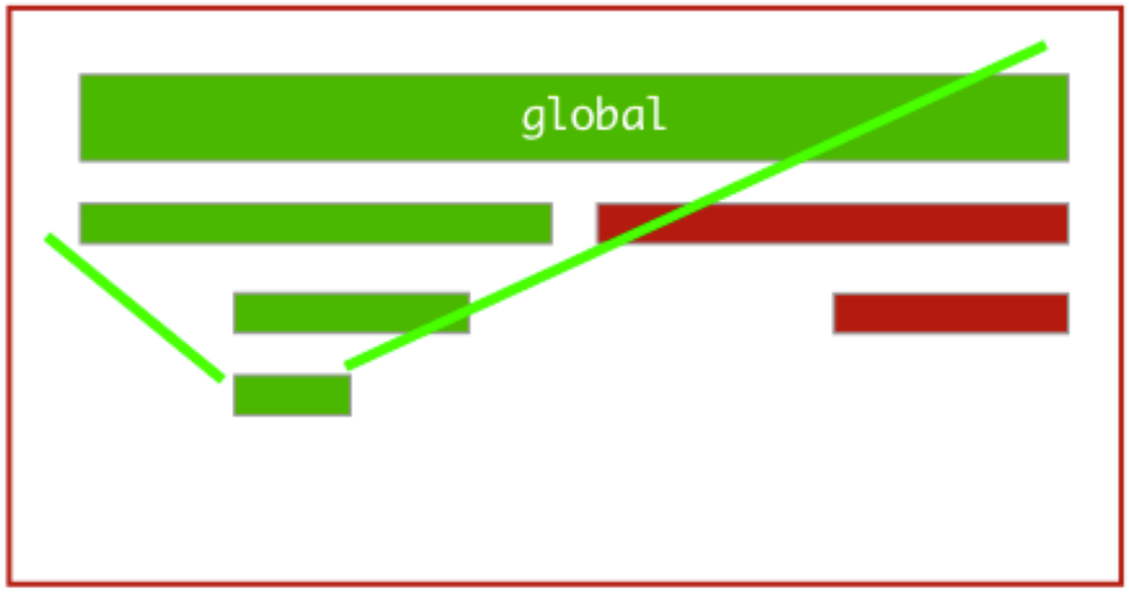
```
function foo(a) {
        var b = a * 2;

        function bar(c) {
                console.log( a, b, c );
        }

        bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

1: **foo**

2: **a**
b
bar

3: **c**

Scope visibility



SCOPE

- **Scope**
  - series of "bubbles" in which **identifiers** (variables, functions) are declared
  - collects and maintains a **look-up list** of all the declared identifiers

```
function foo(a) {
        var b = a * 2;

        function bar(c) {
                console.log( a, b, c );
        }

        bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

1:	**foo**
2:	**a** **b** **bar**
3:	**c**

- **Scope**

```
function foo(a) {
    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

1: **foo**

2: **a** **b** **bar**

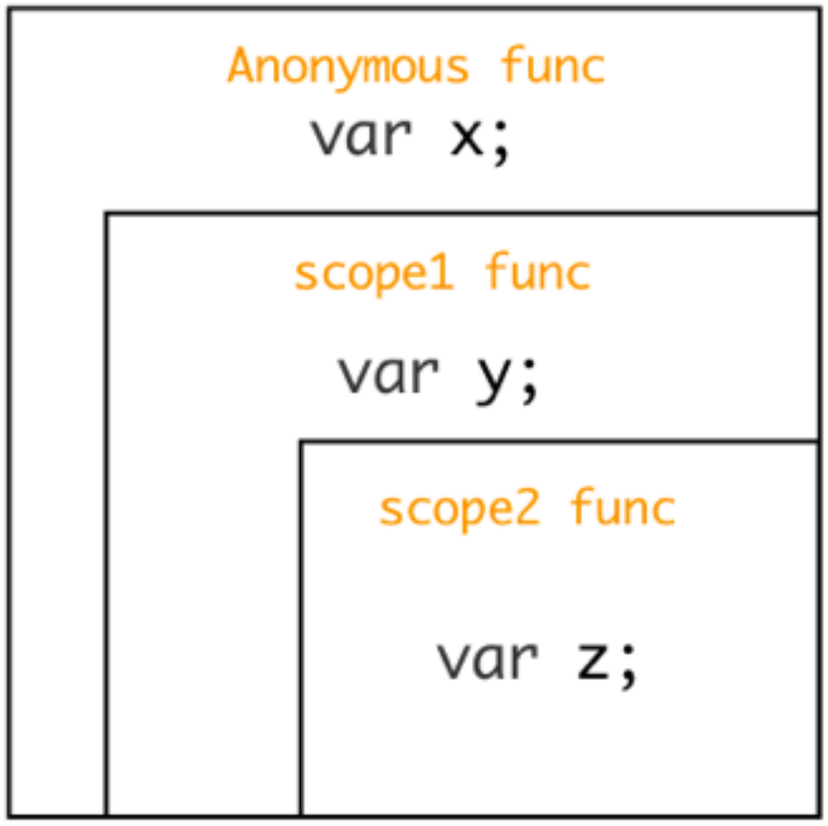
3: **c**

- identifiers are accessed using **nested scope hierarchy** from bottom to top
- when *assigning look up* (x = *1*) for identifier fails, it will create new variable at the top scope (global)

SCOPE

- **Scope hierarchy**

```
(function(){
    var x=1;
    function scope1() {
        var y=2;
        function scope2() {
            var z = 3;
            console.log(x,y,z); // 1,2,3
        }
        console.log(x,y,z); // 1,2,undefined
    }
    console.log(x,y,z); // 1,undefined,undefined
})();
console.log(x,y,z); // undefined,undefined,undefined
```



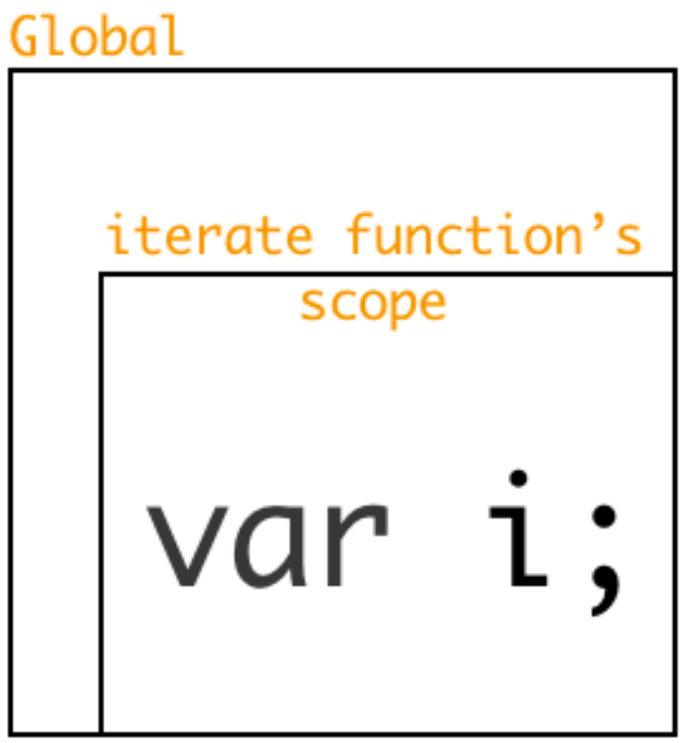
- **Identifier lookup fail on assignment operation**

