Construction of User Interfaces (SE/ComS 319)

Jinu Susan Kabala

Department of Computer Science

Iowa State University, Fall 2021

EVENT-DRIVEN PROGRAMMING

Outline

- Event-Driven Programming (EDP):
  - Concepts
    - Event handling
    - Event-driven architecture
    - Asynchronous programming, etc.
- Web UI and EDP with JavaScript
- UI and EDP with Node.js

What is event-driven programming?

- A programming paradigm in which the flow of the program is determined by **events** such as:
  - User actions (mouse clicks, key presses, motion/talking)
  - Sensor outputs (mostly in embedded systems)
  - Messages from other programs/threads (device drivers)
- Programmer does not control when code is executed
  - User controls that; programmer provides capabilities then the user invokes them
- "event handlers”: small bits of code that the application calls when certain events occur

Event-driven programming – Applications

- Event-driven programming
  - … is the dominant paradigm used in **graphical user interfaces** and other applications that perform certain actions in response to user input!
    - e.g. JavaScript web applications: performing actions in response to user input.
  - … is widely used in **Human-computer interaction (HCI)**

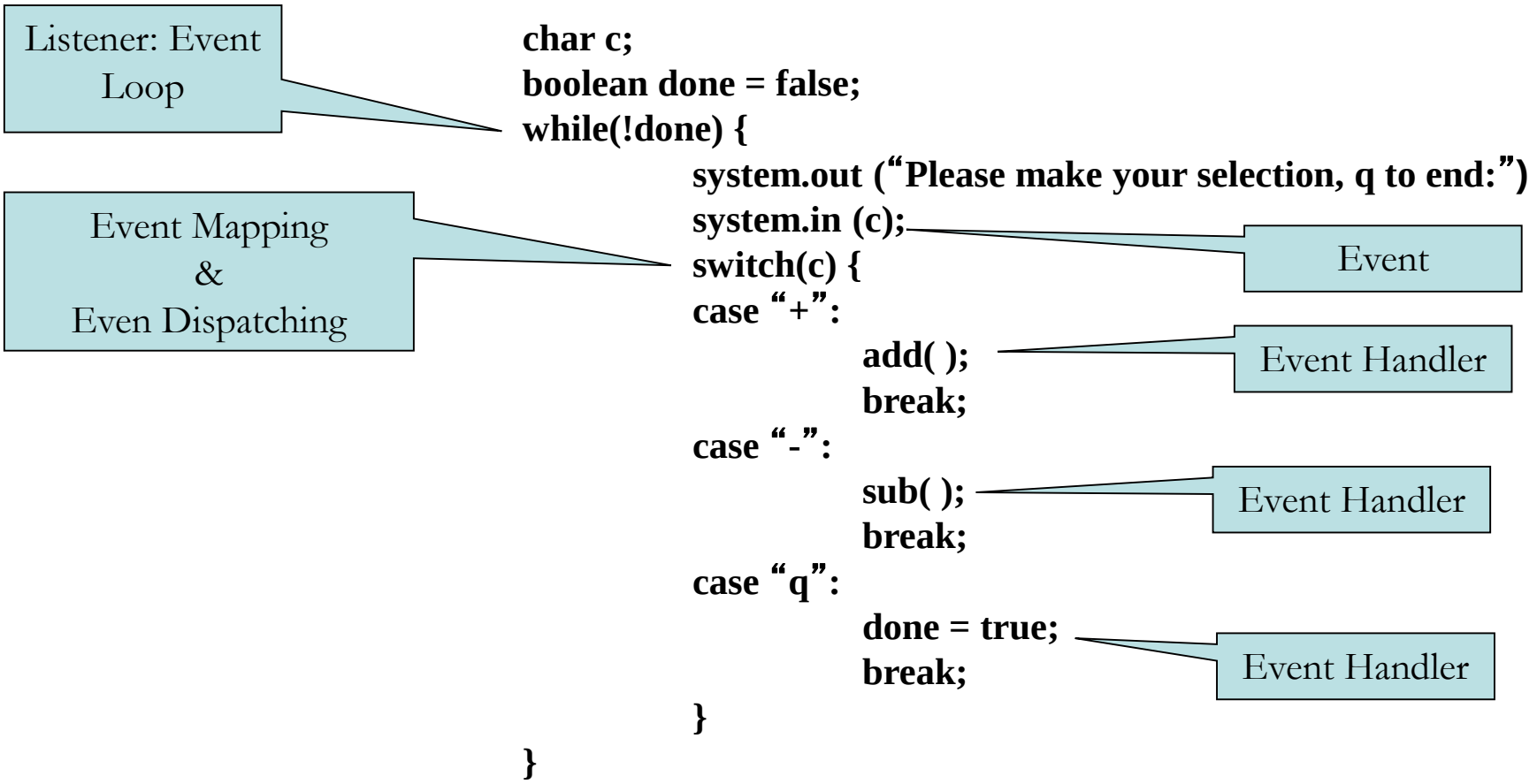
Human-computer interaction (HCI)

- HCI: Interactive computing systems for human use
  - CLI: command line interface (with keyboard)
  - **GUI: graphical user interface (mouse/touch)**
  - NUI: natural user interface with Audio/Video (Kinect)
- A main HCI component: **Interaction** – four components:
  - User interaction
  - Event
  - Event Handling
  - Output
- A **GOOD GUI** allows users to perform interactive tasks easily:
  - **What you see is what you get**

Event-driven programming – Sequence of events/actions

- Application waits (idles) after initialization until the user generates an event through an input device (keyboard, mouse, …).
- The OS dispatches the event to the application who owns the active window.
- The corresponding event handler(s) of the application is invoked to process the event.

Event-Driven programming – main components

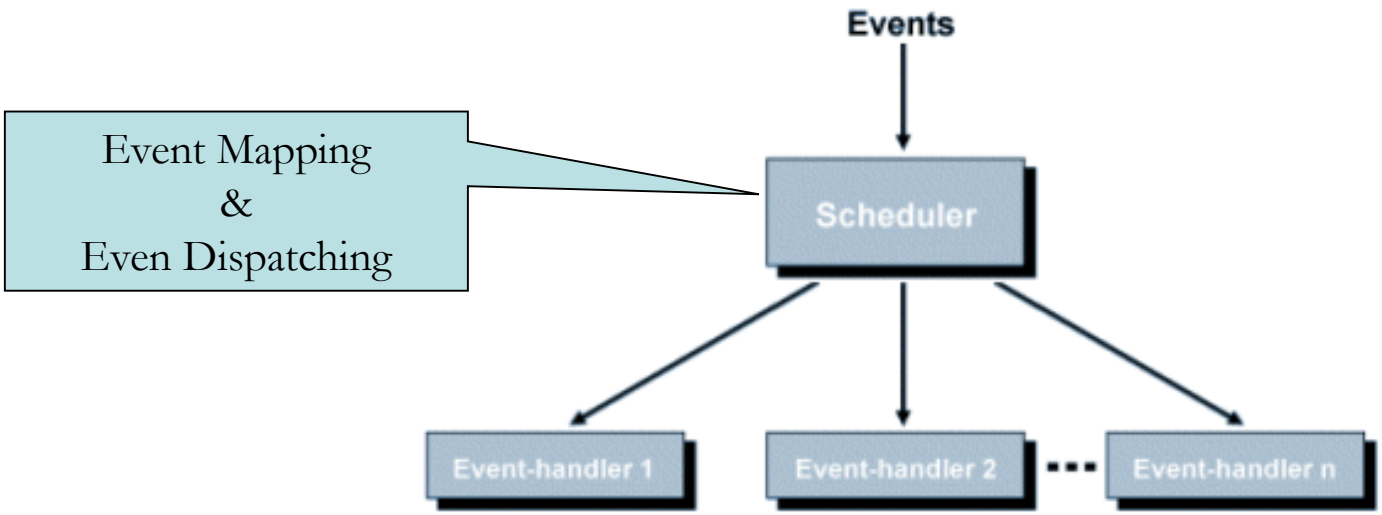


Event-driven programming – main components (2)

1. Event generators: GUI components (e.g. buttons, menus, …)
2. Events/Messages: e.g. MouseClick, …
3. Event loop (Listener): an **infinite loop** constantly waits for events
4. Event mapping / Event registration: informs event dispatcher which event is corresponding to which event handler
5. Event dispatcher: dispatch events to the corresponding event handlers
6. Event handlers: methods for processing events. E.g. OnMouseClick(), …

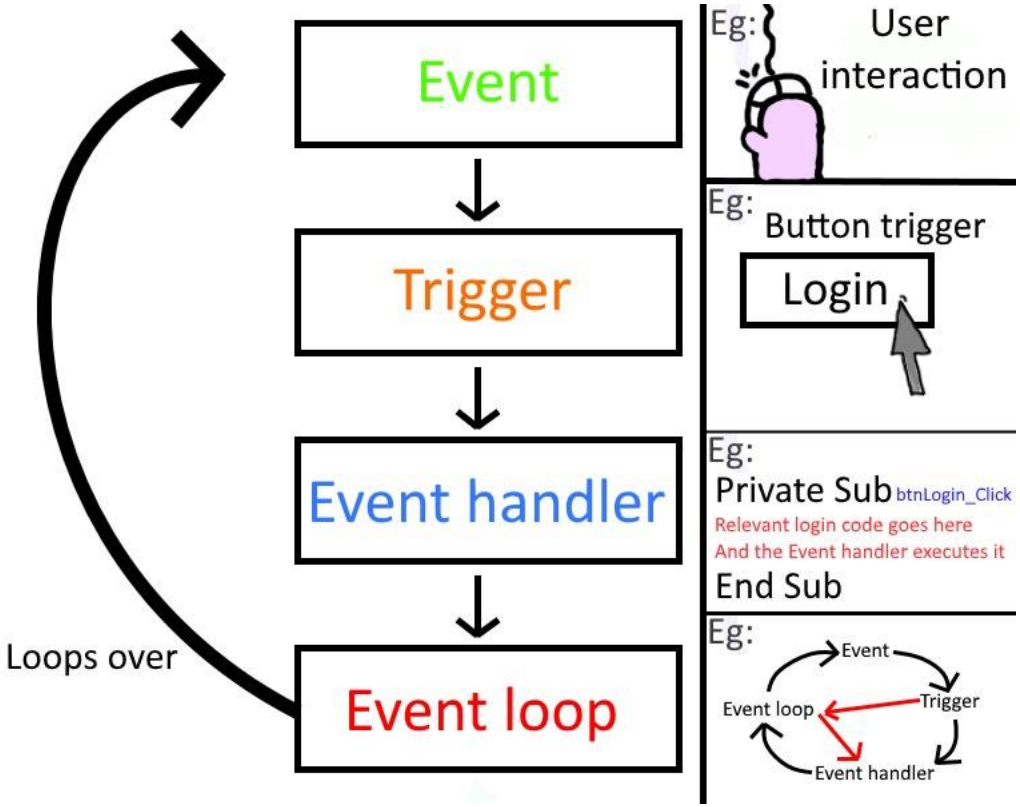
Event-driven programming paradigm

- Event-driven programming paradigm:



