Introduction to JavaScript and Asynchronous Control Flow

Tom Van Cutsem
DistriNet KU Leuven
Comparative Programming Languages
October 2023

Outline

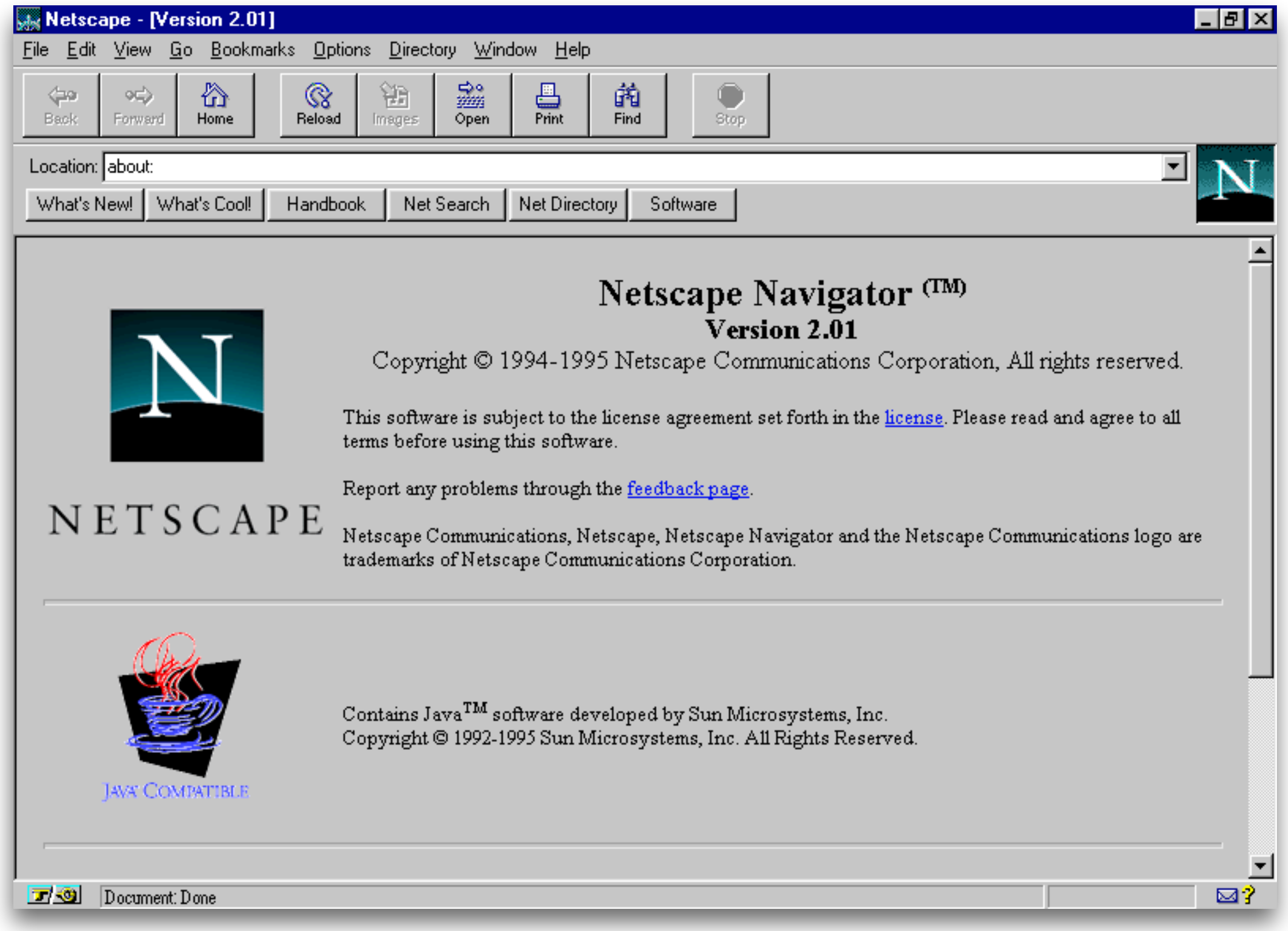
- Part 1: What is JavaScript?
- Part 2: A taste of JavaScript
- Part 3: Event loops and asynchronous control flow

Part 1: What is JavaScript?

JavaScript: origins

- Invented by Brendan Eich in 1995 at Netscape
- To support “scripting” of web pages in the Netscape Navigator browser
- First called LiveScript, then JavaScript, later standardized as ECMAScript

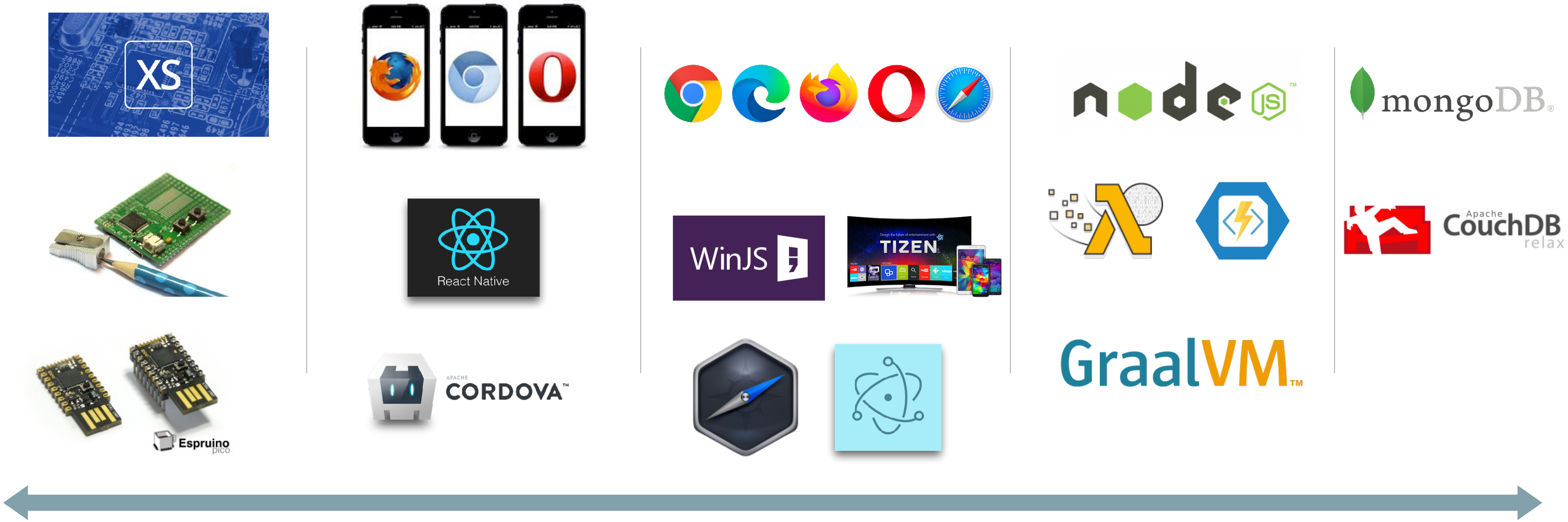
Brendan Eich



JavaScript & the Web

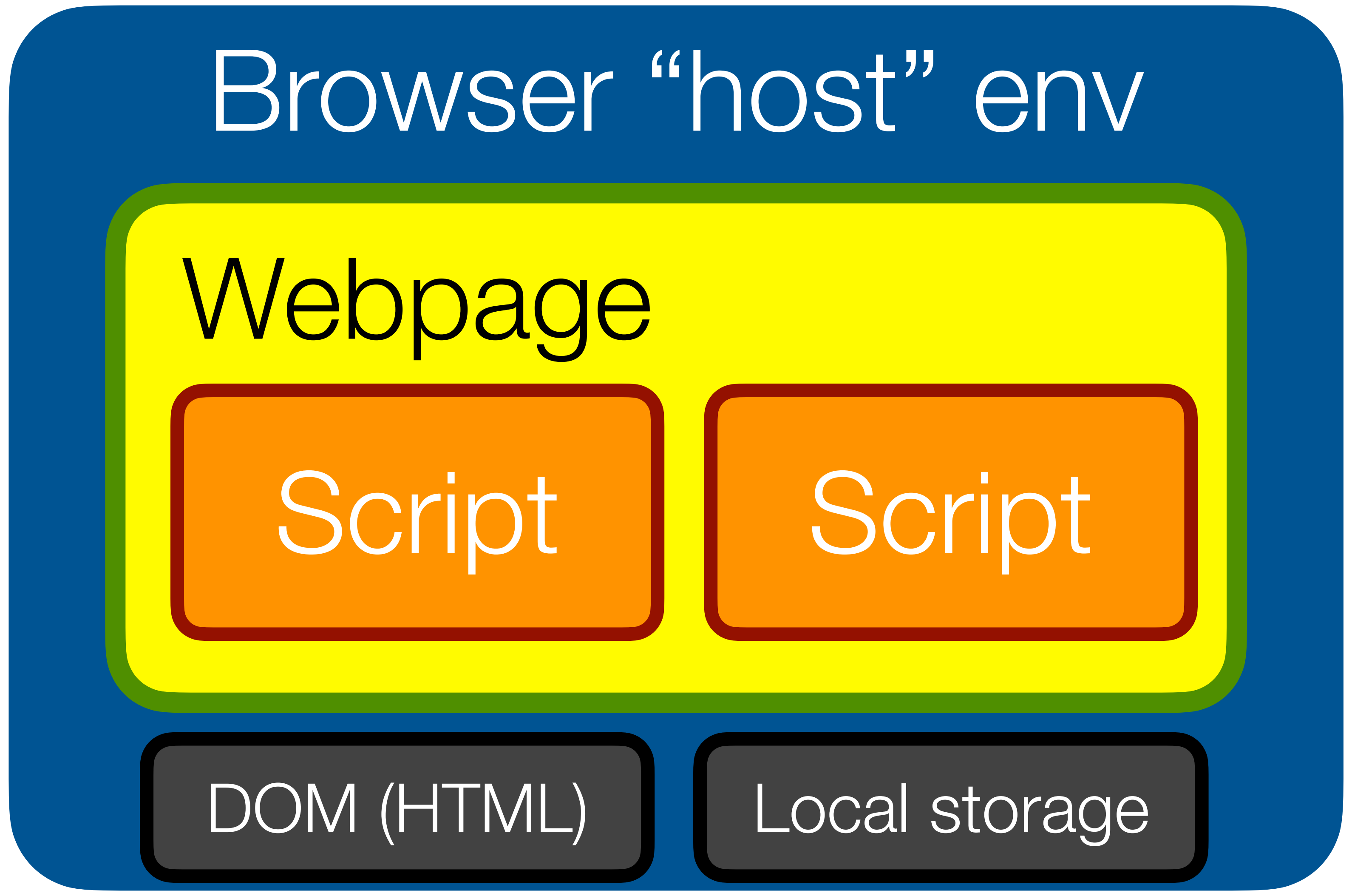
- Scripts embedded in web pages, executed on the client (in the browser)
- "Mobile" code. Remote code execution!
- Original use case: client-side form validation and UI effects

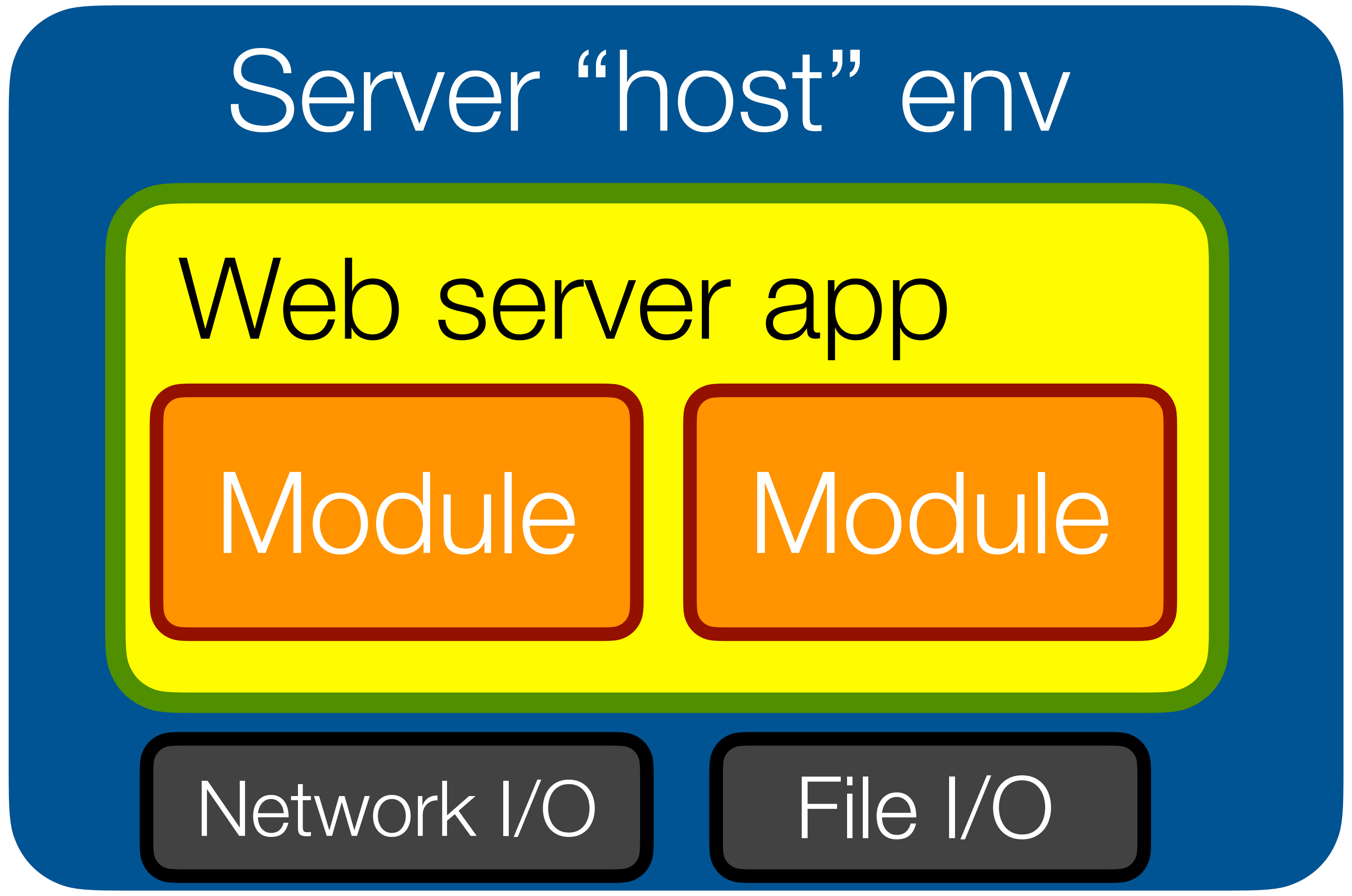
It’s no longer just about the Web. JavaScript is used widely across tiers