```
function scoped() {
    x = 1;
}
window.x === 1 // true
```

SCOPE

- **Scope is function-based**
  - functions are the only scope creating elements *

```
function iterate() {
    for (var i=0; i<10; i++) {
        console.log( i );
    }
    for (i=0; i>-12; i--) { // no var !
        console.log( i );
    }
}
console.log(i); // fails
```



* actually not true, but *switch-case* is slow and *with* breaks things

SCOPE

- **Immediately Evoked Function Expressions (IEFE)**
  - common pattern for encapsulation of modules
  - creates scope
  - function expression (not declaration) invoked just after definition

```
var a = 2;
(function(){ // <--
    var a = 3;
    console.log( a ); // 3
})(); // <--
console.log( a ); // 2
```

- **syntax error**
  - you have to define them in **expression** context

```
> function(){
    // SCOPED
  }()
✕ ▶ Uncaught SyntaxError: Unexpected token ((…)
> (function(){
    // SCOPED
  })()
```

SCOPE

- **Why should I scope?**
  - encapsulation of identifiers
  - *Least Exposure Principle:*
    - you should expose only what is minimally necessary
  - Collision Avoidance (especially for libraries)

```
// Library 1
var x = 0;
function incr() { x++; }
 // Library 2 (separate file)
var x = 0;
function decr() { x--; }
```

->

```
var incr = {function(){
    var x;
    return function(){ x++; }
}}();
var decr = {function(){
    var x;
    return function(){ x++; }
}}();
```

SCOPE > HOISTING

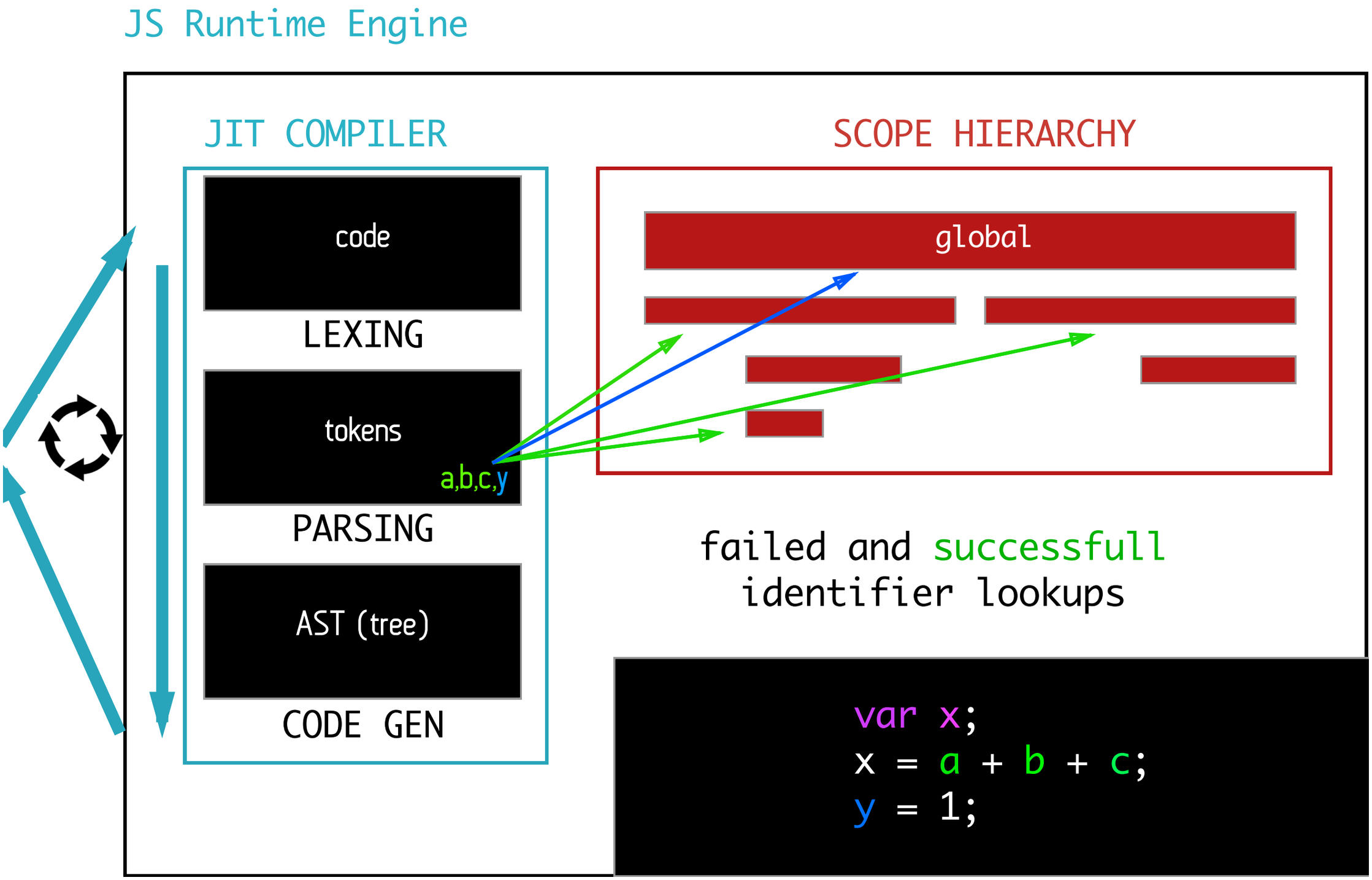
- What will this do?