Event-driven programming paradigm (2)

- Event-driven programming paradigm:



https://commons.wikimedia.org/wiki/File:Event_driven_programming_Simply_Explained.jpg

Benefits of event-driven programming

- GUI applications can experience the biggest benefits:
  - Intuitive, responsive and flexible
  - Compatible with Object Oriented and builds upon OO Programming
  - Naturally well-suited to applications whose control flow are based on **internal or external events** (those applications that watch changes in application states)
  - Leads to better software design

Benefits of event-driven programming (2)

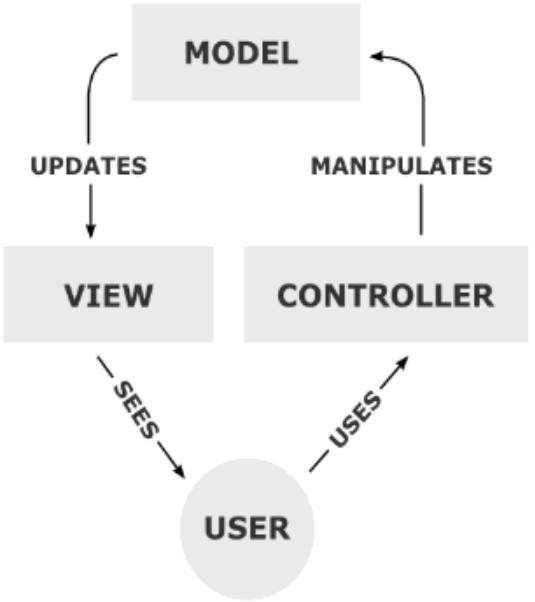
- GUI applications can experience the biggest benefits:
  - Speeds up processing (data distributed across multiple processors/handlers)
    - Ideal for RAD (rapid application development): less planning and more working prototypes
  - Improves application scalability (easily extend your application)
  - Increases loose coupling
  - ➔ Serverless computing
    - Write your functions, upload the code one cloud providers (AWS Lambda) and let them handle all the backend work!

Event-Driven Programming with JavaScript

WEB USER INTERFACES

Event-Driven Programming (EDP) – Web UI

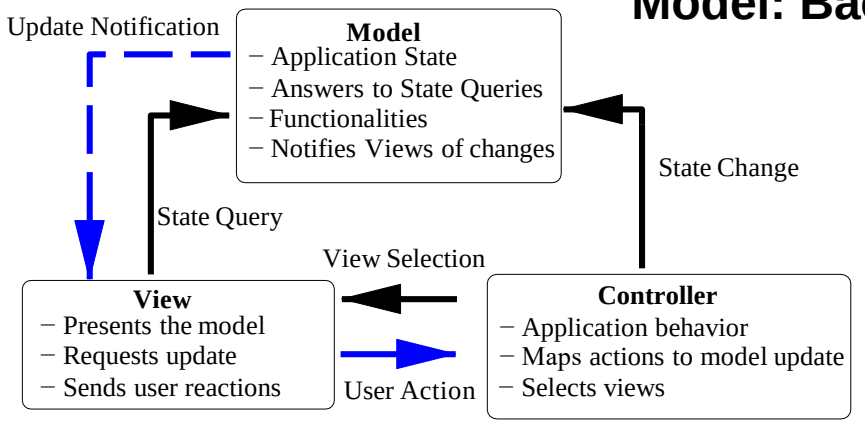
- **MVC (Model – View – Controller) in Web UI** (EDP)**:**
  - **View**: Browser presentation (HTML)
  - **Model**: Data (Backend Database or (simple) embedded)
  - **Controller**:
    - Client scripts/programs, e.g. JavaScript
    - Server scripts/programs, e.g. Node.js



MVC architecture

- Model-View-Controller architecture:

Model: Backend Database



View: HTML

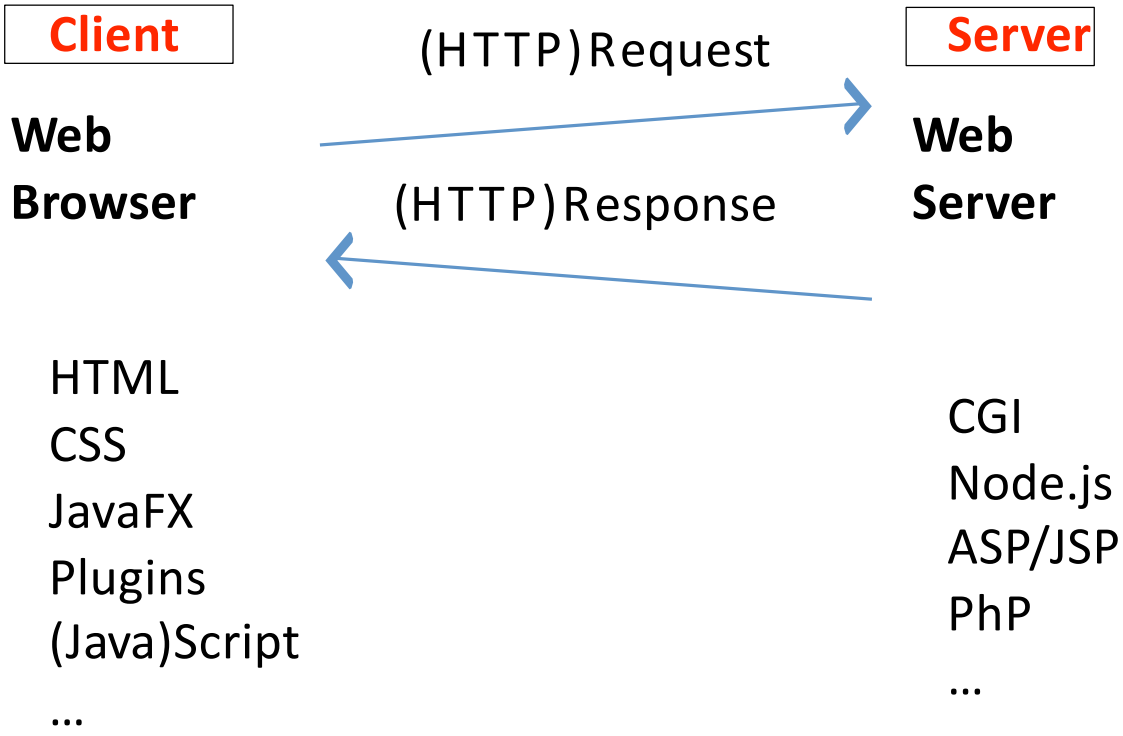
Controller: JavaScript, Node.js code

- Advantages:
  - Loosely coupled, modular
  - Model with different views
  - Controller decides when/how to update the model and/or the view
  - Model can change the view

Client/Server programming

- Use **client-side** programming for
  - Validating user input
  - Prompting users for confirmation, presenting quick information
  - Calculations on the client side
  - Preparing user-oriented presentation
  - Any function that does not require server-side information
- Use **server-side** programming for
  - Maintaining data across sessions, clients, applications

Web software: Client/Server (1)