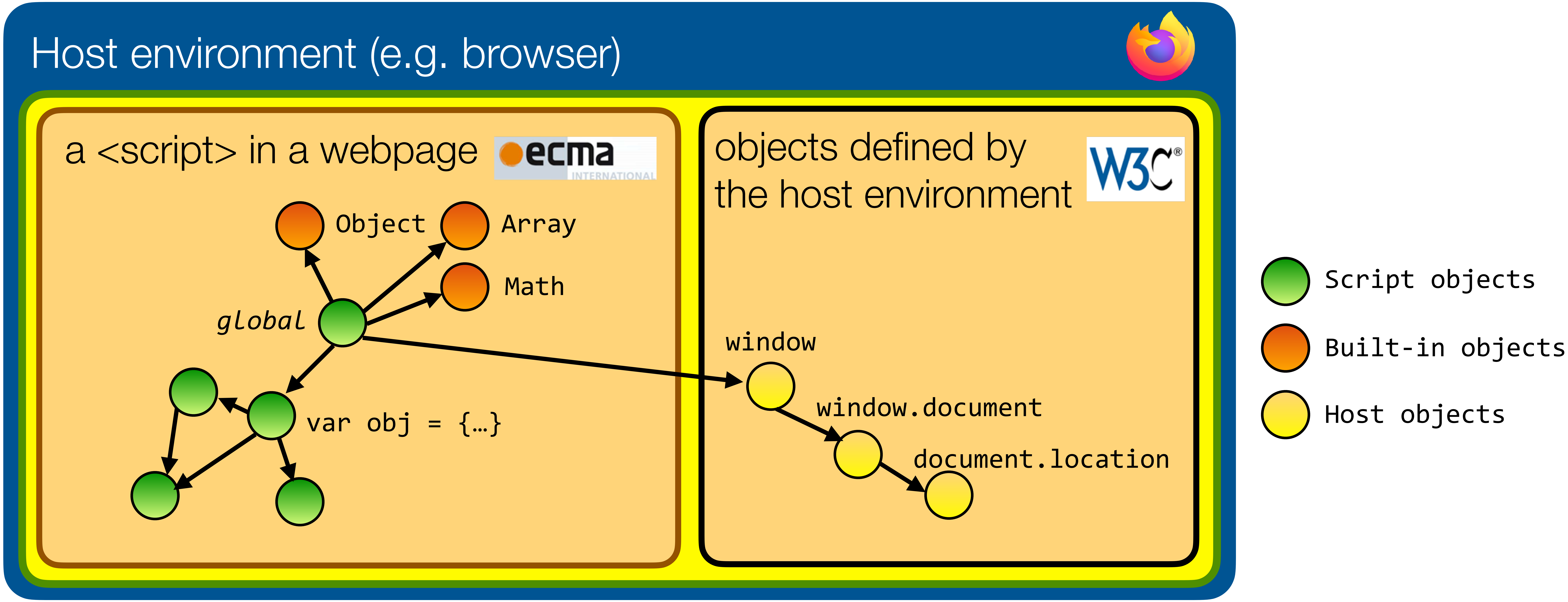
Embedded | Mobile | Desktop/Native | Server/Cloud | Database

Scripting languages are “embedded” in a “host” environment





Example: the Browser host environment



JavaScript as a language is **independent** of the host environment

- For example, on the Web:

ecma INTERNATIONAL

- Standardizes JavaScript
- Core language + relatively small **standard library**
- E.g. Object, Math, JSON, String, Date, Array, …
- Pure computation in a "virtual machine" sandbox
- Like **"User mode"** in an OS

See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

W3C®

- Standardizes browser APIs
- Large set of **system APIs**
- E.g. DOM, LocalStorage, XHR, Media, …
- Privileged access to the host environment
- Like **"Kernel mode"** in an OS

See https://developer.mozilla.org/en-US/docs/Web/API

JavaScript on the server: node.js

- Web and network application server, built on Google's V8 JavaScript runtime
- Extends Javascript with support for asynchronous I/O on files and sockets
- Example: a simple HTTP server

```
let http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

Part 2: A taste of JavaScript

Multi-paradigm: can use both object-oriented and functional styles

Object-oriented (classes & methods)

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `(${this.x} , ${this.y})`;
  }
}

let p = new Point(1,2);
p.x; // 1
p.toString(); // “(1 , 2)”
```

Functional (“records” & functions)

```
function makePoint(x, y) {
  return {
    x: x,
    y: y
  };
}
function toString(point) {
  return `(${point.x} , ${point.y})`;
}

let p = makePoint(1,2);
p.x; // 1
toString(p); // “(1 , 2)”
```

Multi-paradigm: can use both object-oriented and functional styles

Object-oriented ~~(classes & methods)~~

Objects as "records" of functions

```
function makePoint(x, y) {
  return {
    get x() { return x }
    get y() { return y }
    toString() {
      return `(${x} , ${y})`;
    }
  };
}

let p = makePoint(1,2);
p.x; // 1
p.toString(); // "(1 , 2)"
```

(See also https://martinfowler.com/bliki/FunctionAsObject.html)

Functional ("records" & functions)

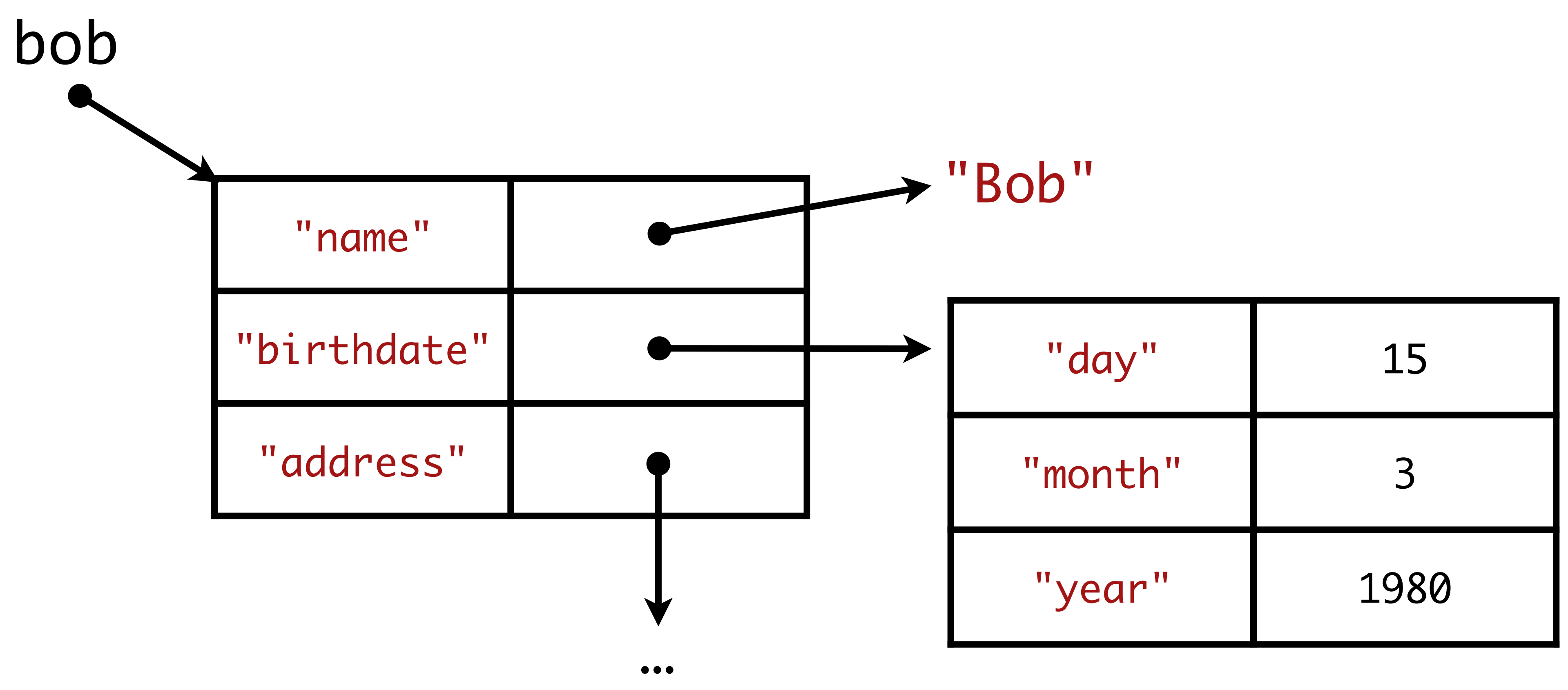
```
function makePoint(x, y) {
  return {
    x: x,
    y: y
  };
}
function toString(point) {
  return `(${point.x} , ${point.y})`;
}

let p = makePoint(1,2);
p.x; // 1
toString(p); // "(1 , 2)"
```

The three most important values in JavaScript programs

- Objects
- Arrays
- Functions

Objects

- JavaScript Objects are records that map keys (strings or “symbols”) to values
- Key-value pairs are called “properties” in JavaScript
- **Object literals** are expressions that evaluate to a fresh object, and can be arbitrarily nested
- Lookup a property using the dot-operator




```
let bob = {
  name: "Bob",
  birthdate: {
    day: 15,
    month: 3,
    year: 1980
  },
  address: {
    street: "...",
    number: 5,
    zip: 94040,
    country: "US"
  }
};

bob.address.number
// 5
```

Arrays

- JavaScript arrays are sequences of values, similar to Python or Java Lists
- Can dynamically grow/shrink to add/remove elements
- The `length` property is a computed property that returns the current number of elements
- Can access elements from index `0` up to `length-1`
- Indexing out of bounds returns the value `undefined`
- Arrays are also objects, and provide many utility methods (e.g. forEach, map, reduce, …)

```
let a = [1, "a", {x:1, y:1}]

// iterate over array, imperative style
for (let i = 0; i < a.length; i++) {
  let x = a[i];
  console.log(x);
}

// iterate over array, functional style
a.forEach(function (x) {
  console.log(x);
});

// iterate over array, using iterators
for (let x of a) {
 console.log(x);
}
```

Functions

- May be named or anonymous
- Functions are *values*
- They are “**first-class**” citizens of the language, just like objects, arrays, strings, numbers, etc.
- Must use an explicit `return` statement to return a value to the caller (otherwise, the function returns the value `undefined`)