```
y=2; z=3;
if(x) x();
function x() { console.log(y); }
console.log(z);
var y;
```

- expected `undefined` ?
- No, prints 2 followed by 3
  - why?

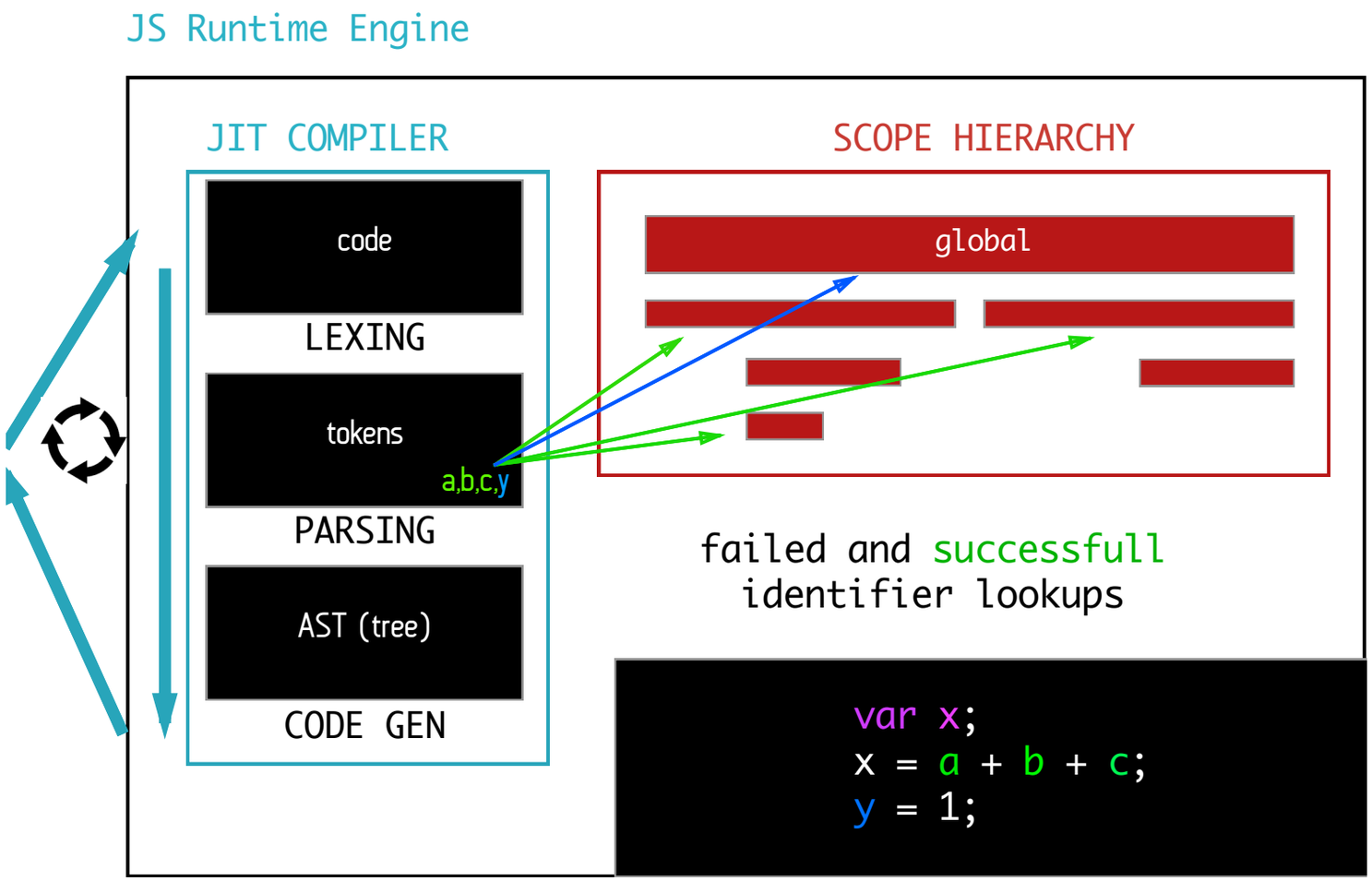
SCOPE & HOISTING

- **When does lookup for identifier happen?**



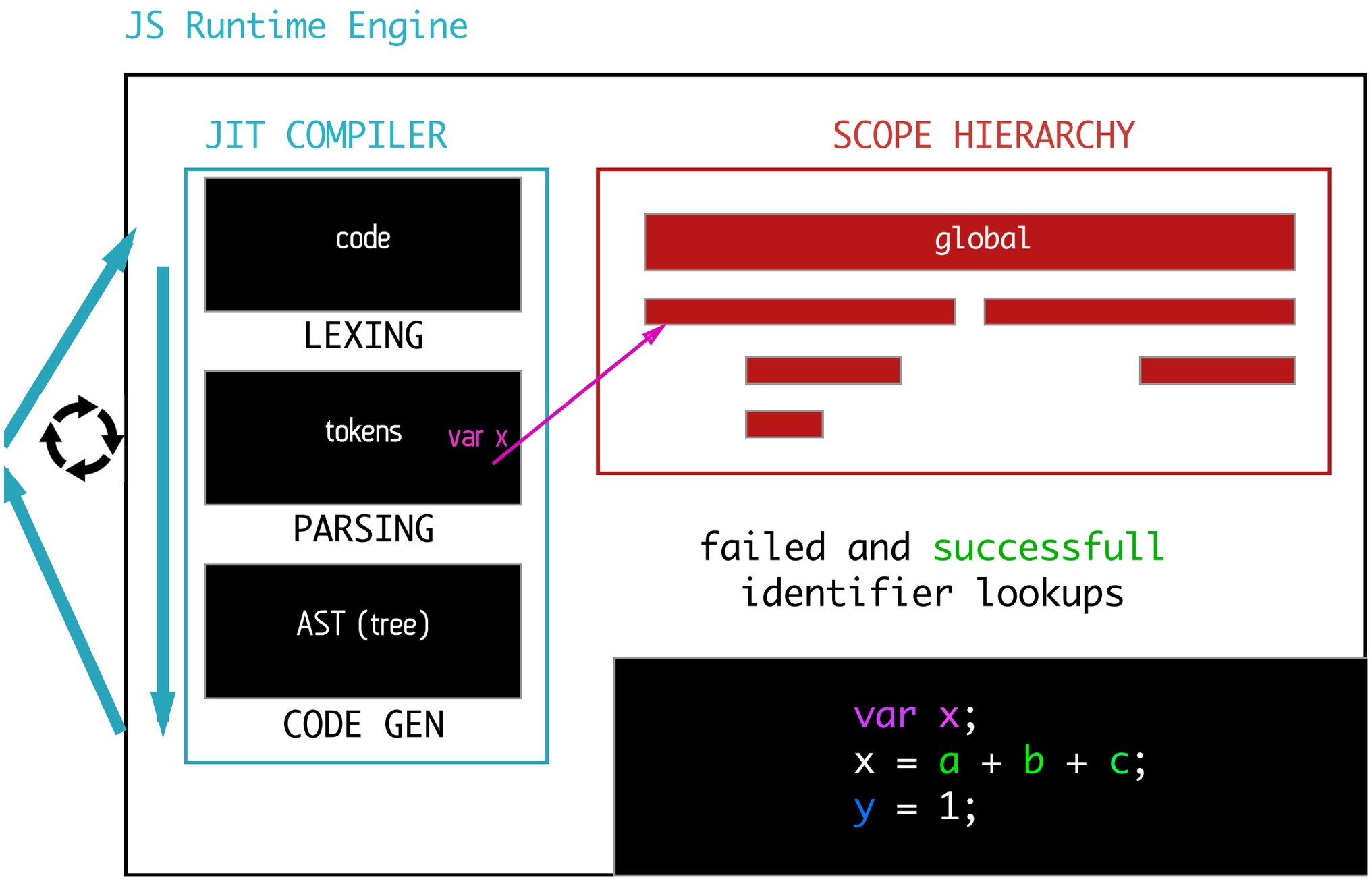
SCOPE

- **When does lookup for identifier happen?**
  - at each just in time compiling
  - for each compiled portion of code (JIT) the variables are determined **before** runtime



SCOPE > HOISTING

- **Each var keyword creates new variable on current scope** (if not already created)



SCOPE > HOISTING

- **HOISTING**
  - this creates effect of “hoisting” of declarations of variables and functions

```
y=2; z=3;
x();
function x() { console.log(y); }
console.log(z);
var y = 4;
// 2, 3
```

->

hoisting

```
function x(){console.log(y);}
var y;
y=2; z=3;
x();
console.log(z);
y = 4;
```

  - declarations are “hoisted up” (executed first) to the top of the scope
  - each time parser finds in code **var** keyword, new variable in current scope is created

SCOPE > HOISTING

- **variable vs. function declarations**

```
console.log(x);
y();
var x = 2;
function y() {console.log(3)};
```

->
hoisting

```
var x;
function y() {console.log(3)};
console.log(x);
y();
x = 2;
// undefined, 3
```

  - the value of hoisted variable **is not** hoisted
  - the implementation of hoisted function **is** hoisted

- multiple declarations are overridden

- **function declaration vs. function expression**

```
var x = function(){console.log(y);}
// vs.
function x() { console.log(y); }
```

- the function expression is **not** hoisted