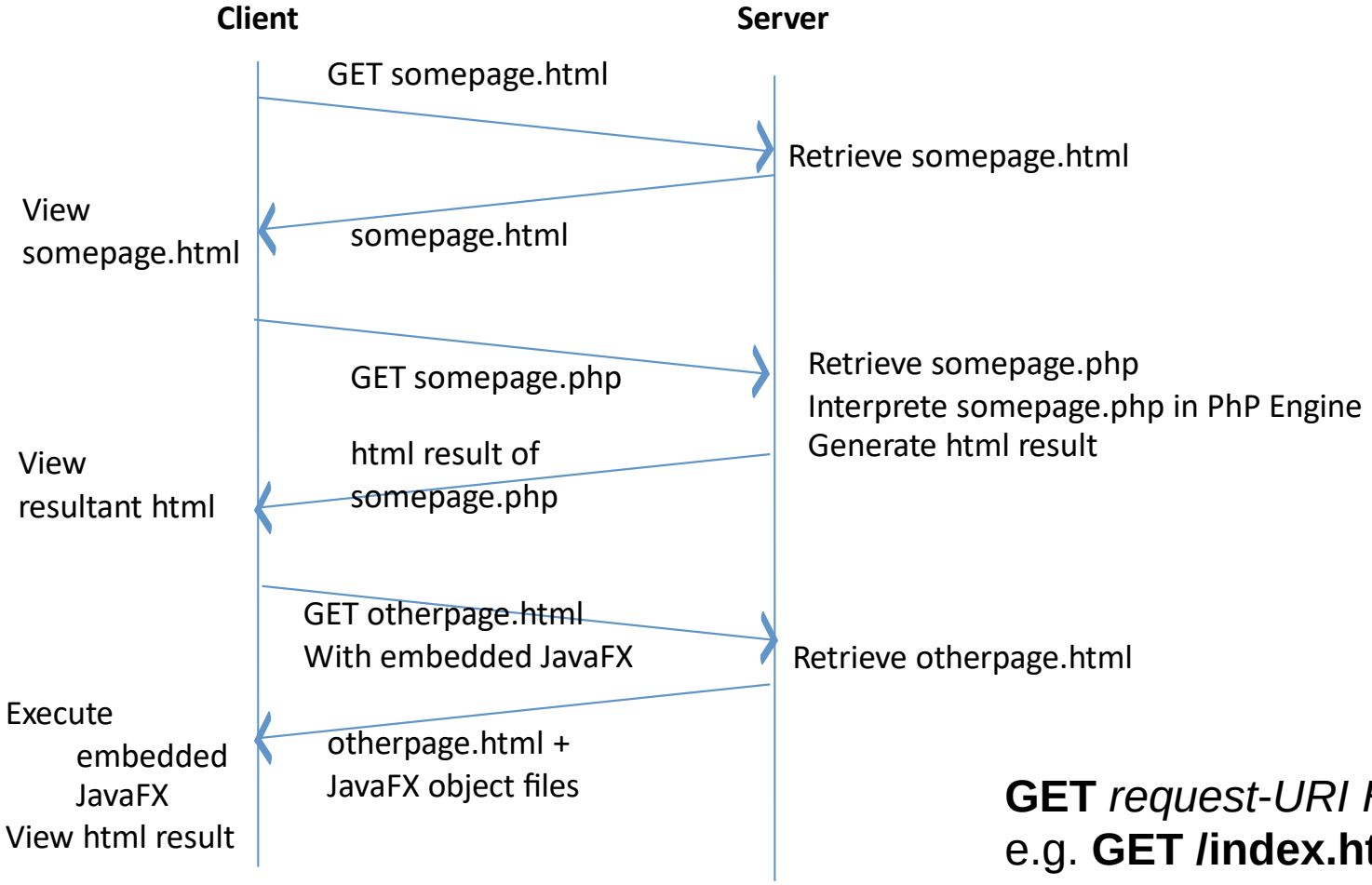


HTTP (Hypertext Transfer Protocol): HTTP is a client-server application-level protocol. It typically runs over a TCP/IP connection.

Web software: Client/Server (2)

- Web-client and Web-server communicates using HTTP protocol
  - Client can send a **HTTP request**: method “`get`" or “`post`"
  - Server can read a HTTP request and produce **HTTP response**
- Server-side programs should be capable of reading HTTP request and producing HTTP response

Web software: Client/Server (3)



GET *request-URI HTTP-version*
e.g. **GET /index.html HTTP/1.0**

Common Gateway Interface (CGI) – Classic method for server side programming

- Standard for the **server** to communicate with external applications
- Server receives a client (Http) request to access a CGI program
- Server creates a new process to execute the program
- Server passes client request data to the program
- Program executes, terminates, produces data (HTML page)
- Server sends back (Http response) the HTML page with result to the client

HTML/CGI – Example

```
<html>
<head></head>
<body>
<form action="<some-server side cgi program>" method="post">
First Name: <input type="text" name="fname"/>
Last Name: <input type="text" name="lname"/>
<input type="submit" value="Submit"/>
</form>
</body>
</html>
```

- Once the user clicks the submit button, the data provided in the form fields are “submitted" to the server where it is processed by a CGI program!

HTTP Request/Response Message

- Message Header
  - Who is the requester/responder
  - Time of request/response
  - Protocol used …
- Message Body
  - Actual message being exchanged

HTTP Request

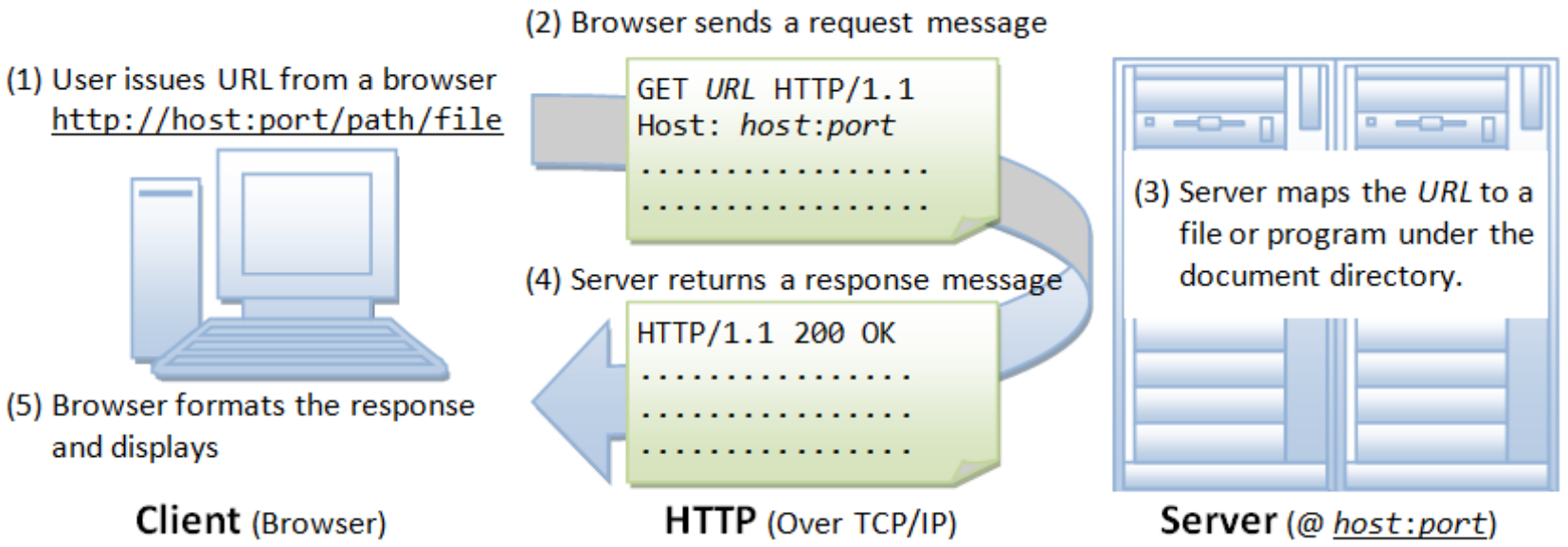
```
GET /index.html HTTP/1.1
Host: http://www.se.iastate.edu
Accept-Language: en
User-Agent: Mozilla/8.0
Query-String: ...
```

HTTP Response

```
HTTP/1.1 200 OK
Date: Sat, 27 Oct 2007 16:00:00 GMT
Server: Apache
Content-Type: text/html
```

- Response Codes:
  - 200s: good request/response
  - 300s: redirection as the requested resource is not available
  - 400s: bad request leading to failure to respond
  - 500s: server failure