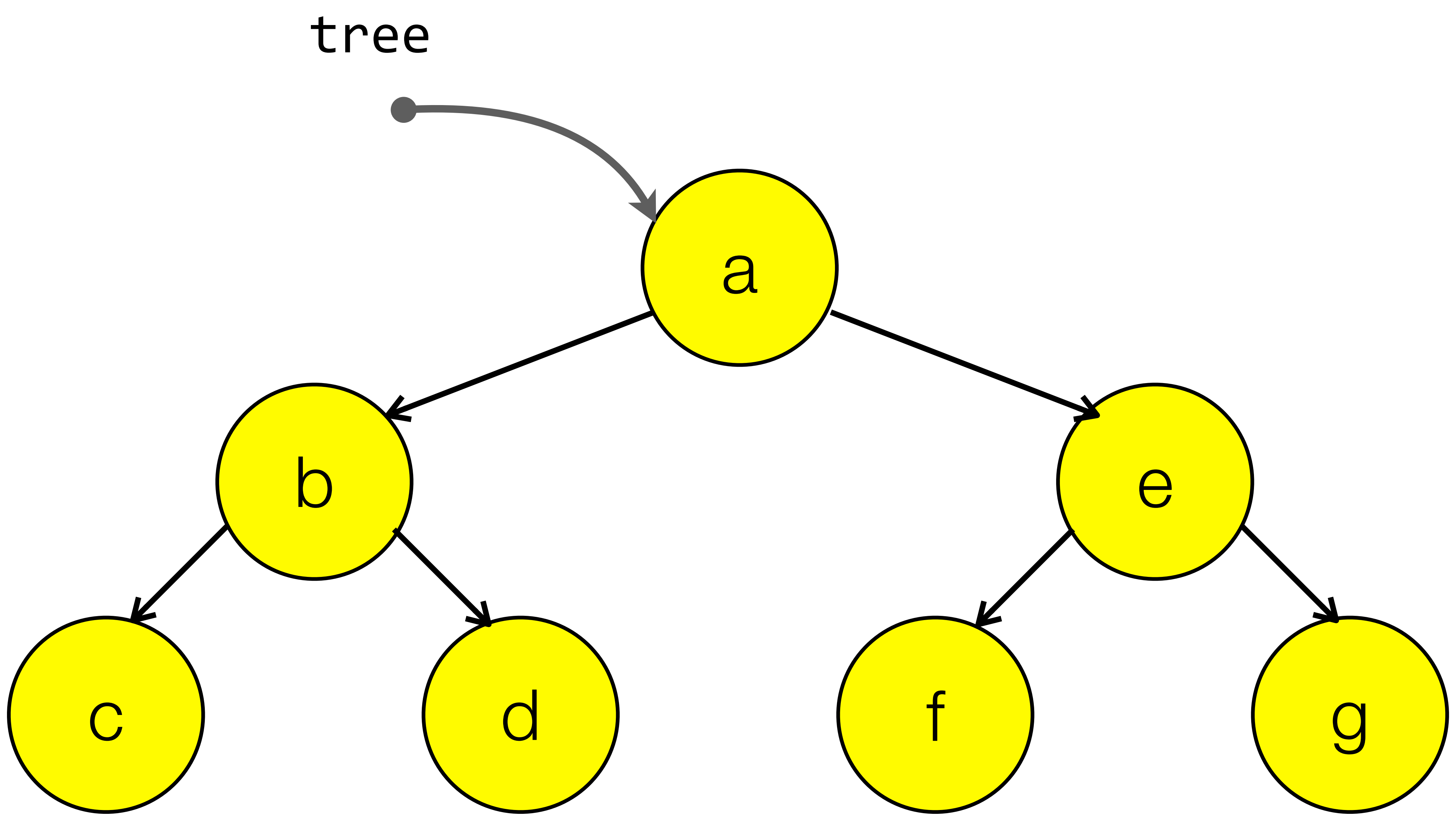
```
// a function declaration (a statement)
function add(a, b) {
  return a + b;
}

add(2, 3); // 5
```

```
// a function expression
let add = function(a, b) {
  return a + b;
}

add(2, 3); // 5
```

Algorithms 101 example: walking a binary tree

```
let tree = {
  key: "a",
  left: {
    key: "b",
    left:  { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left:  { key: "f" },
    right: { key: "g" }
  }
};
```

```
function walk(tree, keys = []) {
  if (tree) {
    keys.push(tree.key);
    walk(tree.left, keys);
    walk(tree.right, keys);
  }
  return keys;
}

walk(tree) // ["a", "b", "c", "d", "e", "f", "g"]
```



Functions

- **Higher-order** functions: functions that take other functions as input or return other functions as output
- Functions may use variables from their “outer” lexical scope (they are **closures**)

```
function makeAccumulator(init) {
  let accum = init;
  return function(val) {
    accum += val;
    return accum;
  }
}

let a = makeAccumulator(0);
a(2)    // 2
a(3)    // 5
a(0)
a(1)    // ?
```

Functions

- Higher-order functions are used everywhere in JavaScript
  - Loop over collections
  - Register event listeners
  - …

```
let a = [1, 2, 3]

a.map(function (x) { return x * x; })
// [1, 4, 9]

a.reduce(function (acc, x) { return acc + x; }, 0)
// 6
```

```
let clicks = 0;
button.addEventListener("click", function (event) {
  clicks++;
});
```

Arrow functions

- Notational shorthand
- Always anonymous
- Function body is an expression (no `return` statement needed!)
- Function body can be a statement if enclosed with `{}`

```
let a = [1, 2, 3]

a.map(x => x * x)
// [1, 4, 9]

a.reduce((acc, x) => (acc + x), 0)
// 6
```

```
let clicks = 0;
button.addEventListener("click", event => {
  clicks++;
});
```


KU LEUVEN DistriNet

Arrow functions

- Function body can be a statement if enclosed with `{}`
- Don't confuse with the syntax for object literals!
- In the first example, the `value: x` syntax is interpreted as a *labeled* statement (can be used along with `break <label>;` and `continue <label>;` statements - but this is rarely done)

```
let a = [1, 2, 3]

a.map(x => {value: x})
// [undefined, undefined, undefined]

a.map(x => ({value: x}))
// [{value: 1}, {value: 2}, {value: 3}]
```

JavaScript objects are dynamic collections of (name, value) pairs

```
let point = {x: 1, y: 2};

// can add more properties at runtime
point.z = 3;

// can delete properties at runtime (!)
delete point.z;

// computed property access
let key = input("x or y?")
point[key]

// computed property update
point[key] = 42

// can iterate over properties of an object
for (let key in point) {
  console.log(`${key} => ${point[key]}`));
}
// x => 1
// y => 2
```

```
let point = {x: 1, y: 2};

// objects can be made tamper-proof or ‘frozen’
Object.freeze(point);

point.z = 0;
// error: can’t add properties to
// a frozen object

delete point.x;
// error: can’t delete properties of
// a frozen object

point.x = 7;
// error: can’t update properties of
// a frozen object
```

JavaScript is a “dynamic” language (?)

- What do people mean by that? Unclear: no precise definition.
- JavaScript is “**interpreted**” (vs. compiled): this is a property of the *implementation*, not of the *language*. JIT and AOT JavaScript compilers exist. But it is indeed common for JavaScript code to be interpreted based directly on source files
- JavaScript is **dynamically typed**: values have a runtime type, but variables or object properties do not have a static type
- Many JavaScript operators perform **implicit type coercion**. This encourages sloppy code and invites mistakes (see examples on the right)
- The “shape” of JavaScript objects and arrays is not fixed (they support a **dynamic set of properties**, see previous slide)
- JavaScript supports “**eval**”: interpret the contents of a string as a program and execute it on-the-fly at runtime
  - Powerful and flexible, but a security nightmare if the string input can be influenced by an attacker.
  - Prefer to use modules and module loaders. Similar to dynamic class loading in e.g. the Java Virtual Machine

```
// values have a type, variables don’t
let x = 42
typeof x // “number”
x = "hello world"
typeof x // “string”
```

```
// implicit type coercions
"0" == 0  // true (!)
"0" === 0 // false (so always prefer ===)
1 + "2"   // ?
```

```
// evaluate a string as a program
let x = eval("1 + 2")

let f = eval(`(function() { return ${x} })`)
f()          // ?
f.toString() // ?
```

Static types: TypeScript

- TypeScript is a **dialect** of JavaScript that extends the language with *optional* static type annotations
- TypeScript is a **superset** of JavaScript: every valid JavaScript program is a valid Typescript program, but not the other way around.
- TypeScript supports **type inference**: types can sometimes be derived based on program context. Values for which the type cannot be derived are given the `any` type
- Typescript's type system is **unsound**:
  - The `any` type is considered compatible with all other types
  - A program that type-checks may still fail with a runtime type error
  - Typescript is translated into JavaScript by *removing* the type annotations (and the compiler does *not* insert additional runtime type checks!)
  - But still ***useful***: catches many bugs at compile-time, serves as developer documentation, enables the IDE to provide intelligent autocompletion

```
type Point = {x: number, y: number};

function makePoint(x: number, y: number): Point {
  return { x: x, y: y };
}

function toString(point: Point): string {
  return `(${point.x} , ${point.y})`;
}

let p = makePoint(1,2); // p has type Point
p.x;          // p.x has type number
toString(p); // toString(p) has type string
```

Static types: TypeScript

Syntax confusion

- TypeScript is a **dialect** of JavaScript that extends the language with *optional* static type annotations
- TypeScript is a **superset** of JavaScript: every valid JavaScript program is a valid Typescript program, but not the other way around.
- TypeScript supports **type inference**: types can sometimes be derived based on program context. Values for which the type cannot be derived are given the `any` type
- Typescript's type system is **unsound**:
    - The `any` type is considered compatible with all other types
    - A program that type-checks may still fail with a runtime type error
    - Typescript is translated into JavaScript by *removing* the type annotations (and the compiler does *not* insert additional runtime type checks!)
    - But still ***useful***: catches many bugs at compile-time, serves as developer documentation, enables the IDE to provide intelligent autocompletion

An object type declaration

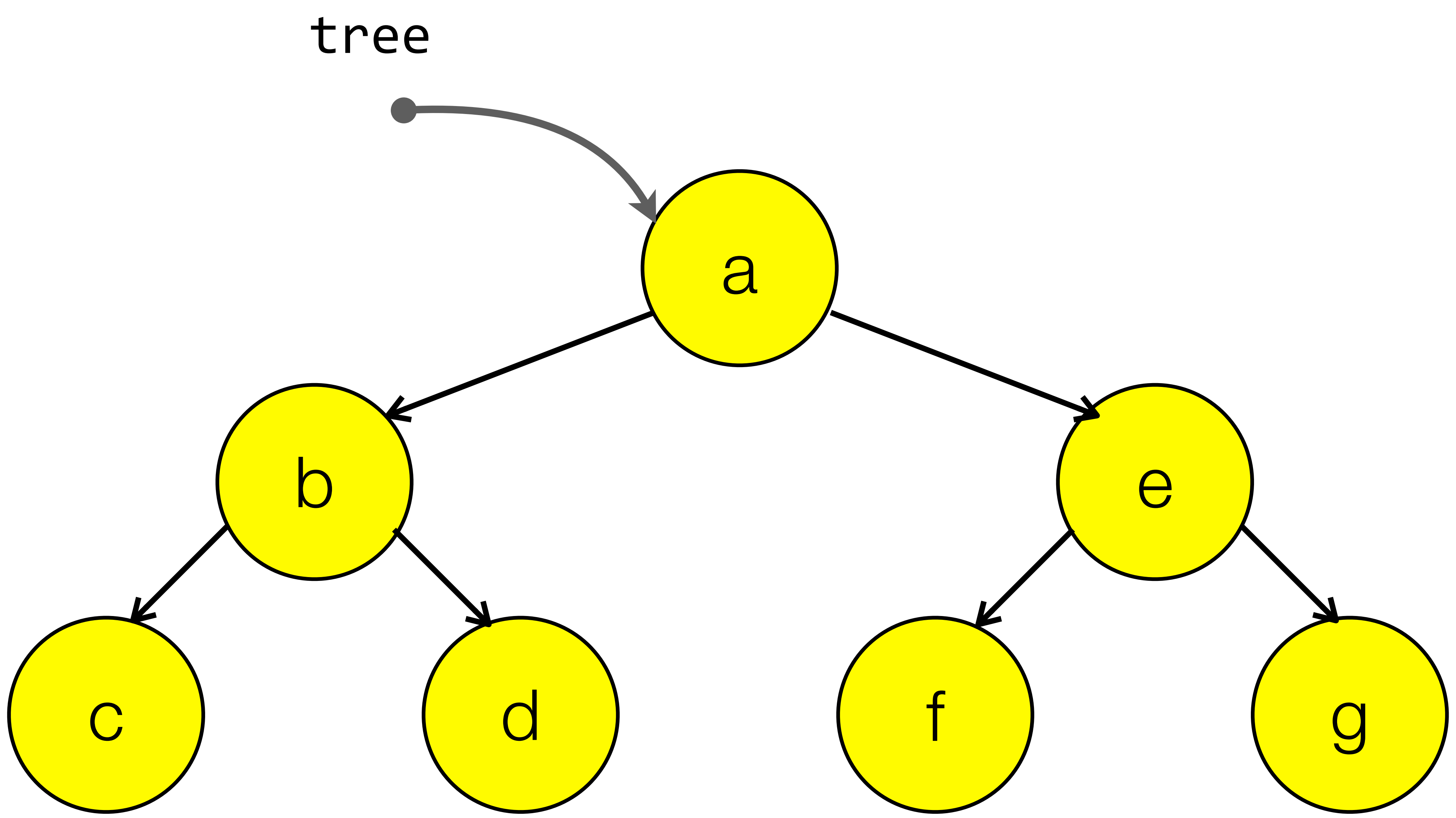
```
type Point = {x: number, y: number};

function makePoint(x: number, y: number): Point {
  return { x: x, y: y };
}

function toString(point: Point): string {
  return `(${point.x} , ${point.y})`;
}

let p = makePoint(1,2); // p has type Point
p.x;          // p.x has type number
toString(p); // toString(p) has type string
```

An object literal

KU LEUVEN DistriNet

Example: binary trees (untyped)