```
x();
function x() { console.log(1); }
var x = function() { console.log(2); }
// 2

function x() { console.log(1); }
x();
var x = function() { console.log(2); }
// 1
```

SCOPE & CLOSURES

SCOPE > CLOSURES

- **Asynchronicity**
  - when is message released? not at the end of function call?

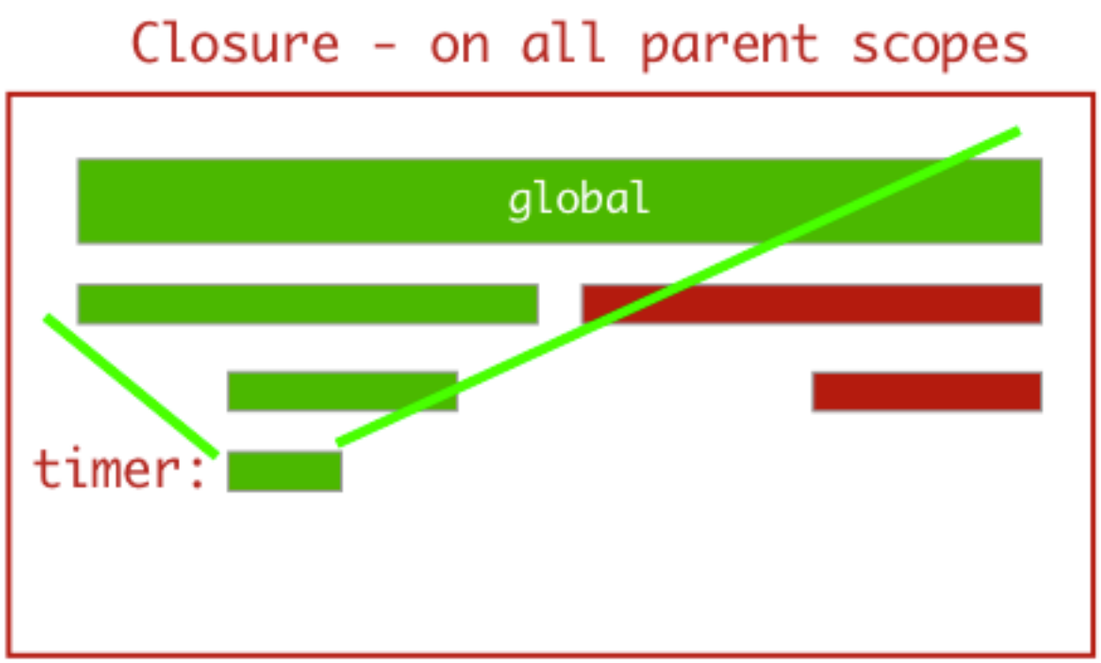
```
function wait(message) {
    setTimeout( function timer(){
        console.log( message );
    }, 1000 );
}
wait( "Hello, closure!" );
```

- **no**, because function timer still holds reference on message and timer will be deallocated after 1s
  - alloc(message) > alloc(timer) > wait 1s > call(timer) release(message) > release(timer)

SCOPE > CLOSURES

```
function wait(message) {
    setTimeout( function timer(){
        console.log( message );
    }, 1000 );
}
wait( "Hello, closure!" );
```

- we say that function timer creates closure, which “encloses” over scopes timer is nested in



SCOPE > CLOSURES

- event callback create closure as well

```
function createCounter() {
    var cnt = 0;
    $("#counter").click( function(){
        cnt++;
        console.log(cnt);
    });
}
createCounter();
```

- **Infamous for loop problem**

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i* 1000 );
}
```

- logs `6 6 6 6 6` , why?
- executed after end of cycle, **single** closure is held by reference
- I need a **closure for every iteration**
  - I create scope using IIFE

SCOPE > CLOSURES

- **Infamous for loop problem**
  - **solution:** conserving each **i** in its own scope

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i* 1000 );
}
```

->

```
for (var i=1; i<=5; i++) {
    (function(i){
        setTimeout( function timer(){
            console.log( i );
        }, i* 1000 );
    })();
}
```

* ES6 solves with **let** operator

- **Efficiency & memory management**

```
function Datacentre() {
    var BigData1 = {...}; // 1GB
    var BigData2 = {...}; // 2GB
    $("#counter").click( function(){
        console.log(BigData1);
    });
}
Datacentre();
// How much data does your app takes?
// only 1GB - BigData1
```

PROTOTYPES

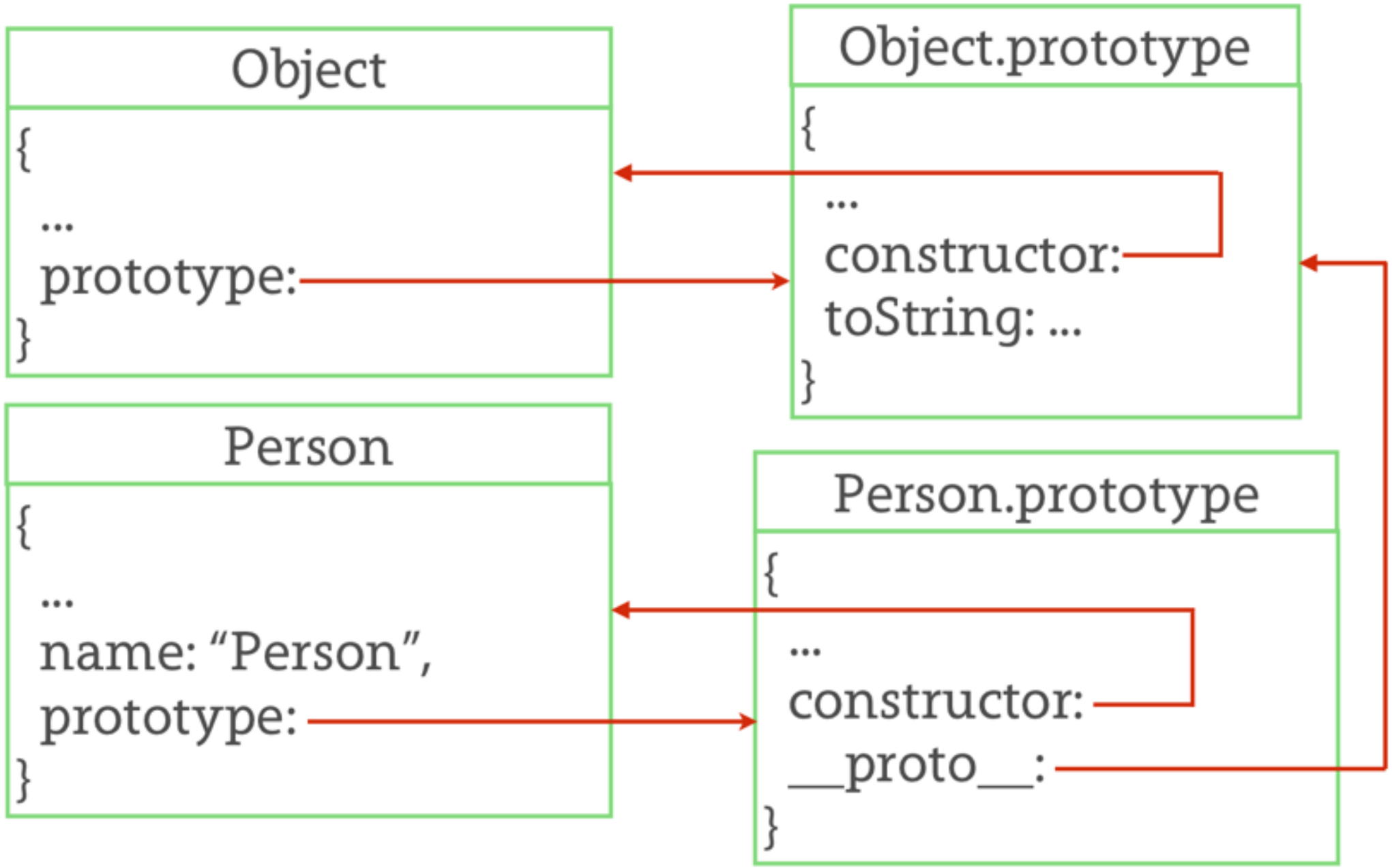
PROTOTYPES

- JS's objects are not **class-based**, but **prototype-based**
  - each object is its own entity
  - reusability of methods is done using reference to it's *prototype object*
  - this prototype chaining create tree
  - every property lookup follows the tree up to its root

PROTOTYPES