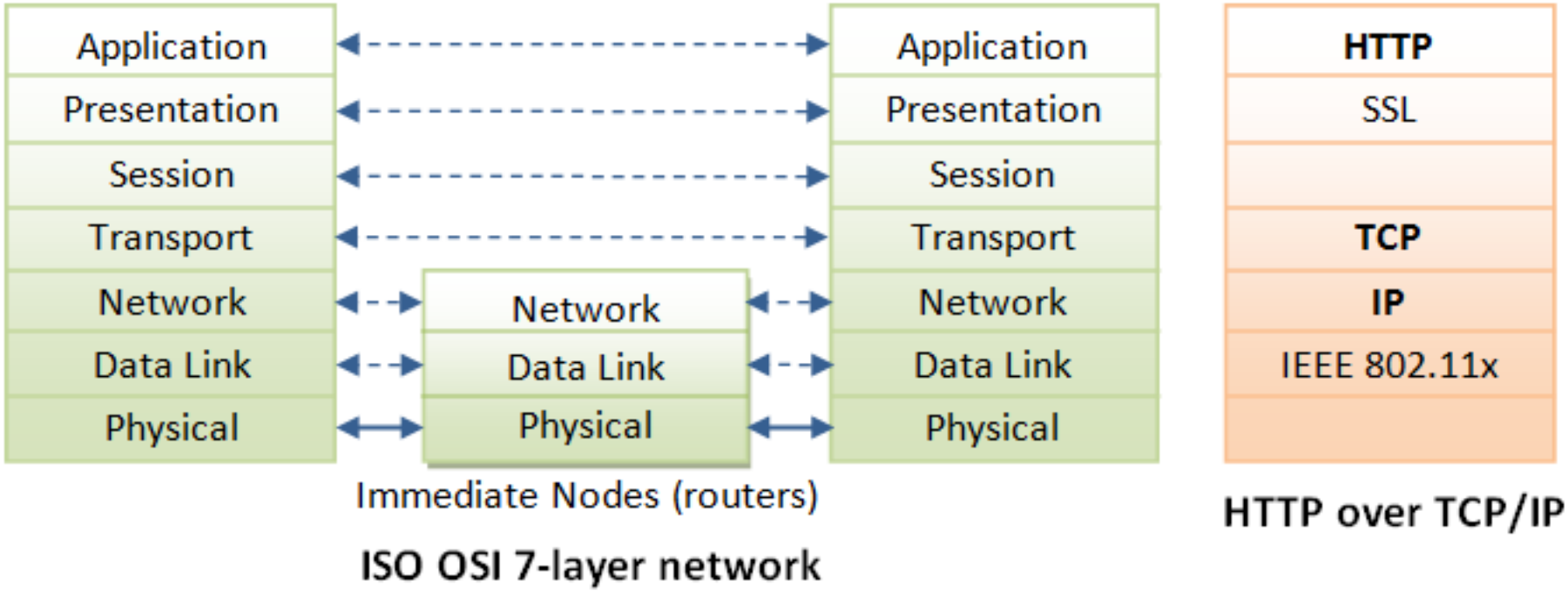
Web software: Client/Server



Source: https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

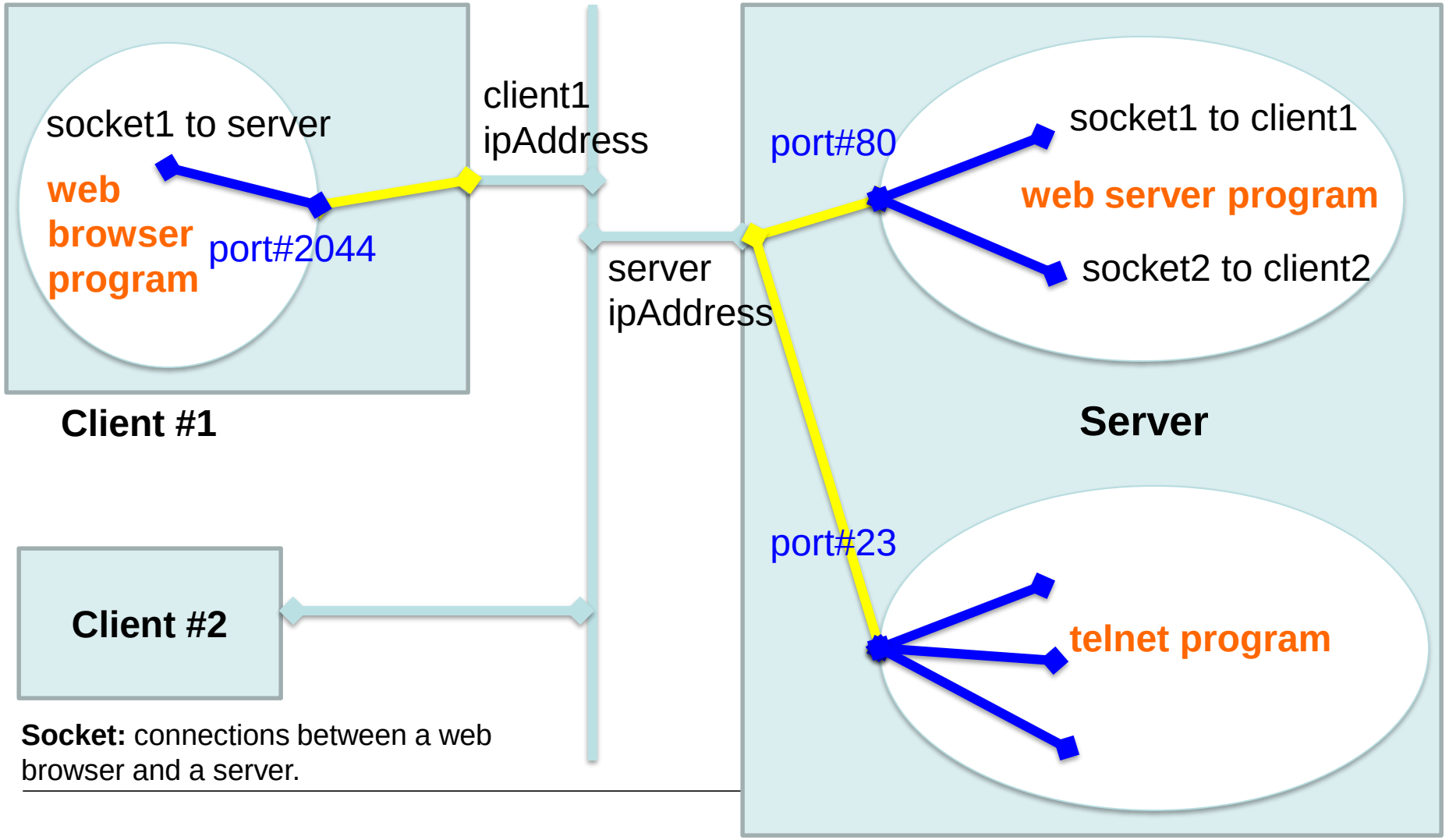
- **GET**: The GET method is used to retrieve information from the given server using a given URL.
  - Requests using GET should only retrieve data and should have no other effect on the data.

Client/Server: HTTP over TCP/IP



Source: https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

Web software: Client/Server – Connections



Socket: connections between a web browser and a server.

EDP – Client-Side Dynamics (1)

- HTML + Javascript
- Html elements: **forms**
- Html style elements: fonts, headings, breaks
- CSS: uniformly manipulate styles
- JavaScript:
  - manipulate styles (CSS)
  - manipulate html elements
  - validate user data
  - communicate with the server-side programs
- In HTML: `<input id="clkb" type="button" value="Click" onclick="clkF()"/>`
- In Javascript file: `function clkF() { alert("Hello"); }`

- Html elements: **View**
- CSS: **Model**
- Javascript: **Controller**

EDP – Client-Side Dynamics (2)

- Html elements: **View**
- CSS: **Model**
- Javascript: **Controller**
- CSS: A simple mechanism for adding style to Web documents.
  - Look & feel of Webpages
  - Layouts, fonts, text, image size, location
  - Objective: Uniform update
- Javascript as a client-side event-driven programming
  - Client-side computations
  - Form validation + warnings
  - Dynamic views

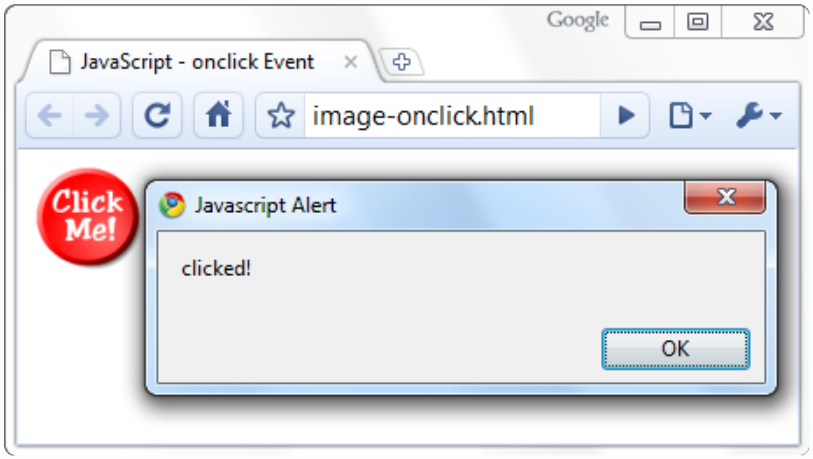
Recap – How to add JavaScript to html file?

- Include in html file:
  - `<script> your javascript code goes in here </script>`
- Can also include from a separate file:
  - `<script src="./01_example.js"></script>`
- Can include from a remote web site:
  - `<script src="http://…./a.js"></script>`

JavaScript Event Handler – Example

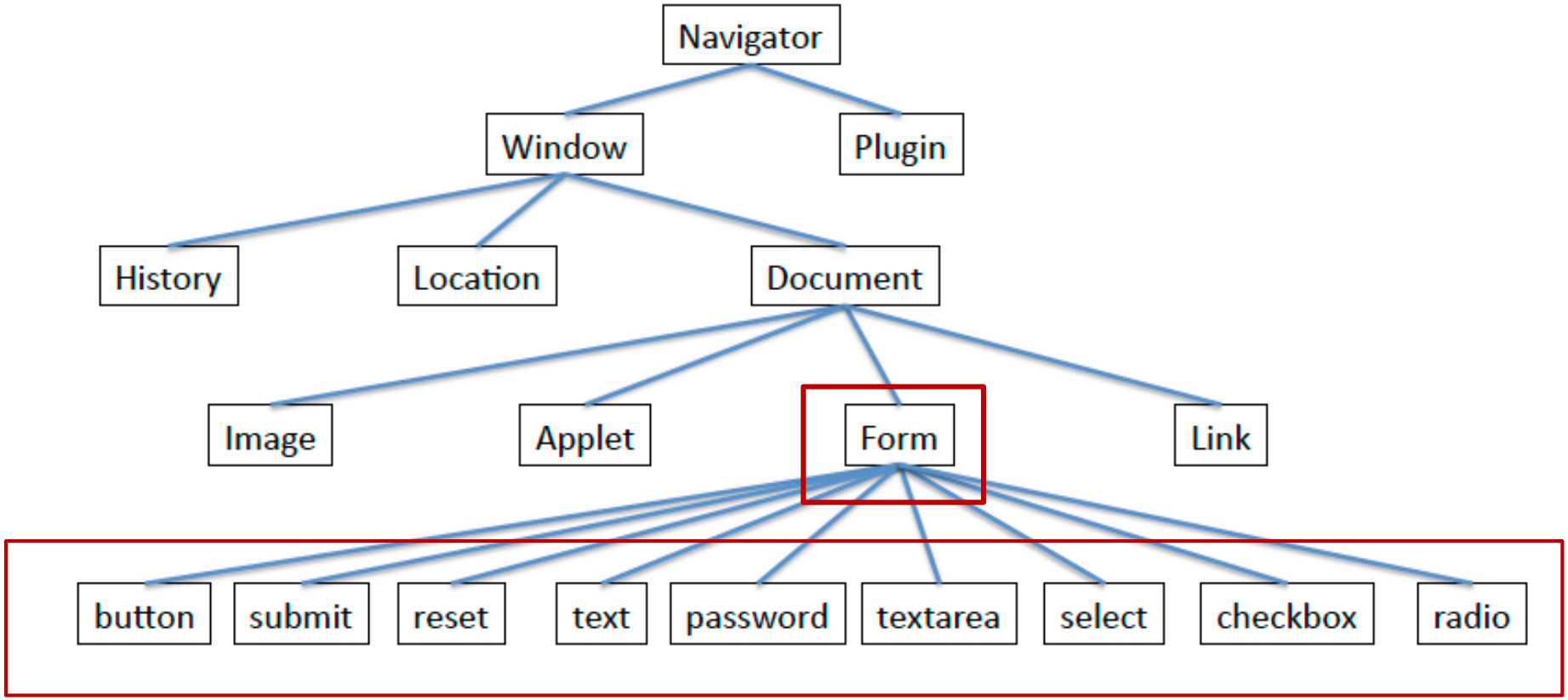
```
<html>
<head>
<script type="text/javascript">
  function test (message) {
    alert(message);
  }
</script>
</head>

<body>
  <img src="logo.gif"
    onclick="test('clicked!')" />
</body>
</html>
```



Using **onclick**, we attach **event handlers**.