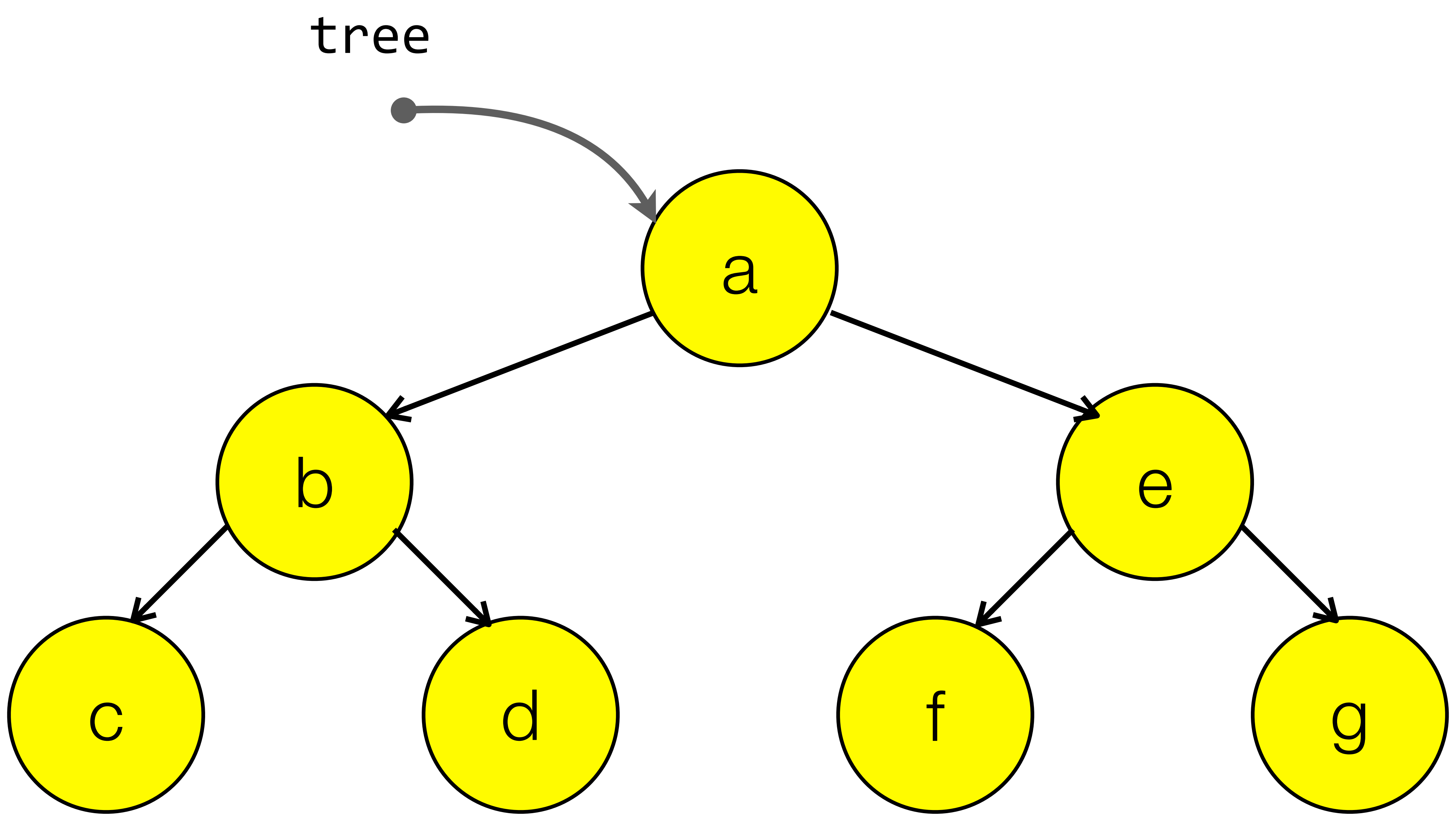
```
let tree = {
  key: "a",
  left: {
    key: "b",
    left:  { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left:  { key: "f" },
    right: { key: "g" }
  }
};
```



Example: binary trees (with Typescript type annotations)

```
type Tree<T> = {
  key     : T,
  left?   : Tree<T>,
  right?  : Tree<T>
}
```

```
let tree: Tree<string> = {
  key: "a",
  left: {
    key: "b",
    left:  { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left:  { key: "f" },
    right: { key: "g" }
  }
};
```



Example: binary trees (with Typescript type annotations)

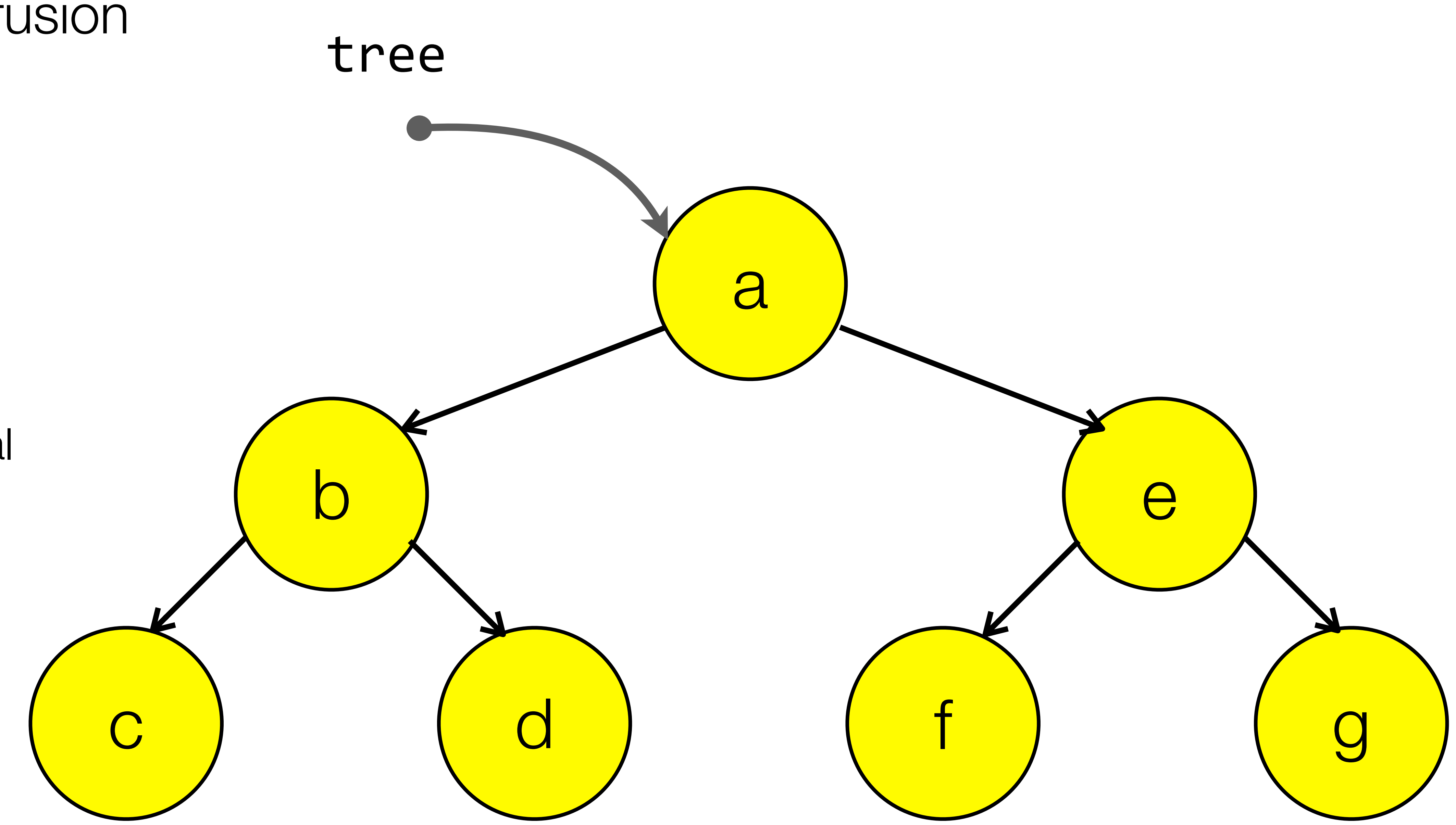
```
type Tree<T> = {
  key    : T,
  left?  : Tree<T>,
  right? : Tree<T>
}
```

Syntax confusion

A type annotation on a variable

```
let tree: Tree<string> = {
  key: "a",
  left: {
    key: "b",
    left:  { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left:  { key: "f" },
    right: { key: "g" }
  }
};
```

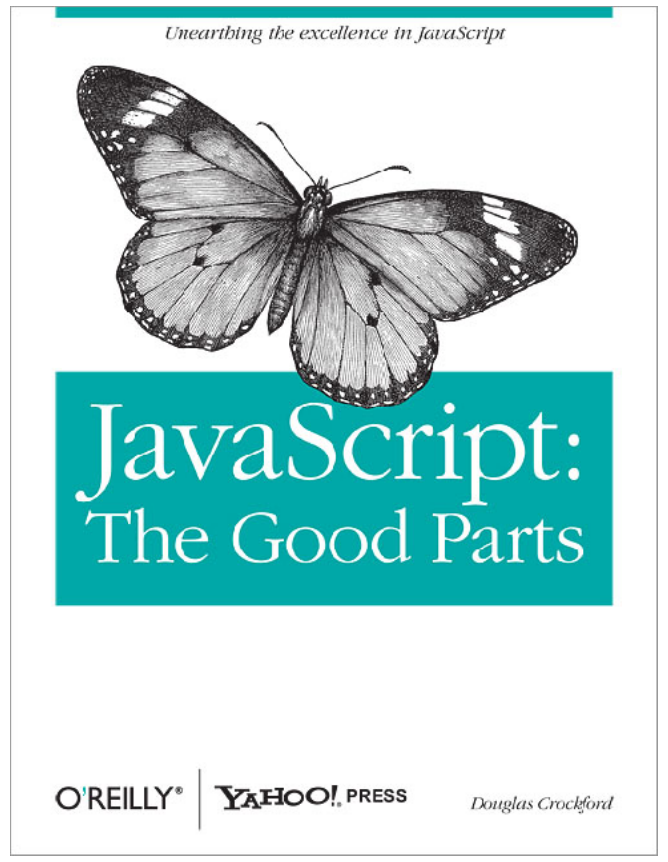
A property definition in an object literal



JavaScript: don’t let the Java-like syntax fool you!

- Java and JavaScript are two very different languages
- Doug Crockford: **“JavaScript is a Lisp in C’s clothing”**
- JavScript is more akin to Scheme or Lisp than it is to Java or C
- Stop and think: why do you think this is the case?

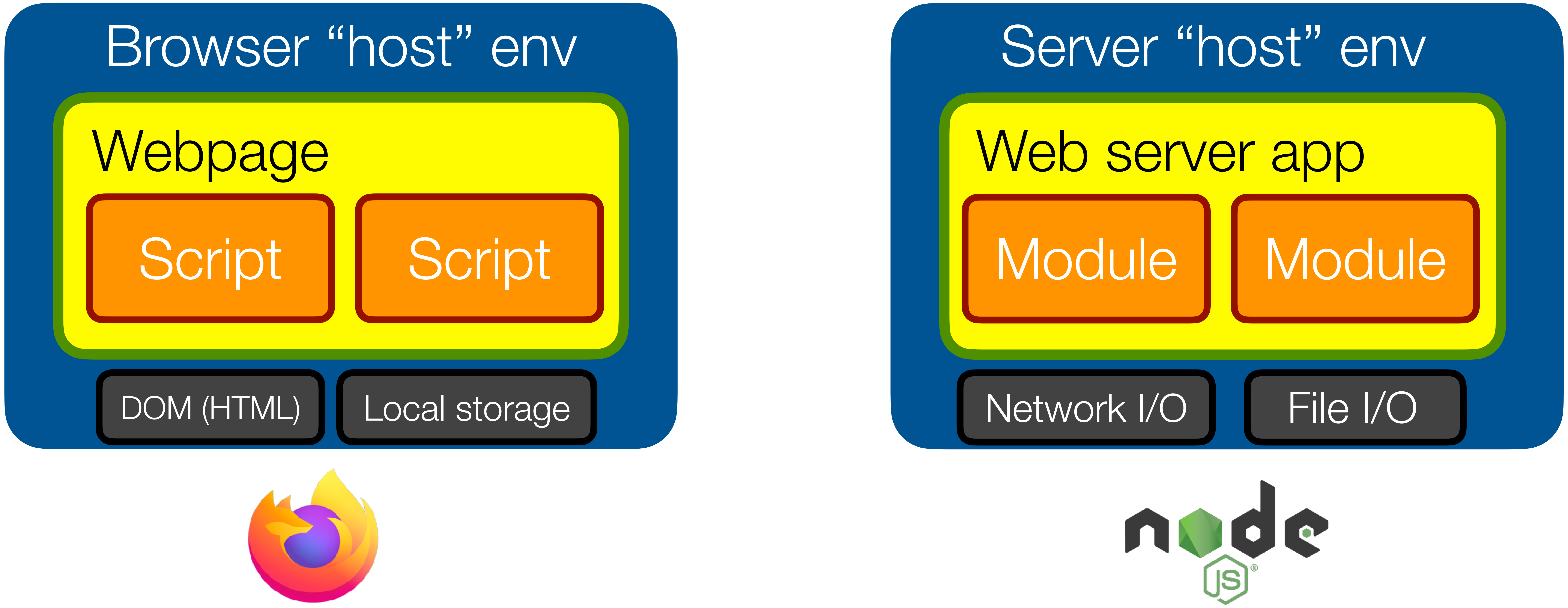
Douglas Crockford,
Inventor of JSON
and author of JS: The Good Parts