```
// CONSTRUCTOR:
function Person(name) {
    this.name = name;
}
// method implementation:
Person.prototype.getName = function() {
    return this.name;
}
// creating of instance
var bibr = new Person("Bibr");
```

PROTOTYPES


Object
{
...
prototype:
}
Object.prototype
{
...
constructor:
toString: ...
}
Person
{
...
name: "Person",
prototype:
}
Person.prototype
{
...
constructor:
__proto__:
}

PROTOTYPES

- important keywords
  - **new** - operator to create new instance from constructor
  - **prototype** - property of constructor - object that will be used as prototype of new instances
  - **constructor** - property of instance, link to constructing function
  - **__proto__** - property of instance - link to its prototype object

PROTOTYPES

- **new**
  - what happens when `new Person("Name")` is executed?

    1. New empty object is created

       ```
       var obj= {}
       ```

    2. Its __proto__ is set to constructor's prototype (it *inherits* from that)

       ```
       obj.__proto__ = {}
       ```

    3. The constructor function is called with **this** bound to **obj**

       ```
       Person.call(obj, "Name")
       ```

    4. Constructor function is set on obj

       ```
       obj.constructor = Person
       ```

    5. Obj is returned

       ```
       return obj;
       ```

OOP

OOP

- How to implement inheritance:

```
var extends = function(child, parent) {
    var F = function(){};
    F.prototype = parent.prototype;
    child.prototype = new F();
};
extends(Employee, Person);
```

OOP

- we have “almost” classes now
- so does JS have classes?
  - No - even though we can now very closely implement class pattern in JS, the operations behind this implementation differs very much from the design pattern
  - ES6 **class** keyword is just syntactic sugar

OOP

- **this** quirks

```
var obj = {
    a: 1,
    method: function(){console.log(this.a);}
}
obj.method(); // 1
var tmpMethod = obj.method;
tmpMethod(); // undefined
```

- solution:

```
tmpMethod.call(obj);
```

- **call, apply, bind**

NEW TECHNOLOGIES

Why I should use Web workers?

- Javascript is single-thread environment
- Multiple scripts cannot run at the same time
- UI events, query and process large amounts of API data, and manipulate the DOM in the same time?
- Script execution happens within a single thread

Developers simulate 'concurrency'

- setTimeout(), setInterval(), XMLHttpRequest, and event handlers
- Yes, all of these features run asynchronously

BUT...

non-blocking doesn't necessarily mean concurrency.

Asynchronous events are processed after the current executing script has yielded.

What Web worker actually is?

- The Web Workers specification defines an API for spawning background scripts in your web application
- Web Workers allow you to do things like fire up long-running scripts, but without:
    - blocking the UI
    - blocking other scripts to handle user interactions

WEB WORKERS

- Web Workers run in an isolated thread
- As a result, the code that they execute needs to be contained in a separate file

```
var worker = new Worker('task.js');
```

- If the specified file exists, the browser will spawn a new worker thread which is downloaded **asynchronously**

```
worker.postMessage(); // Start the worker.
```

How to communicate?

- Communication between a work and its parent page is done using:
    - an event model
    - and the postMessage() method