JavaScript accessibility hierarchy



EVENT-DRIVEN PROGRAMMING – NODE.JS

Event-driven programming – Node.js

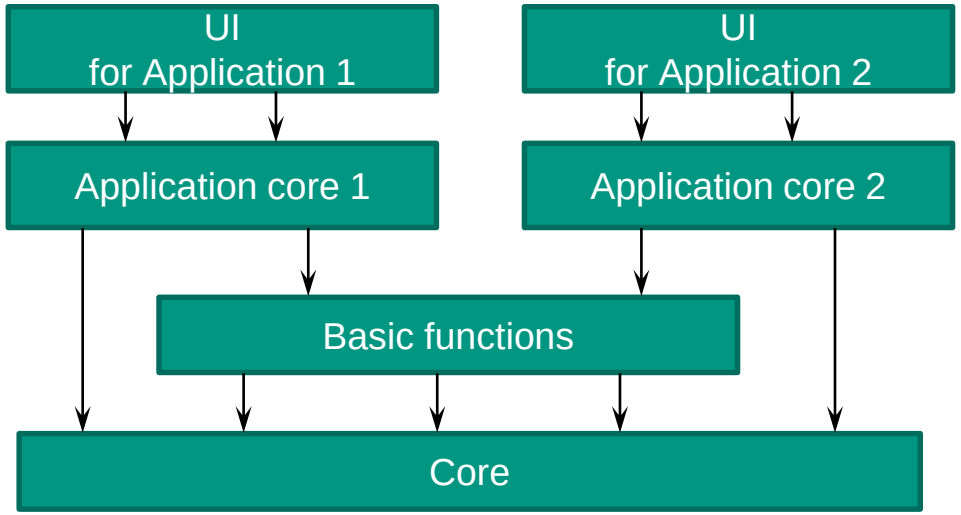
- Event-driven architecture
  - Asynchronous I/O
  - Scalability with many input/output operations
  - Real-time web applications
    - e.g., real-time communication programs, browser games and data streaming, etc.
- Node.js functions are non-blocking
  - Commands execute concurrently or even in parallel (unlike PHP that commands execute only after previous commands finish)
    - Node.js uses callbacks to signal completion or failure

Event-driven programming – When to use Node.js?

- Creation of Web servers and networking tools
  - Ideal for applications that serve a lot of requests but don't use/need lots of computational power per request
- Using JavaScript and a collection of **modules** that handle various core functionality such as:
  - File system I/O, networking (DNS, HTTP, TCP, TLS/SSL, or UDP), binary data (buffers), cryptography functions, data streams, etc.
  - Modules use an API (**interfaces**) designed to **reduce the complexity** of writing server applications

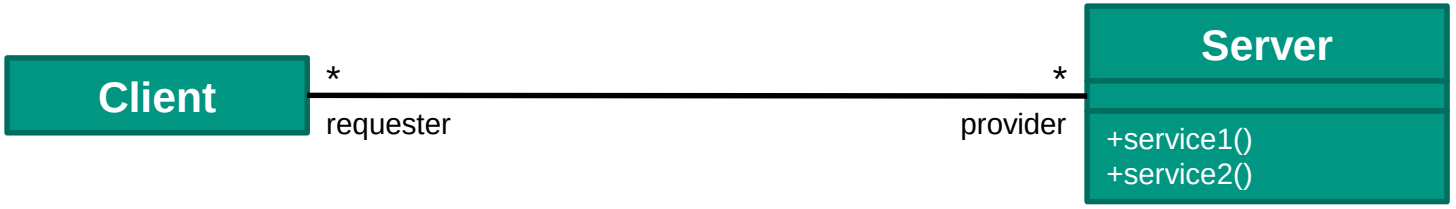
Architectural styles: n-tier

- Architectural styles (**n-tier**, client/server, …) may be applied by the design and implementation of applications and systems
  - Transmit events among loosely coupled software components and services
- **n-tier architecture** (layered architecture, see section Architectural styles)
  - Example**: 4-tier**:



2-tier architecture – Client/Server

- One or more servers provide services for other subsystems called clients.
- Each client invokes a function of the server which performs the desired service and returns the result.
  - **The client must know the interface of the server**!
  - Conversely, the **server does not need to know the client's interface**.
- An example of a 2-tier, distributed architecture:



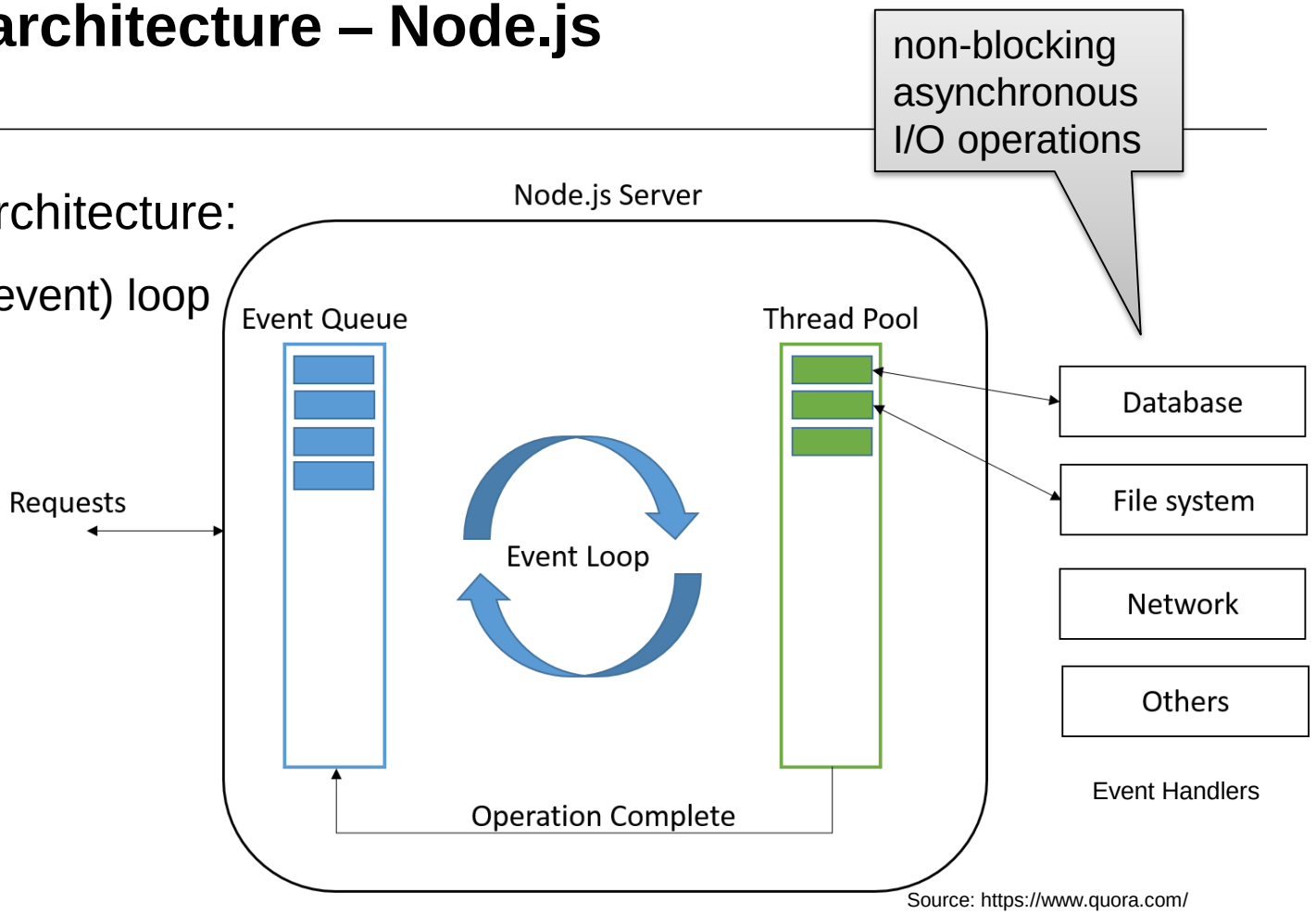
- Event-driven architecture: A **single thread** (server), of the **event loop** processes all the requests from clients (**event queue**)

Event-driven architecture

- Event-driven architecture:
  - **Processing loop (Event loop)**
  - **Event queue**
  - **Call-back**

Event-driven architecture – Node.js

- Event-driven architecture:
  - Processing (event) loop
  - Event queue
  - Call-back