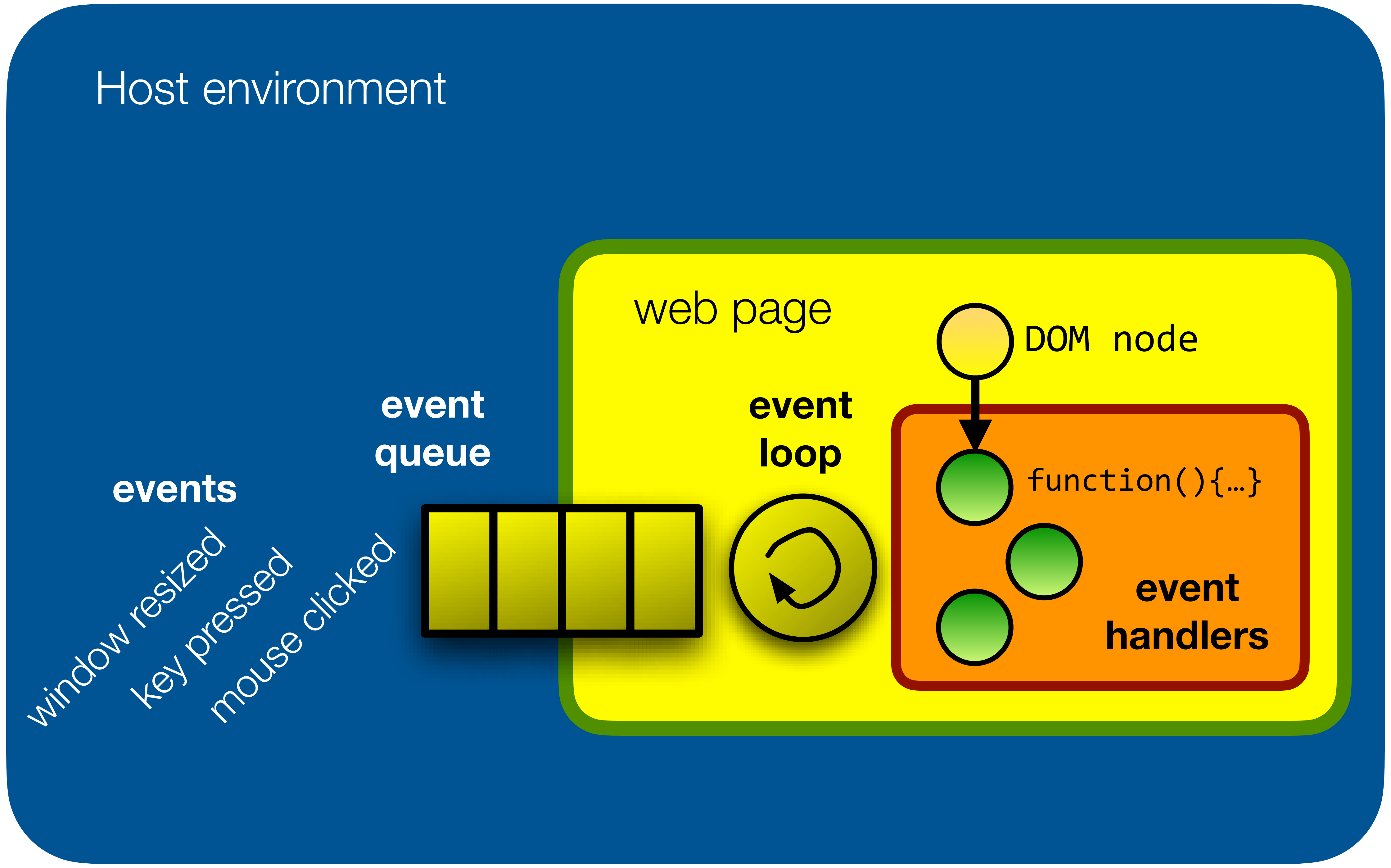
See “JavaScript: The World's Most Misunderstood Programming Language” by Doug Crockford at http://www.crockford.com/javascript/javascript.html for a 2001 perspective on JavaScript

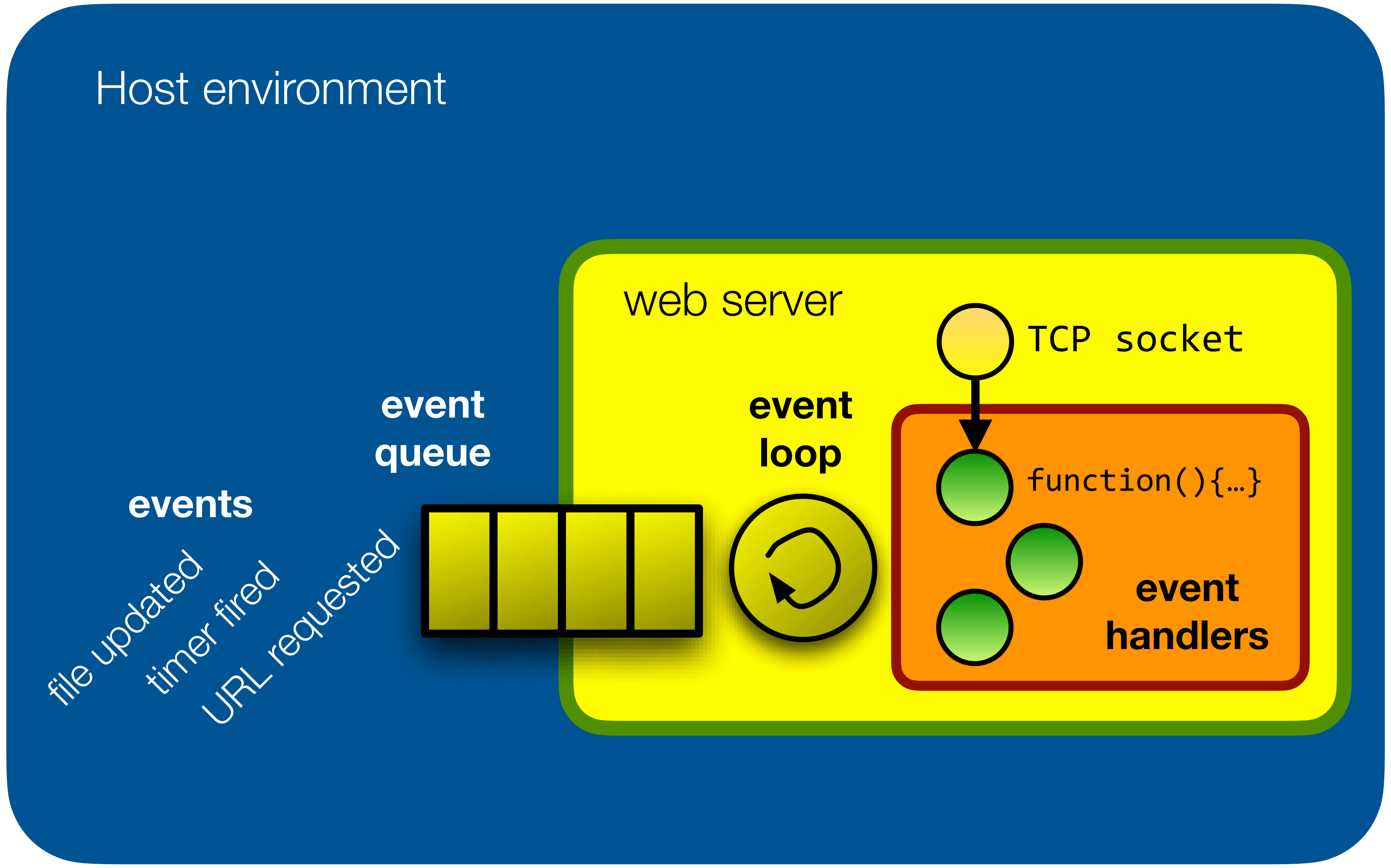
Part 3: Event loops and asynchronous control flow

Recall: scripting languages are “embedded” in a “host” environment



Events, event loops and callbacks





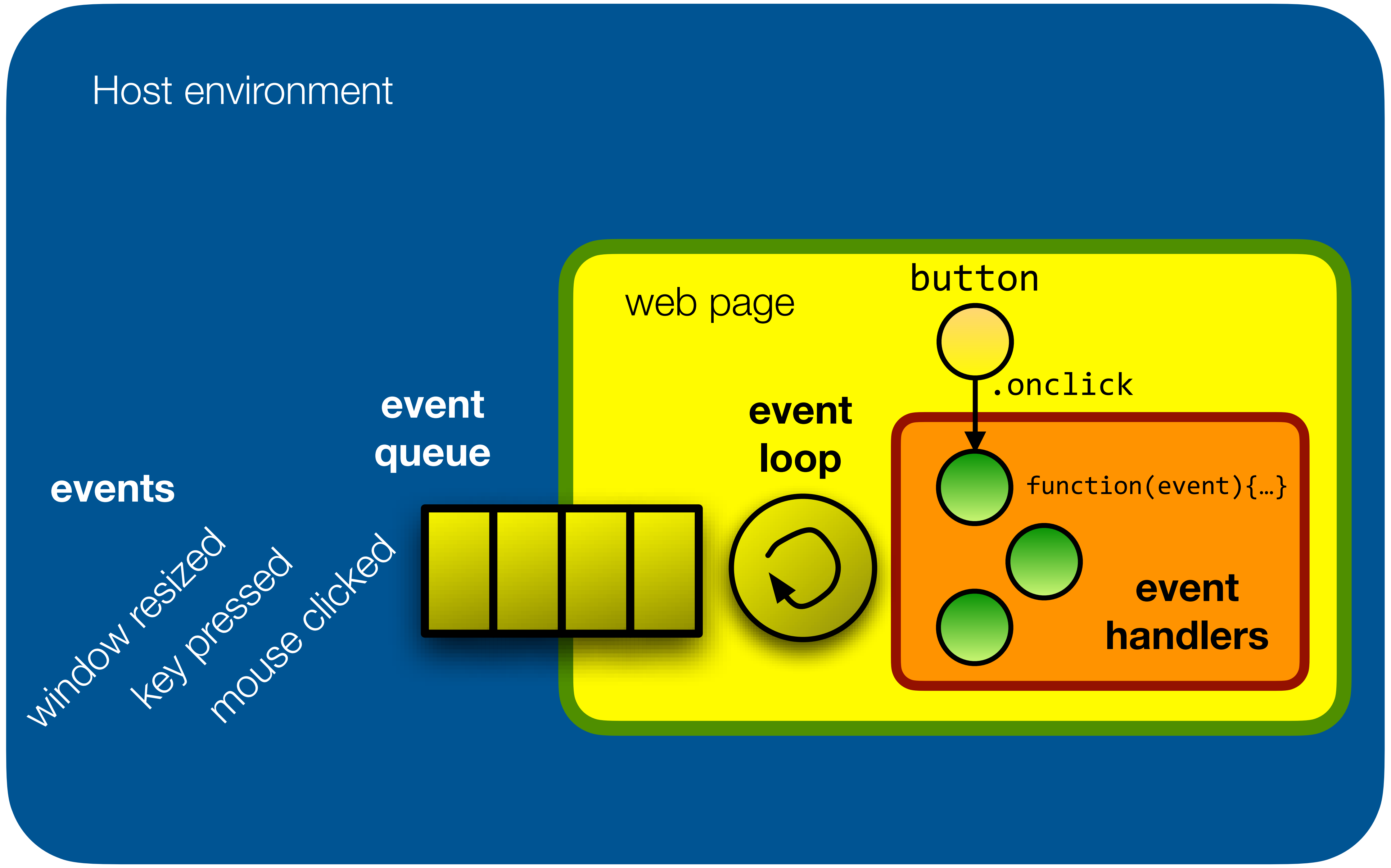
Event loops and event handlers: basic principles

- JavaScript code is called from an infinite loop called the **event loop**
- To respond to an event, **register an event handler** (e.g. when a <script> is first executed)
- The event handler is often a function, called a **callback**
- When the event occurs, it gets enqueued in the event queue
- For each event in order, the event loop dequeues the event and calls the registered callback function (if any), with the event
- Events are **processed one at a time**: the next event is only dequeued and dispatched when the callback function has returned
- When there are no more events to process, the event loop sits **idle** waiting for events
- The event loop is executed by a **single thread** of control
- No parallel event processing, but also no need for concurrency control (i.e. locking)

“Callback” functions: examples in the browser

- In the browser, all JavaScript <script> elements from the same webpage share a single event loop (actually, there is one event loop per *frame* within the webpage)
- Events include page lifecycle events, UI events, timer events, …
- Example UI event: clicking a button