```
self.addEventListener('message', function(e) {
    self.postMessage(e.data);
}, false);
```

Workers do **NOT** have access to:

- The DOM (it's not thread-safe)
- The window object
- The document object
- The parent object

What Sub Workers are?

- Workers have the ability to spawn child workers
- Must be hosted on **same origin** as parent

What Inline Workers are?

- What if you want to create your worker script on the fly, or create a self-contained page **without** having to create separate worker files?

With **Blob()**, you can "inline" your worker in the same HTML file as your main logic

Real-time communication without?

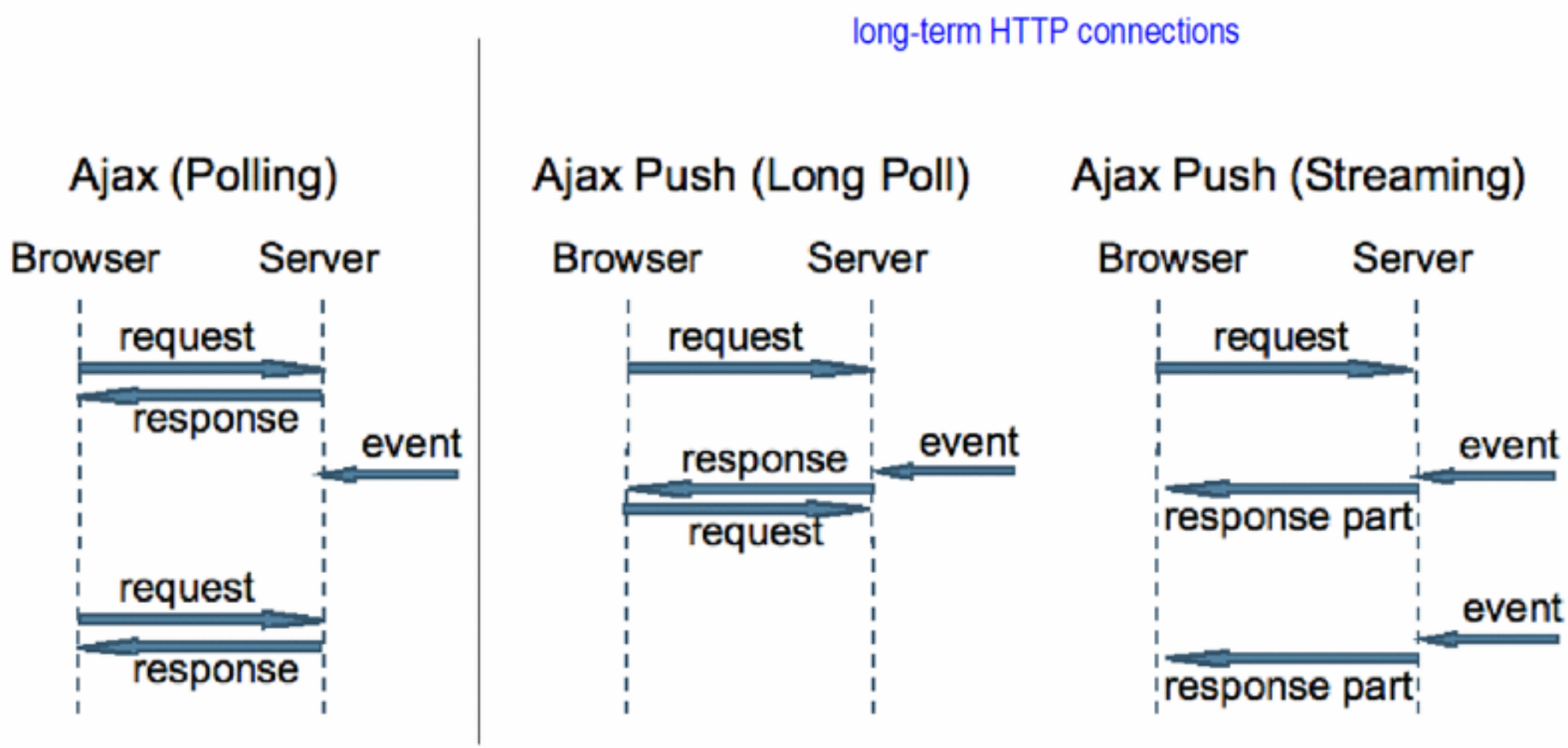
- All HTTP communication was steered by the client
    - user interaction
    - periodic **polling**

… to load new data from the server

Technologies that enable the server to send data to the client
"**Push**" or "**Comet**"

What about **long polling/streaming**?

- Client polls the server for new data
- Server holds the request open until new data is available



What Websocket actually is?

- The WebSocket specification defines an API establishing "socket" connections between a web browser and a server

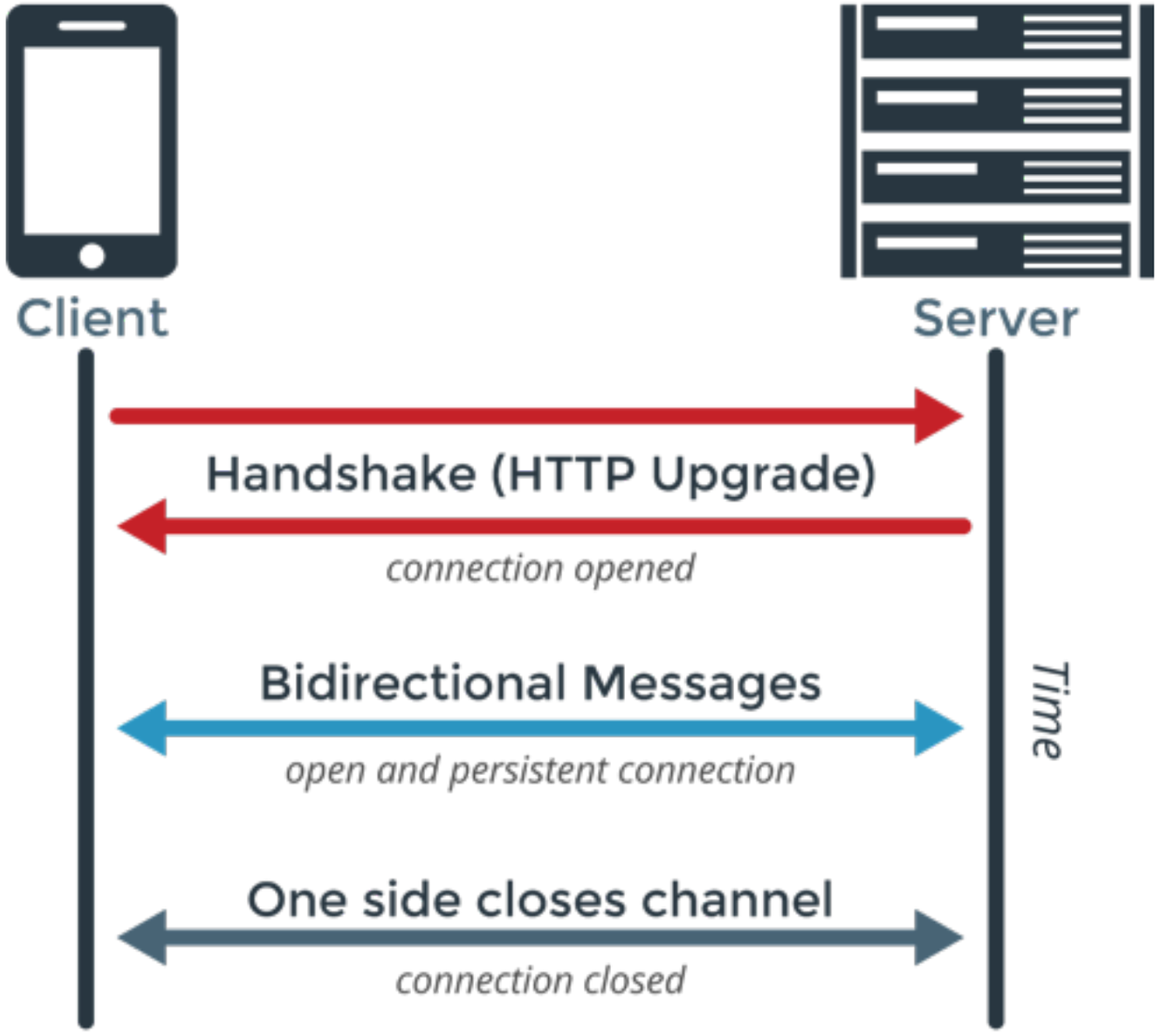
- There is an persistent connection between the client and the server and both parties can start sending data at any time.

Where should I start?

```
var connection = new WebSocket('ws://localhost', ['soap', 'xmpp']);
//opens the connection
```

WEBSOCKETS

1. handshake
2. data transfer



How the communication looks like?

- Using the **send**('your message') method on the connection object
- Message from server fires **onmessage** callback function

```
connection.onmessage = function(e) {
    console.log(e.data);
};
```

Is there some issues?

- **Proxy servers** do not like “upgrade” HTTP connection
- **Cross-origin communication**
  - Make sure only to communicate with clients and servers that you trust
  - WebSocket enables communication between parties on any domain
  - It is up to the server which domains will be allowed

HTML and 2D graphics?

YES! Use <**canvas**>!

- HTML element which can be used to draw graphics using scripting
- draw graphs, make photo composition or simple (and not so simple) animations

<CANVAS>

Get started!

```
<canvas id="tutorial" width="150" height="150"></canvas>
```

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
```