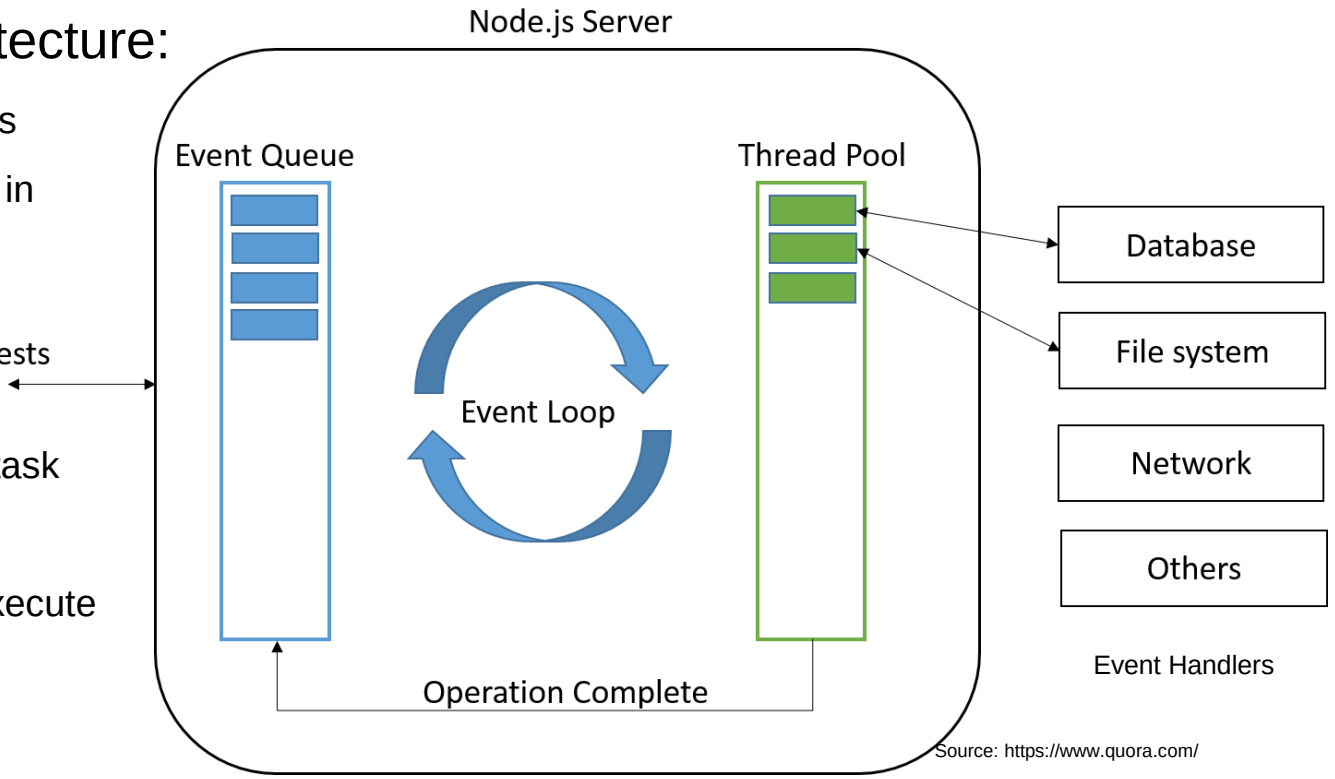


Source: https://www.quora.com/

- Node.js Architecture: The event loop simply iterates over the event queue (a list of events) and callbacks of completed operations.

Event-driven architecture – Node.js (2)

- Event-driven architecture:
  - A **thread pool** handles execution of parallel tasks in Node.js
  - The **main thread (event loop)** posts tasks to the shared task queue, where threads in the thread pool pull and execute



Source: https://www.quora.com/

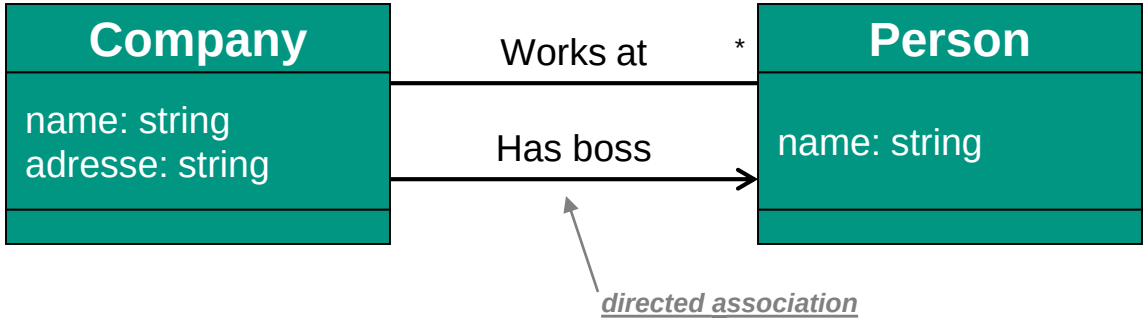
- A task goes to the file system, the system is ready for next requests. When a file is opened and read, the system sends the content to the client.
- Inherently blocking system functions such as file I/O run in a blocking way on their own threads

Event-driven architecture – Service-Oriented Architecture (SOA)

- Event-driven architecture can complement **service-oriented architecture (SOA)**
- Services can be activated by triggers fired on incoming events
  - **SOA is an architecture style** that assembles applications from (independent) services (see later section Architectural styles)
  - Services are considered as central elements of a company (keyword: services)
  - Provide encapsulated functionality to other services and applications
- ➔ **Observer design pattern** is implemented in event-driven architecture

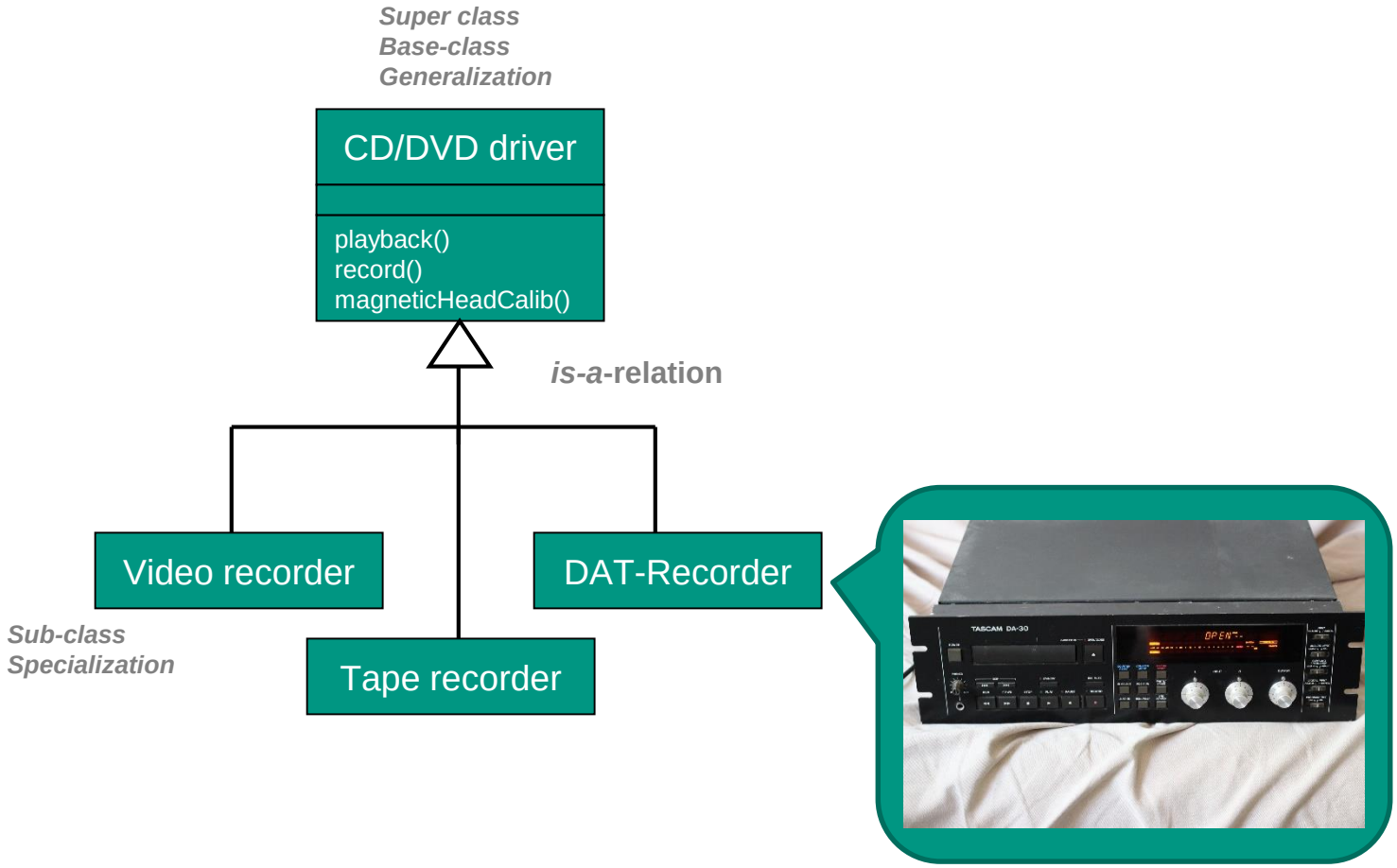
Class diagram – Example

- Describes the types of objects in the system
- Describes the static relationships among them



- Note: Class diagrams are multigraphs, i.e. several edges can consist between identical nodes.

Inheritance – Example



Design pattern definition

- Design pattern definition:

> A software design pattern describes a family of solutions to a software design problem

- The purpose of design patterns is the **reuse** of design knowledge.
- Design patterns are for designing (or programming in the large) similar to algorithms that are for programming in the small.

What are design patterns?

- **Design Patterns:**
  - Software design patterns are borrowed from the concept of design patterns in architecture and urban planning.

  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
  - Source: Christopher Alexander et al: A Pattern Language, Oxford University Press, New York, 1977
  - Christopher Alexander is an architect, architectural theorist and systems theorist.

Why design patterns?