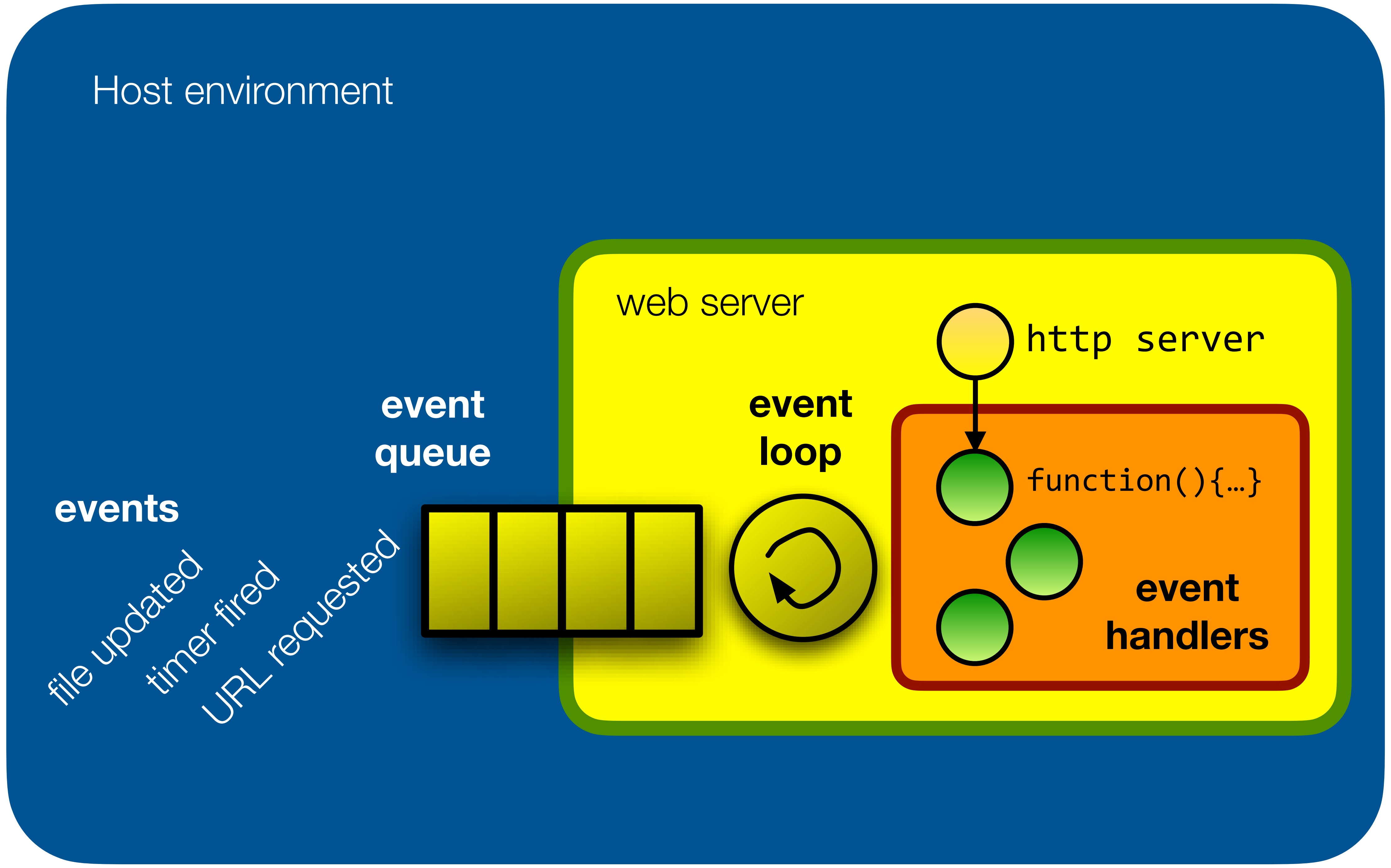
```
let button = document.getElementById("button-id")

button.onclick = function(event) {
  window.alert("button was clicked")
}
```



“Callback” functions: examples on the server

- A node.js process, like a web page, provides a single event loop for code to execute in
- Events include things like incoming HTTP requests, bytes read from a file on disk, operating system interrupt signals, etc.
- Example: responding to HTTP requests

```
let http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
```



Callbacks & The “Hollywood Principle”

- Inversion of control: **“don’t call us, we’ll call you”**


```
let button = document.getElementById("button-id")

button.onclick = function(event) {
  window.alert("button was clicked")
}
```

```
let http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
```

I/O in event loops: the XMLHTTPRequest (XHR) API in the browser

- So far, all actions we saw originated from the host. What if your JS app needs to initiate an action itself? E.g. fetch a URL, lookup a value in a database, read a file…

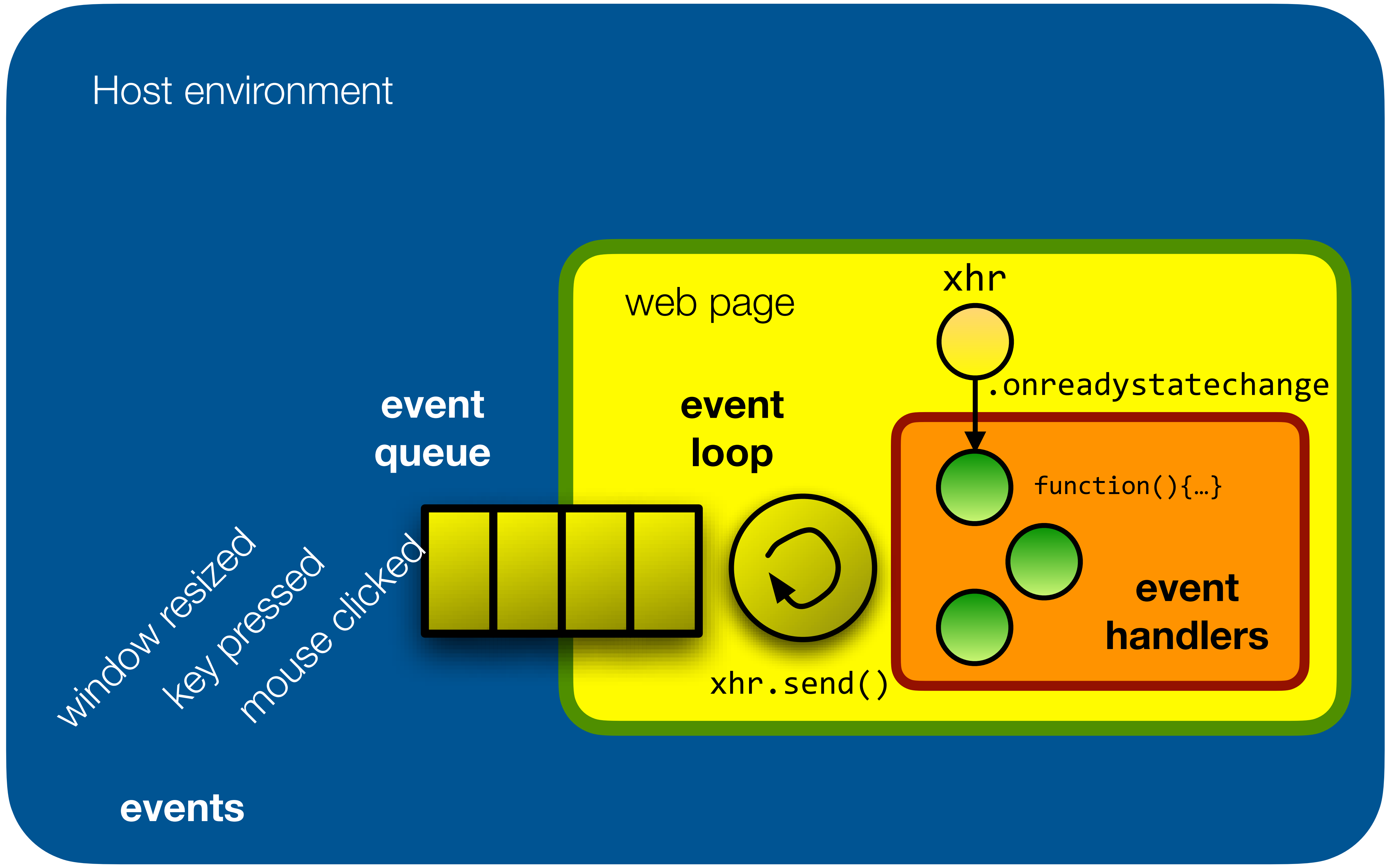
- XMLHTTPRequest is a browser API that allows JavaScript scripts to make HTTP requests to a server, after the page has loaded.

- Legacy API. Modern alternatives exist (see later), but the term “XHR” is still sometimes used to refer to dynamic HTTP requests made by JavaScript scripts in browsers.

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == XMLHttpRequest.DONE) {
    handleResponse(xhr.responseText);
  }
}
xhr.open("GET", "http://example.com");
xhr.send(); // asynchronous call

function handleResponse(text) {
  // show the text in a new <div> element on the page
  let div = document.createElement("div");
  div.textContent = text;
  document.getElementById("result").appendChild(div);
}
```

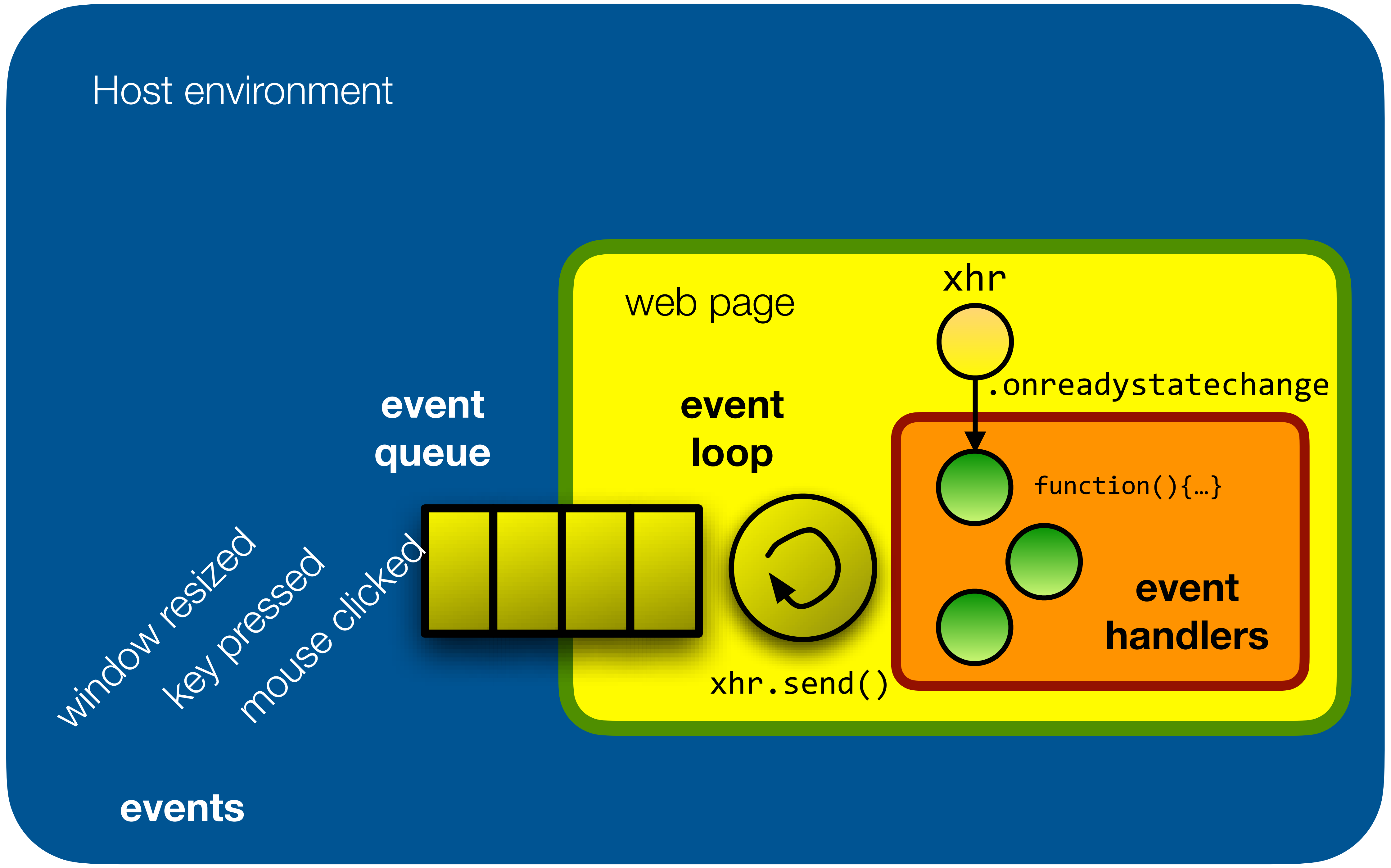
I/O in event loops: why is the XHR asynchronous?

- By making the XHR asynchronous, the event loop is free to process *other events* while the response to the XHR request is pending.
- In particular, UI rendering updates are done by the same event loop thread in between events (when no script code is running)
- If the XHR were synchronous, it would **block** the entire event loop, rendering the entire webpage **unresponsive** while waiting for the server's response!
- Side-note: the browser XHR API actually allows to make blocking (synchronous) XHR calls. It is widely considered bad practice to do so.



I/O in event loops

- The golden rule of event-based programming:
  never block the event loop!



Event loops and “non-blocking” I/O

- Benefits:
  - **Run-to-completion:** simple, consistent model to reason about: functions are never pre-empted while running. Only one function is executing at a time.
  - **Write lock-free code:** no multithreading, so no need to manage locks or worry about data races when reading/writing variables, no need to manage deadlock, etc.
  - **Better resource utilization:** the event loop never “blocks” on external I/O. Get maximum performance out of a single thread of control.

- Drawbacks:
  - **No parallelism:** events cannot be processed in parallel, even if they touch different parts of the application state.
  - **Inversion of control:** whenever we want to do asynchronous I/O, we can no longer use the call stack to sequence control flow (let the caller wait until the callee returns). Instead, we must “nest” the work to be done inside a callback function. This can lead to deeply nested code, sometimes referred to as **“callback hell”**
  - **Harder to debug:** stack traces in event handlers don’t reveal the context of where the event was originally fired. Also, with async I/O, the callee can no longer use exceptions to signal errors, as there is no call stack to unwind.

“Callback Hell”