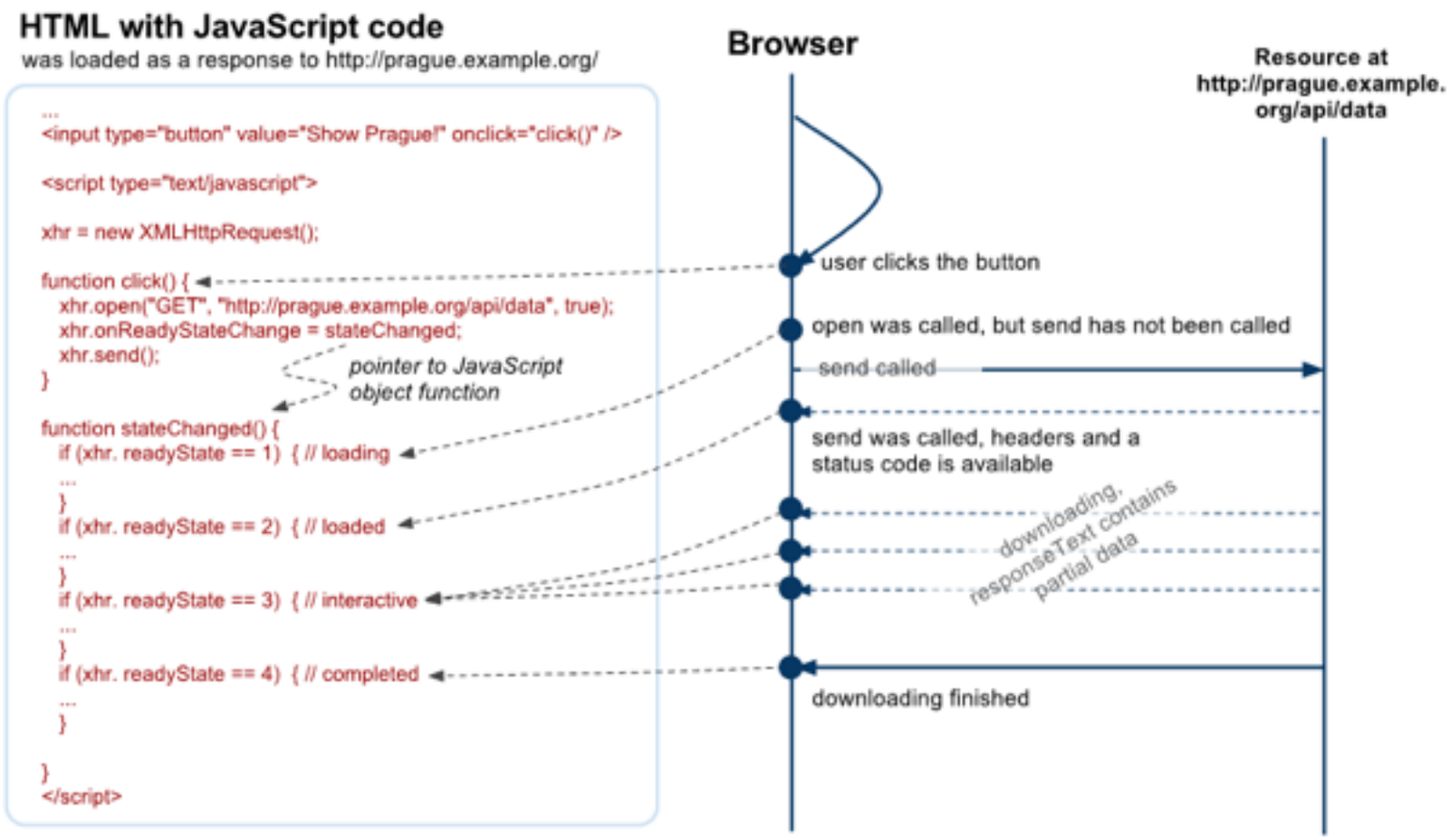
What is that **context**?

- rendering context is used to create content
- 2D to images graphs
- 3D for WebGL (based on openGL)

AJAX

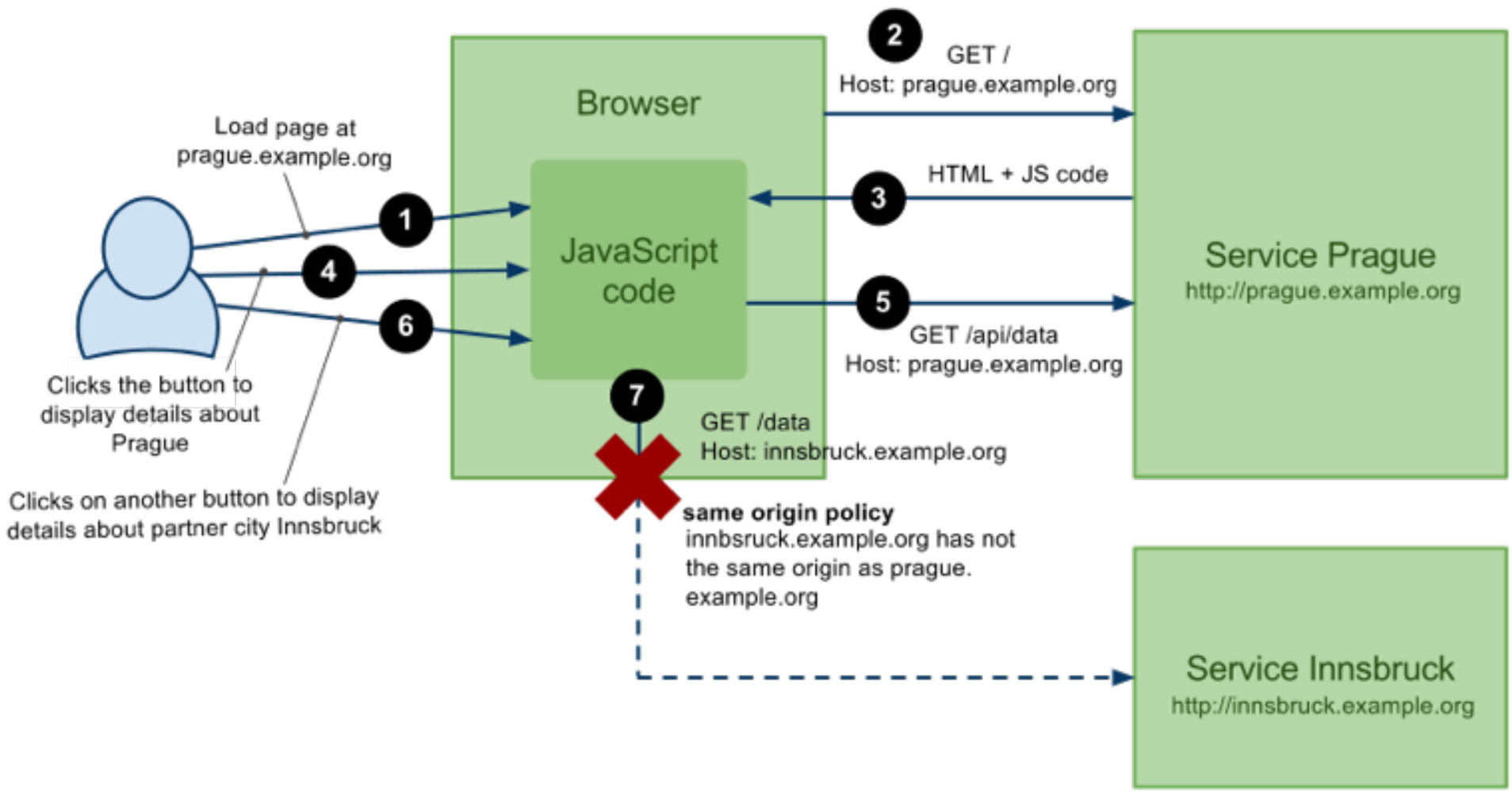
AJAX

- XMLHttpRequest
- AJAX & jQUERY
- CORS
  - client-side
  - server-side
- JSONP
- AJAX Crawling