- Patterns improve **communication** in the team
  - Design patterns provide useful terminology, i.e. they provide terms and shorthand for discussion between developers about complex concepts.
- Patterns capture essential **concepts** and bring them into an understandable form
  - Patterns help to understand designs
  - Patterns document designs in a nutshell
  - Patterns prevent architecture-drift (i.e. degradation of original architecture in case of changes)
  - Patterns clarify design knowledge

Why design patterns? (2)

- Patterns **document** and **promote** the state of the art
  - Patterns help less experienced designers
  - Patterns avoid the reinvention of the wheel
  - A pattern is not a fixed rule to follow blindly, but a suggestion and set of alternatives to solve a problem. **Adaptation required**!
- Patterns can **improve** code quality and code structure
  - Patterns promote good design and good code by providing constructive examples

Observer design pattern in event-driven architecture

- A **single thread**, using non-blocking I/O calls

➔ **Observer** design pattern**:**

  - Sharing a single thread among all the requests
  - Defines a **1-to-n** dependency between objects so that changing a state of an object causes all dependent objects to be **notified** and **updated automatically**
  - One to many relationship
  - The many need to know changes in “one” immediately

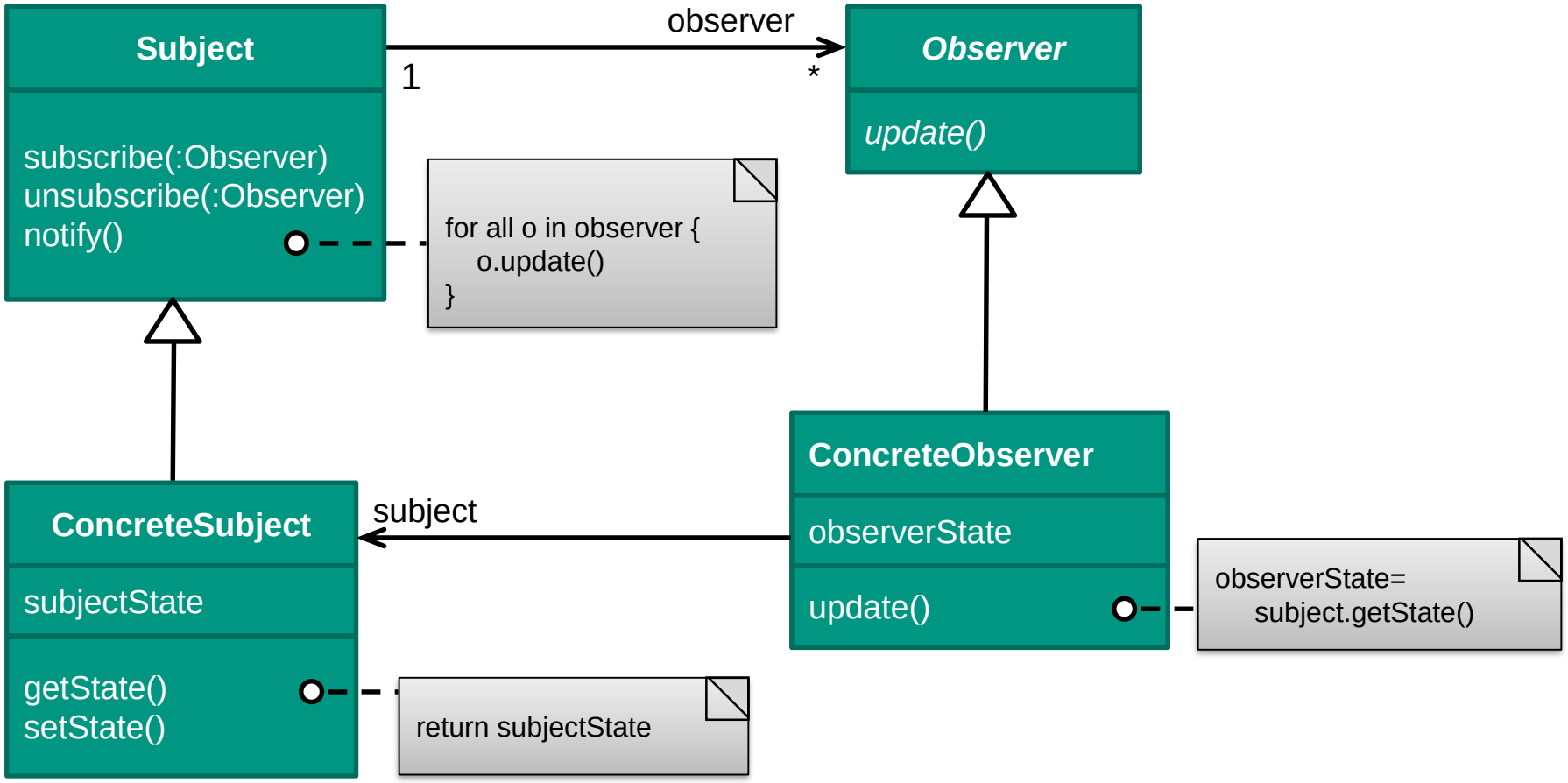
- Synonyms (aka)
  - Dependence, Publisher-subscriber, Subject-observer

Observer pattern: Motivation

- If one splits a system into a set of interacting classes, the **consistency** between the interrelated objects must be maintained.
- **Strong coupling** of these classes is **not recommended** because it limits individual reusability.
- If an object changes its status, how to let all its “subscribers” know? What if there are different types of subscribers?

Observer: Structure

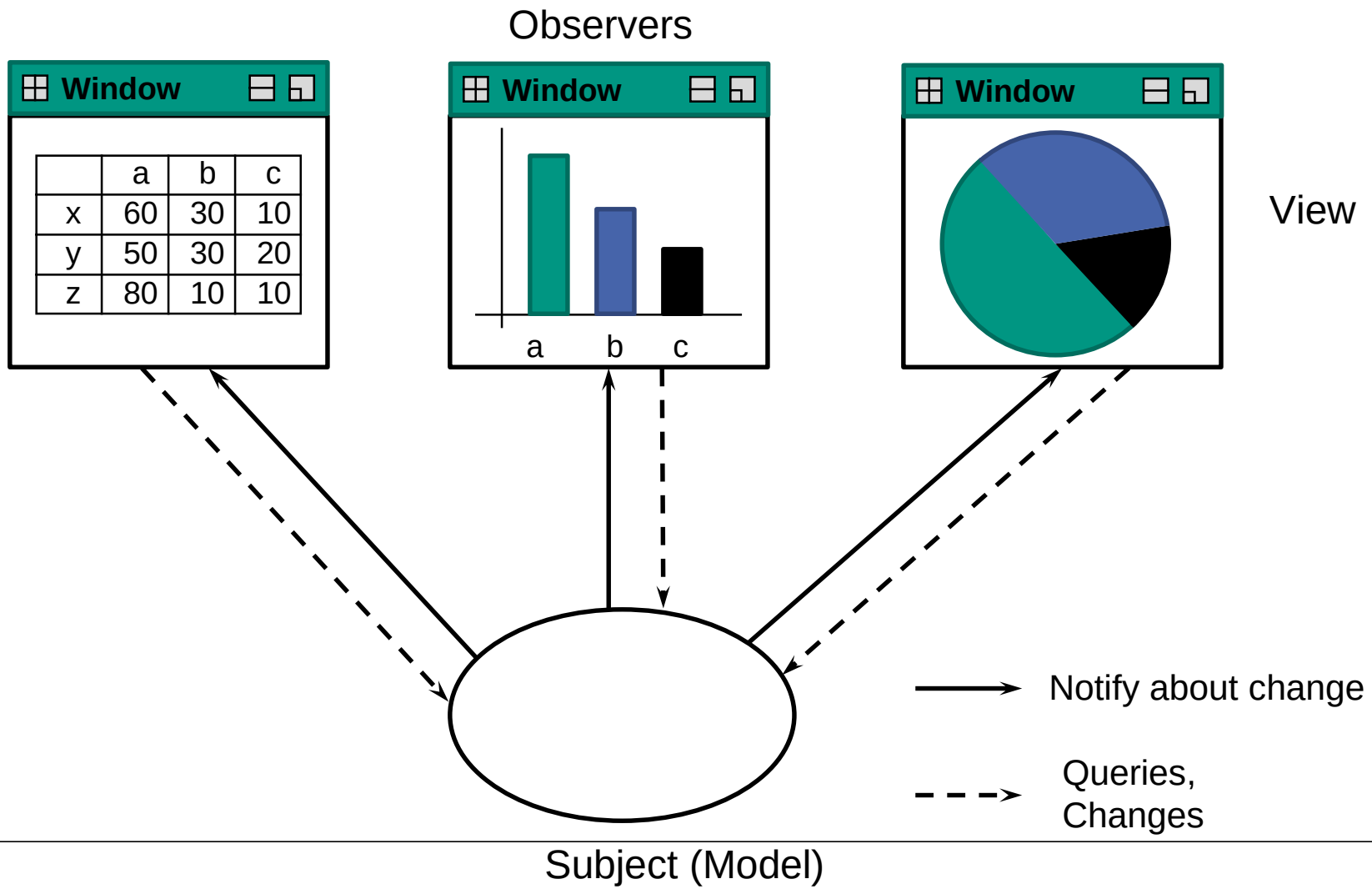
- Observer pattern helps to understand event-driven architecture concepts:



Observer in event-driven architecture

- In observer pattern an object, called the **subject**, maintains a list of its dependents, called **observers**, and notifies them of any state changes
- Subjects and observers can be reused **independently**
- Observers can be **added** or **removed without changing** the subject or other observers
- Making use of events and observers makes **services (objects)** even more **loosely coupled**