```
// synchronous call chain

let value1 = step1()
let value2 = step2(value1)
let value3 = step3(value2)
let value4 = step4(value3)
// do something with value4
```

```
// asynchronous call chain
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // do something with value4
            });
        });
    });
});
```

Callbacks: dealing with exceptions

- Normal function calls can return normally, or throw an exception
- **Exceptions don't work for asynchronous operations!** The "caller" has already returned when the operation is executed. There is no more call stack to unwind!
- So, how to handle "exceptions" for asynchronous calls?
- A common pattern is to pass an error object as first argument to the callback function:
  - If the operation succeeded, the error will be `undefined`
  - If the operation failed, the error will contain an `Error` object with details

```
// synchronous call
function readFile(path: string): string;

try {
  let content = readFile("hello.txt");
  // use content
} catch (err) {
  // handle error
}
```

```
// asynchronous call
function readFile(path: string,
                  cb: (e: Error, v: string) => void);

readFile("hello.txt", function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
})
```

Promises

- A promise is a placeholder for a value that may only be available in the future
- Introduced in recent versions of JavaScript (after 2015)
- Most asynchronous APIs now use Promises instead of callbacks

```
function readFile(path: string,
                  cb: (e: Error, v: string) => void);

// callback-based asynchronous call
readFile("hello.txt", function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
})
```

```
function readFile(path: string) => Promise<string>;

// Promise-based asynchronous function call
let promise = readFile("hello.txt");
promise.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

XMLHTTPRequest (XHR) example revisited

- The modern way to make an HTTP request from a script in the browser:

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == XMLHttpRequest.DONE) {
    handleResponse(xhr.responseText);
  }
}
xhr.open("GET", "http://example.com");
xhr.send(); // asynchronous call

function handleResponse(text) {
  // show the text in a new <div> element on the page
  let div = document.createElement("div");
  div.textContent = text;
  document.getElementById("result").appendChild(div);
}
```

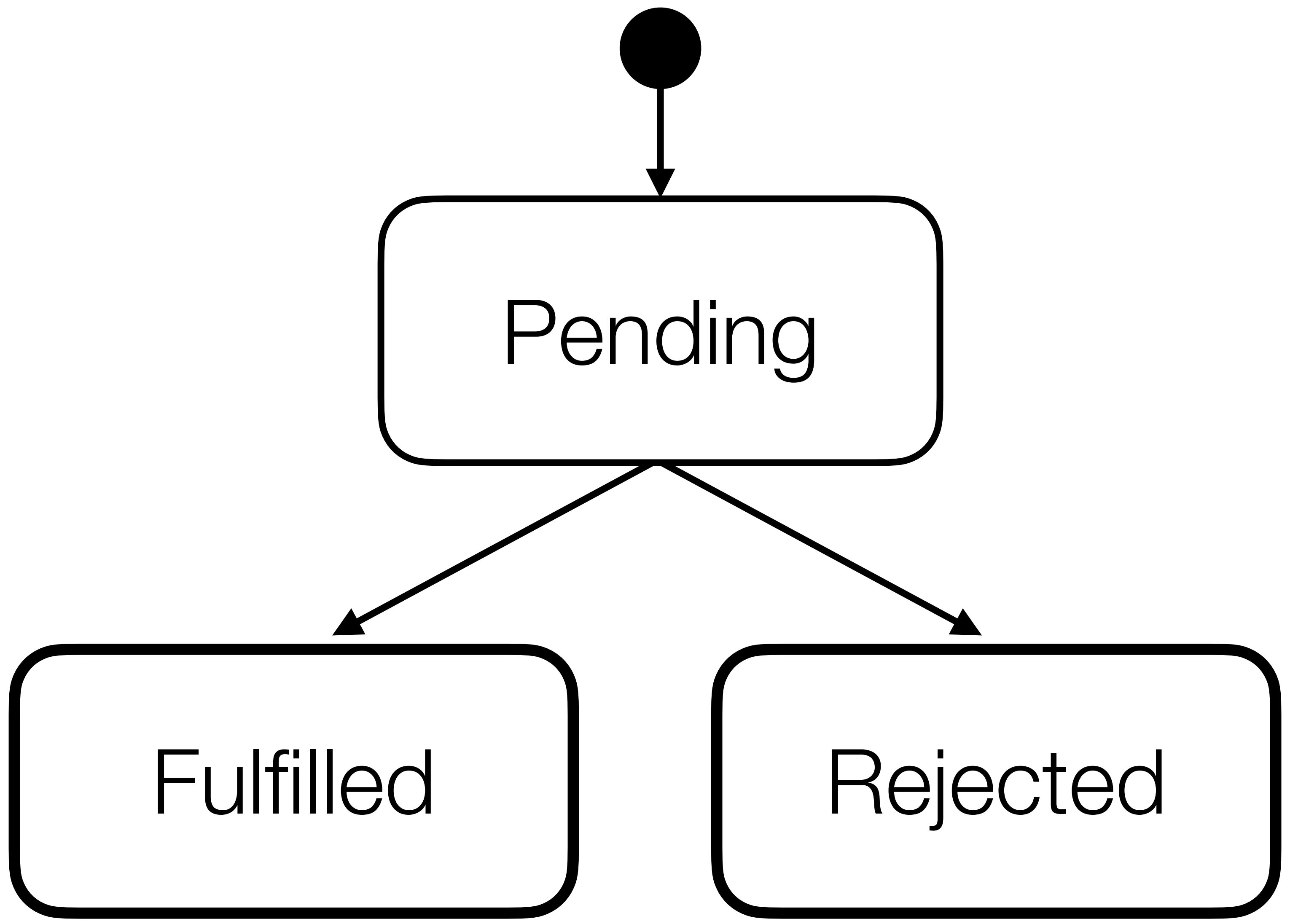
➡

```
let response = fetch("http://example.com");

response.then(text => {
  // show the text in a new <div> element on the page
  let div = document.createElement("div");
  div.textContent = text;
  document.getElementById("result").appendChild(div);
});
```

Promises

- A promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- It is an object that can be in one of three states:
  - **Pending**: the initial state
  - **Fulfilled** (with a value)
  - **Rejected** (with an error)
- Once a promise is either fulfilled or rejected, it remains in that state.

```
let promise = readFile("hello.txt");
// A: promise is pending
promise.then(function (content) {
  // B: promise is fulfilled with a value
}, function (err) {
  // C: promise is rejected with an error
});
```



Promise “chaining”

- Have we really solved the problem? We are still passing callback functions to the `then` method.

- Promises have a secret ability: they can be “chained”:

```
let promise = readFile("hello.txt");
promise.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

Promise “chaining”

- Have we really solved the problem? We are still passing callback functions to the `then` method.

- Promises have a secret ability: they can be “chained”:

```
let promise = readFile("hello.txt");
let p2 = promise.then(function (content) {
  // transform content
}, function (err) {
  // recover from error
});
```

Promise "chaining"

- A call to `then` returns a "chained" promise

- The success and failure callbacks passed to `then` may themselves return a value or throw an exception

- This return value (or exception) is then used to fulfill (or reject) the chained promise

- Resolving a promise `p1` with another promise `p2` causes `p1` to eventually become fulfilled/rejected with the same value/error as `p2`

```
let promise = readFile("hello.txt");
let p2 = promise.then(function (content) {
  // decode may throw
  return decode(content);
}, function (err) {
  // fall back to another file
  return readFile("default.txt");
});
```

Promise chaining solves the problem of “callback hell”

```
function step1(value, callback): void;

step1(function (e, value1) {
    if (e) { return handleError(e); }
    step2(value1, function(e, value2) {
        if (e) { return handleError(e); }
        step3(value2, function(e, value3) {
            if (e) { return handleError(e); }
            step4(value3, function(e, value4) {
                if (e) { return handleError(e); }
                // do something with value4
            });
        });
    });
});
```

```
function step1(value): Promise;

step1()
.then(value1 => step2(value1))
.then(value2 => step3(value2))
.then(value3 => step4(value3))
.then(function (value4) {
    // do something with value4
})
.catch(function (error) {
    // handle any error here
});
```

Promise "combinators"

Plain promises

```
function concatFiles(path1, path2) {
  let p1 = readFile(path1);
  let p2 = readFile(path2);

  return p1.then(text1 => {
    return p2.then(text2 => {
      return text1 + text2;
    });
  });
}

concatFiles("a.txt", "b.txt").then(val => {
  writeFile("merged.txt", val);
});
```

Promise “combinators”

Plain promises

```
function concatFiles(path1, path2) {
  let p1 = readFile(path1);
  let p2 = readFile(path2);

  return p1.then(text1 => {
    return p2.then(text2 => {
      return text1 + text2;
    });
  });
}

concatFiles("a.txt", "b.txt").then(val => {
  writeFile("merged.txt", val);
});
```

Promise combinators

```
function concatFiles(path1, path2) {
  let p1 = readFile(path1);
  let p2 = readFile(path2);

  return Promise.all([p1, p2]).then(vals => {
     let [text1, text2] = vals;
     return text1 + text2;
  });
}

concatFiles("a.txt", "b.txt").then(val => {
  writeFile("merged.txt", val);
});
```

Promise “combinators”

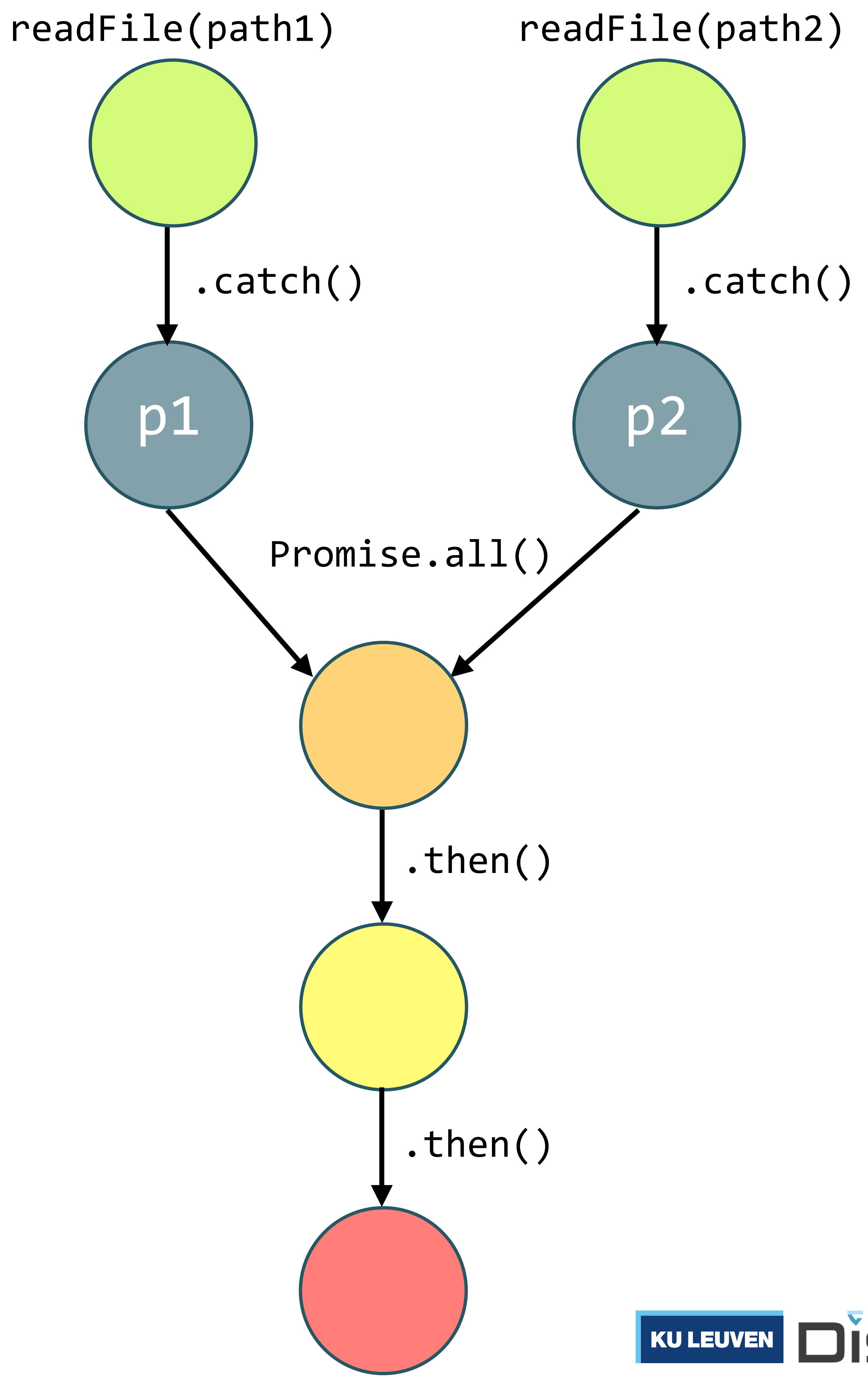
- And now with fallback error logic:

```
function concatFiles(path1, path2, default) {
  let p1 = readFile(path1).catch(err => readFile(default));
  let p2 = readFile(path2).catch(err => readFile(default));

  return Promise.all([p1, p2]).then(vals => {
     let [text1, text2] = vals;
     return text1 + text2;
  });
}

concatFiles("a.txt", "b.txt", "c.txt").then(val => {
  writeFile("merged.txt", val);
});
```