SAME ORIGIN POLICY

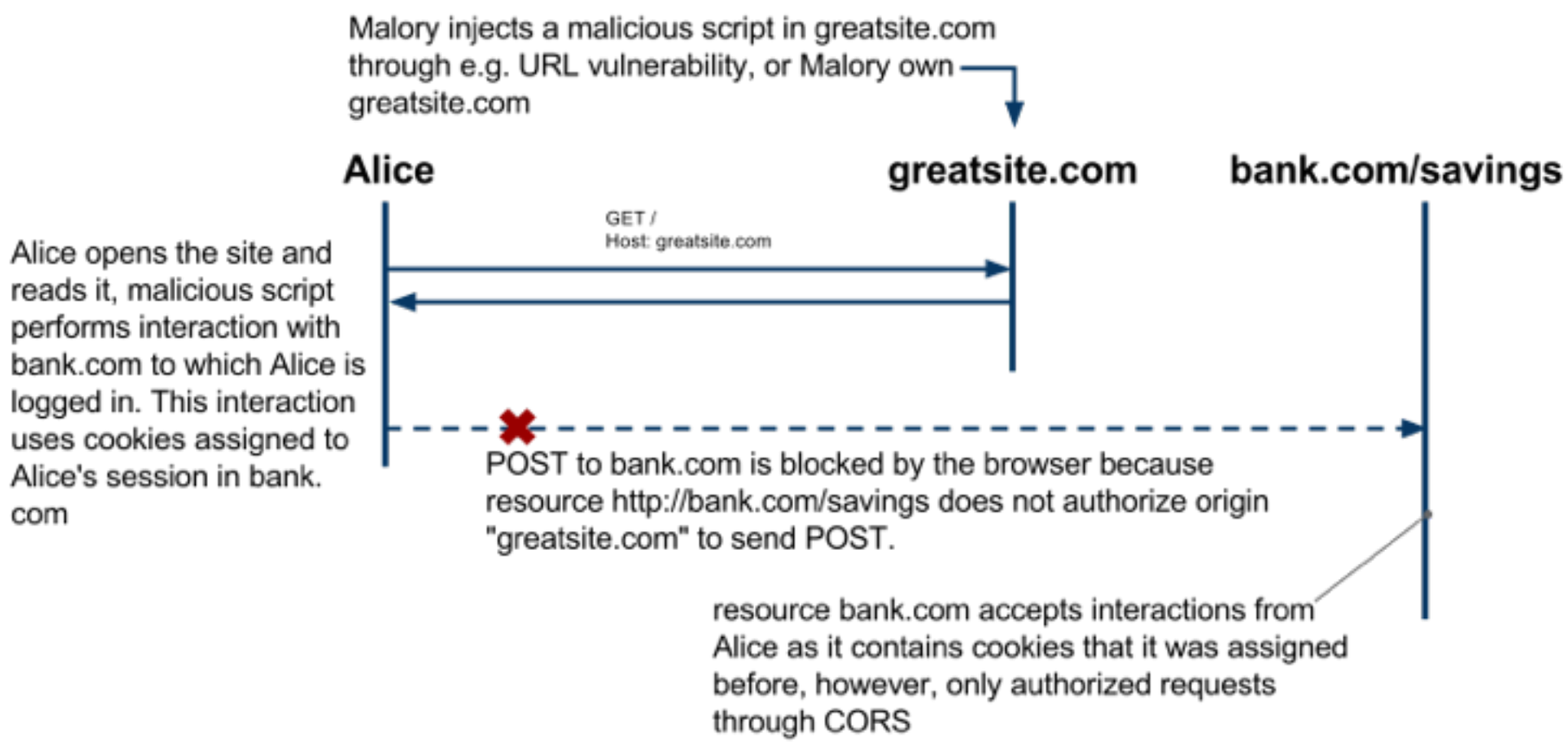
- JavaScript code can only access resources on the same domain
- Solutions: JSONP, CORS (Cross-origin Resource Sharing Protocol)



SAME ORIGIN POLICY

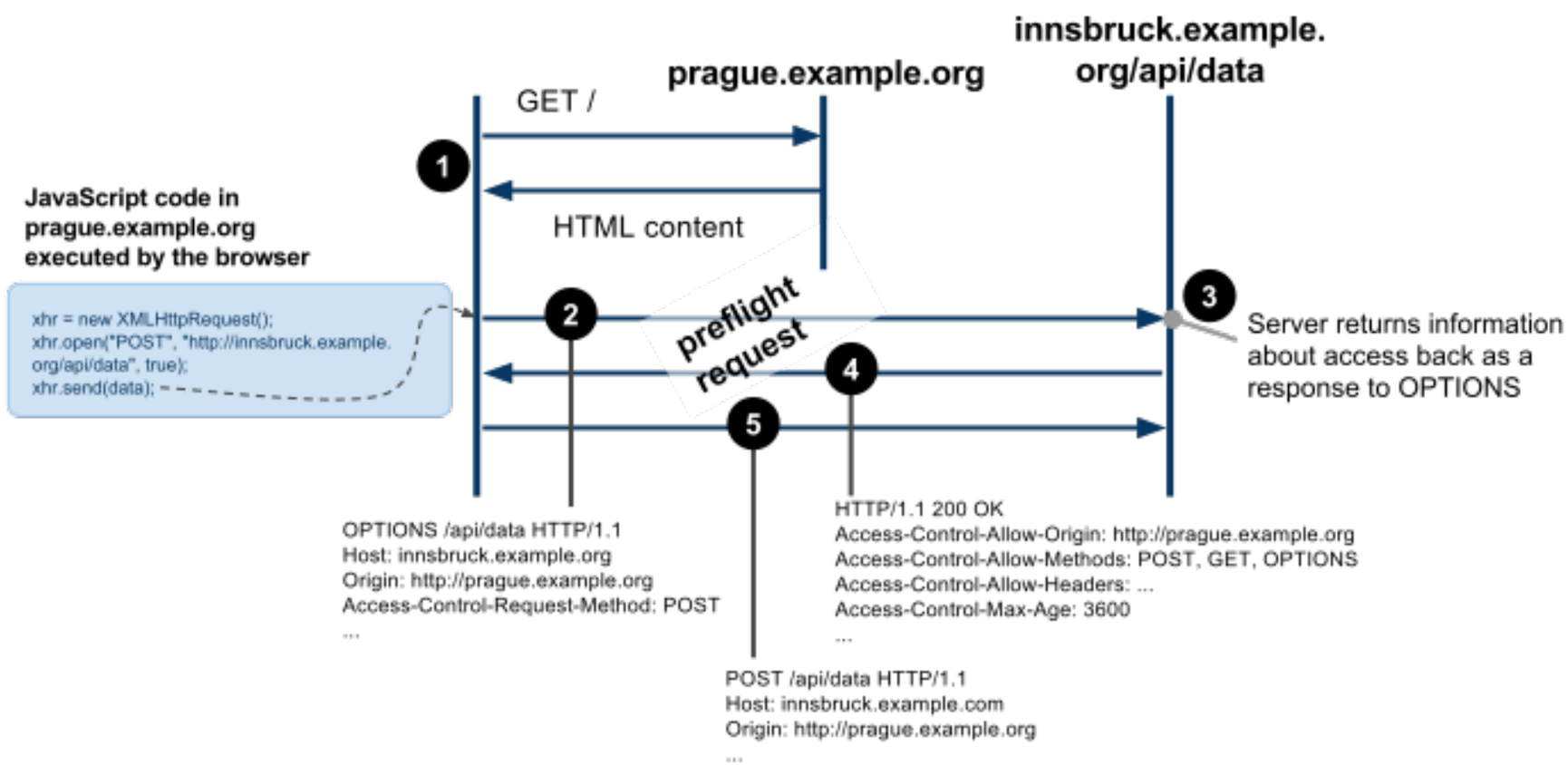
- without same origin policy is possible to do this POST

Danger !!! **Danger !!!**



CORS

- Headers:
  - Origin – identifies the origin of the request
  - Access-Control-Allow-Origin – defines who can access the resource



JAKUB BAIERL

@borecekbaji
jakub.baierl@ackee.cz