Promise “combinators”

- And now with fallback error logic:

```
function concatFiles(path1, path2, default) {
  let p1 = readFile(path1).catch(err => readFile(default));
  let p2 = readFile(path2).catch(err => readFile(default));

  return Promise.all([p1, p2]).then(vals => {
     let [text1, text2] = vals;
     return text1 + text2;
  });
}

concatFiles("a.txt", "b.txt", "c.txt").then(val => {
  writeFile("merged.txt", val);
});
```



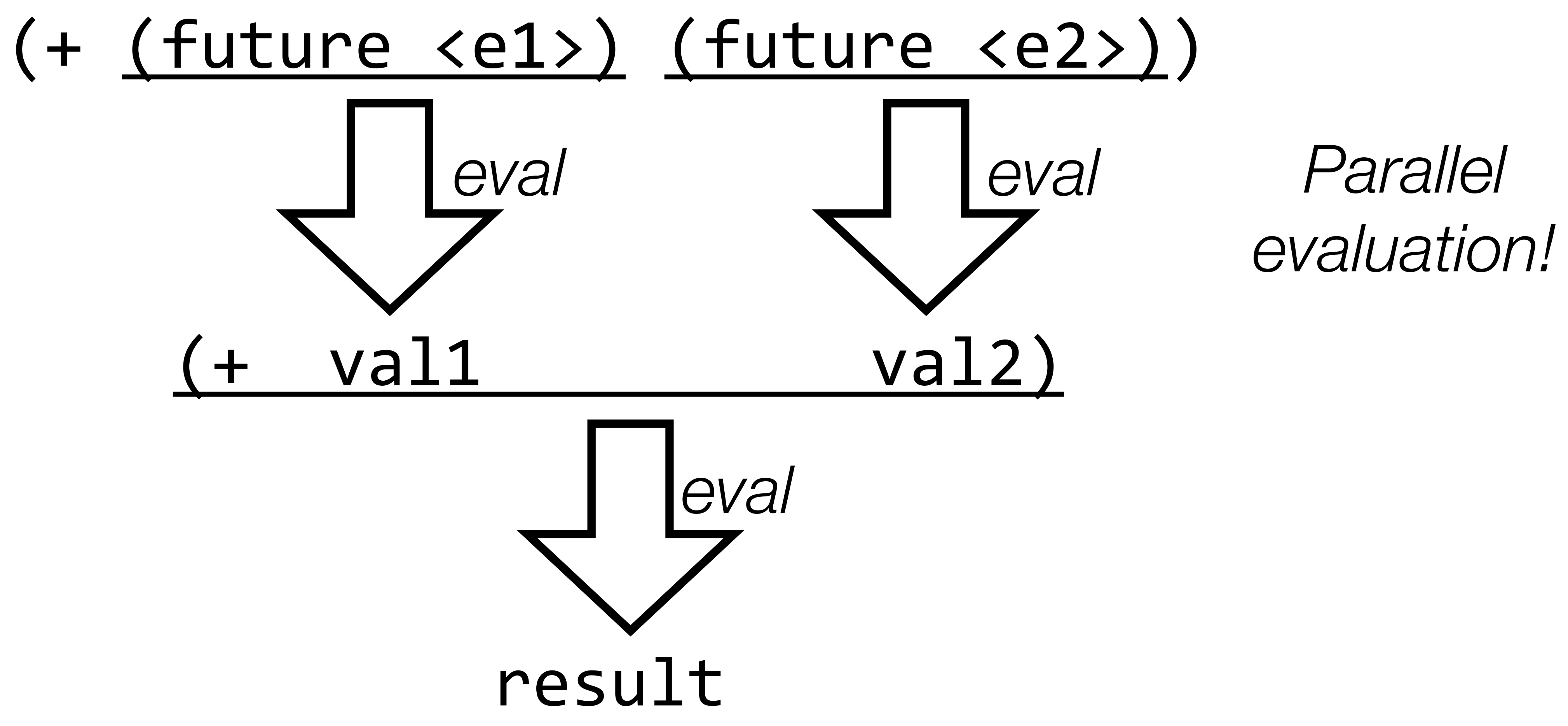
Promise “combinators”

- Promise.all: fulfills when **all** of the promises fulfill; rejects when **any** of the promises rejects.
- The fulfilled value is an array of fulfilled values of the input promises (in the same order)
- Promise.any: fulfills when **any** of the promises fulfills; rejects when **all** of the promises reject.
- The fulfilled value is the value of the *first* input promise to be fulfilled
- Other combinators exist

```
function Promise.all(inputs: Promise<T>[]): Promise<T[]>;
function Promise.any(inputs: Promise<T>[]): Promise<T>;

let vals = [1, 2, 3]
let proms = vals.map(v => Promise.resolve(v))

Promise.all(proms).then(vals => console.log(vals)) // [1,2,3]

Promise.any(proms).then(val => console.log(val))   // 1
```

Promises: origins and other uses

- Compared to callbacks, **promises make *delayed computation* explicit as *data***
- Managing delayed computation using a promise-like concept is an old idea in computer science
- First mention: a 1976 paper by Daniel P. Friedman (the author of this course's textbook!)
- First explored in the context of **parallel computing** in Lisp-like languages
- Later also explored in the context of **distributed computing** to represent the result of non-blocking remote procedure calls
- JavaScript's promises were influenced by Promises in the *E* programming language (Miller, 1997), with additional influences from the *Twisted* framework's "Deferred" objects (a node.js-like framework for Python), which were ported to JavaScript in the *Dojo* framework (Zyp, 2007)
- Wikipedia has a reasonable page on the topic to learn more: https://en.wikipedia.org/wiki/Futures_and_promises



Promises: beyond JavaScript

- **Related concepts** in other programming languages and frameworks: "**futures**", "**deferreds**", "**tasks**".
- Many differences in terms of API: explicit vs implicit use (is Promise<T> a subtype of T?), read-only vs read-write access to the Promise's value, blocking vs non-blocking access to the value.
- Beware that terms are used inconsistently across languages! (E.g. a Scala Promise is not identical to a JavaScript Promise)

C#: `Task<T>`

Java: `CompletableFuture<T>`

Python: `asyncio.Future`

Swift: `Tasks` and `async`

Scala: `Future[T]` and `Promise[T]`

Promises: review

- Compared to callbacks, promises **make *delayed* computation explicit as data**

- Benefits:

  - Delayed computation can now be **composed** through standard function composition

  - Because Promise objects explicitly distinguish success from failure paths, they support principled handling and **automatic propagation of errors** (versus manual error propagation with callbacks)

- But:

  - We must still wrap delayed computation in **nested functions** (syntax overhead)

  - We still **cannot use** our familiar **sequential control flow** constructs (e.g. while-loops, return statement, try-catch-finally statement) when dealing with asynchronous activities

  - Can we have our cake and eat it too?

Async functions

- Modern (post-2017) versions of JavaScript support two new keywords to manage asynchronous activities using standard sequential control flow: `async` and `await`

- `async` is a modifier that can be used to mark a function as an *Async function*

- `await expr` is an expression that evaluates `expr` to a Promise value `p` and then turns the *continuation* of the enclosing Async function into a delayed computation on `p` (*as if* wrapping the code that follows in a function `f` and calling `p.then(f)`)

- The `await` statement can only occur syntactically directly within the body of an Async function

- Async functions always return a Promise. In TypeScript, the return type of an Async function must be of type `Promise<T>`

```
function readFile(path: string): Promise<string>;
```

```
// Promise-based asynchronous call

let promise = readFile("hello.txt");
promise.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

```
// asynchronous call using async/await

async function() {
  try {
    let content = await readFile("hello.txt");
    // use content (it is a string, not a promise!)
  } catch (err) {
    // handle error
  }
}
```

Async functions combine sequential control flow with asynchronous execution

```
function step1(value): Promise;

function run() {
  step1()
  .then(value1 => step2(value1))
  .then(value2 => step3(value2))
  .then(value3 => step4(value3))
  .then(value4 => {
      // do something with value4
  })
  .catch(error => {
      // handle any error here
  });
}
```

```
function step1(value): Promise;

async function run() {
  try {
    let value1 = await step1();
    let value2 = await step2(value1);
    let value3 = await step3(value2);
    let value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}
```

Async functions combine sequential control flow with asynchronous execution

```
function step1(value): Promise;

function run() {
  step1()
  .then(value1 => step2(value1))
  .then(value2 => step3(value2))
  .then(value3 => step4(value3))
  .then(value4 => {
      // do something with value4
  })
  .catch(error => {
      // handle any error here
  });
}
```

```
function step1(value): Promise;

async function run() {
  try {
    let value1 = await step1();
    let value2 = await step2(value1);
    let value3 = await step3(value2);
    let value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}
```

Async functions versus Promises: examples

```
async function foo() {
  return 42;
}
```

⇒

```
function foo() {
  return Promise.resolve(42);
}
```

```
async function foo2() {
  throw new Error("reason")
}
```

⇒

```
function foo2() {
  return Promise.reject(new Error("reason"))
}
```

```
async function bar() {
  let v1 = await foo();
  let v2 = await foo();
  return v1 + v2;
}
```

⇒

```
function bar() {
  return foo()
    .then(v1 => (foo().then(v2 => v1 + v2)));
}
```

Async functions versus Promises: more examples

```
// download multiple files in parallel
function fetchAll(urls: string[]): Promise<string>[];

// process a single file
function process(file: string): Promise<string>;

// download in parallel, then process sequentially
async function processSequentially(urls) {
  let promises = fetchAll(urls);
  let results = []

  for (let fileP of promises) {
    try {
      let file = await fileP;
      results.push(await process(file));
    } catch (err) {
      results.push(undefined);
    }
  }
  return results;
}
```

Async functions versus Promises: more examples

```
// download multiple files in parallel
function fetchAll(urls: string[]): Promise<string>[];

// process a single file
function process(file: string): Promise<string>;

// download in parallel, then process sequentially
async function processSequentially(urls) {
  let promises = fetchAll(urls);
  let results = []

  for (let fileP of promises) {
    try {
      let file = await fileP;
      results.push(await process(file));
    } catch (err) {
      results.push(undefined);
    }
  }
  return results;
}
```

Async functions versus Promises: more examples

```
// download multiple files in parallel
function fetchAll(urls: string[]): Promise<string>[];

// process a single file
function process(file: string): Promise<string>;

// download in parallel, then process sequentially
async function processSequentially(urls) {
  let promises = fetchAll(urls);
  let results = []

  for (let fileP of promises) {
    try {
      let file = await fileP;
      results.push(await process(file));
    } catch (err) {
      results.push(undefined);
    }
  }
  return results;
}
```

```
// download multiple files in parallel
function fetchAll(urls: string[]): Promise<string>[];

// process a single file
function process(file: string): Promise<string>;

// download in parallel, then process sequentially
function processSequentially(urls) {
  let promises = fetchAll(urls);
  let results = [];

  function processNext(promises, i) {
    if (i === promises.length)
      return Promise.resolve(results);

    return promises[i]
      .then(file => process(file), err => undefined)
      .then(result => {
        results.push(result);
        return processNext(promises, i+1);
      });
  }

  return processNext(promises, 0);
}
```

Async functions versus Promises: more examples

```
// download multiple files in parallel
function fetchAll(urls: string[]): Promise<string>[];

// process a single file
function process(file: string): Promise<string>;

// download in parallel, then process sequentially
async function processSequentially(urls) {
  let promises = fetchAll(urls);
  let results = []

  for (let fileP of promises) {
    try {
      let file = await fileP;
      results.push(await process(file));
    } catch (err) {
      results.push(undefined);
    }
  }
  return results;
}
```

```
// download multiple files in parallel
function fetchAll(urls: string[]): Promise<string>[];

// process a single file
function process(file: string): Promise<string>;

// download in parallel, then process sequentially
function processSequentially(urls) {
  let promises = fetchAll(urls);
  let results = [];

  return promises.reduce((waitForPrev, promise) => {
    return waitForPrev
      .then(_ => promise)
      .then(file => process(file), err => undefined)
      .then(result => { results.push(result); });
  }, Promise.resolve(undefined))
  .then(_ => results);
}
```

Wrap up

Summary

- JavaScript is “a Lisp in C’s clothing”: it has C-like syntax, but Lisp-like **first-class functions and closures**
- JavaScript is a “dynamic language”: flexible, but sometimes dangerous
- JavaScript is a “**scripting language**”: it is embedded in a “host” environment
- Most JavaScript host environments use an **event loop** execution model
- Simple, single-threaded execution. But: computation or I/O must never block!
- Hence, **computation must often be delayed** until events arrive, or until responses are available from earlier asynchronous requests. How to manage delayed computation?
- We have reviewed three techniques: **Callbacks**, **Promises** and **Async functions**.

Exercise session

- Focus on the use of Promises and Async functions.
- Available on GitHub: https://github.com/tvcutsem/promises-exercises