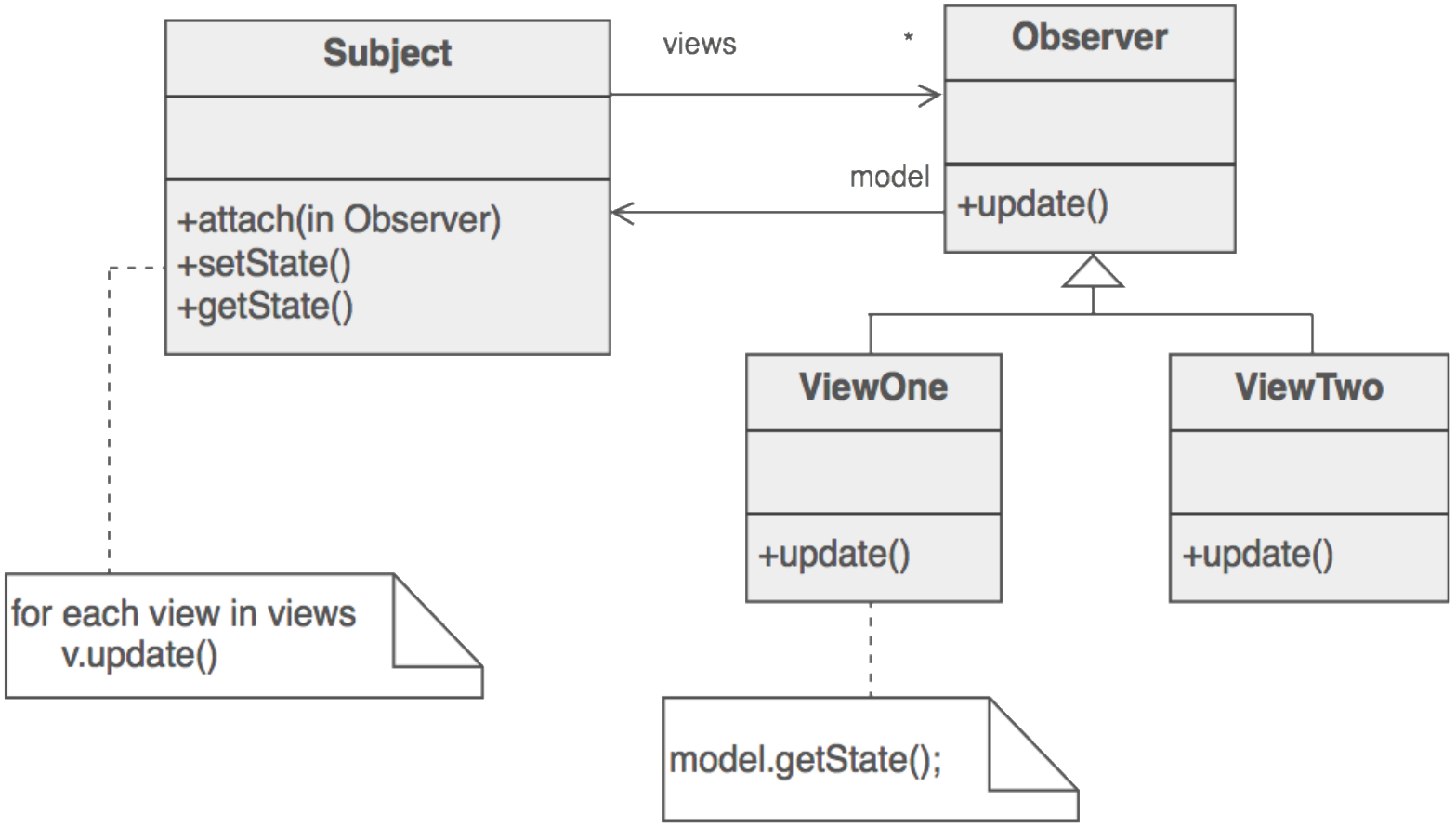
Observer: Example – MVC



Observer: Example – MVC (2)

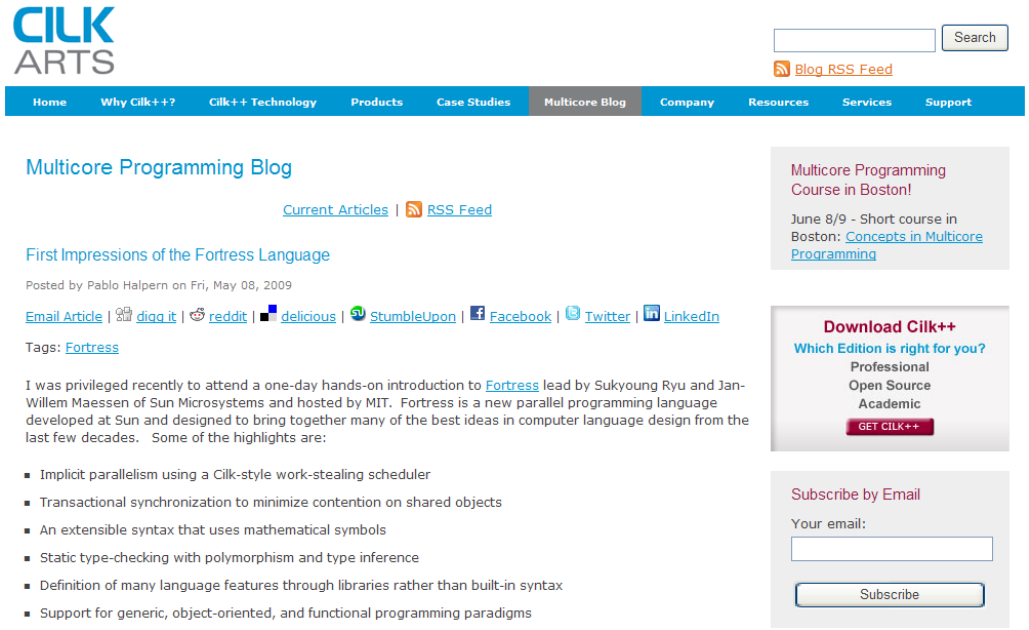
- In the MVC example, the table view and the column view know nothing about each other.
- So they can be reused independently. Nevertheless, both objects behave as if they knew each other.

Observer: Example – MVC (3)



Observer: Example – Blog subscribers

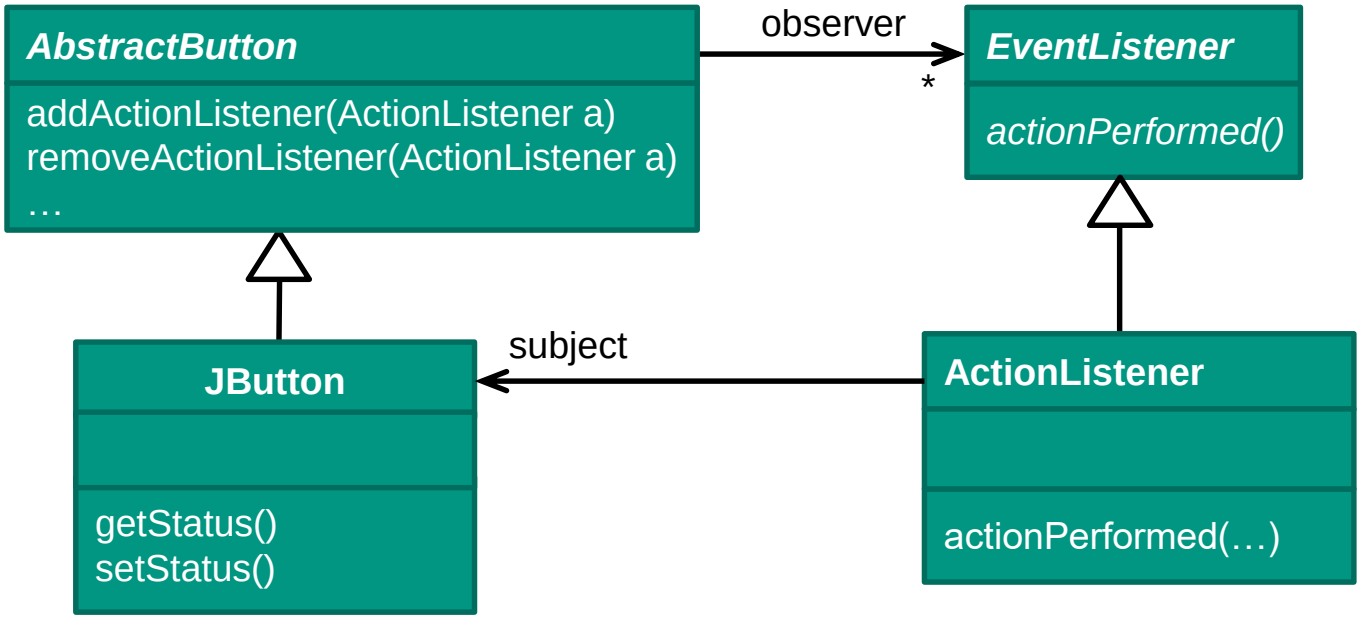
- News feed, e.g. Facebook feed, …
- A blog is created and updated by a blogger.
- If a visitor of the website reads the blog and finds it interesting, she or he can subscribe to the blog and will be informed immediately about new entries.



http://www.cilk.com/multicore-blog/

Observer: Example in Java

- In Java, for example, **events** are handled with observers:



Observer: Applicability

- If changing an object requires altering other objects and you do not know how many and which objects need to be changed.
- When an object needs to notify other objects without making assumptions about those objects.
- If an abstraction has two aspects, one depends on the other. Encapsulating these aspects in separate objects allows for independent reusability.

➔ **Observer design pattern** used in **event-driven architecture!**

What can you do with Node.js? – An event-driven architecture

- Node.js file contains tasks and executes them upon set of events
  - Generate dynamic content (even desktop programs in js)
  - Create, open and read, or delete files on the server
  - Gather and modify data in the database
  - Collect form data, etc.
- Availability of rich frameworks
  - **Angular**, **React**, Node, Backbone, Ember, etc.
- Ability to keep data in native JSON (JavaScript Object Notation, similar to XML) format in your database
- Very good supportive community
  - Linux Foundation, Google, PayPal, Microsoft, …

JSON (JavaScript Object Notation)

- Syntax for storing and exchanging data (text written with JavaScript object notation)
  - Convert any JavaScript object into JSON, and send JSON to the server
  - Convert any JSON received from the server into JavaScript objects
- Work with the data as JavaScript objects (no complicated parsing and translations)
  - Example:

```
myObj = { "name":"John", "age":30, "car":null };
x = myObj.name;
```

key:value ➔ keys are strings; values are valid JSON data type (string, number, object, array, boolean or null).

Event handling – Event emitter pattern

```
// Instead of only completed event, many events may be fired.
// Handlers can be registered for each event.
var fs = require('fs');
var file = fs.createReadStream('./' + process.argv[2]);

file.on('error', function(err) {
  console.log("Error:" + err);
  throw err;
});

file.on('data', function(data) {
  console.log("Data: " + data);
});
file.on ('end', function() {
  console.log("finished reading all of data");
});
```

readStream object fires events when opening and closing a file

createReadStream fires **error**, **data**, and **end** events

Using **on** function, we attach **event handlers**.

Event emitter API

- **Event types (determined by emitter)**
  - error (special type)
  - data
  - end
- **API**
  - `.on` or `.addListener`
  - `.once` (will be called at most once)
  - `.removeEventListener`
  - `.removeAllEventListeners`

Creating an event emitter – Example

```
// file named myEmitter.js
var util = require('util'); // step 1
var eventEmitter = require('events').EventEmitter; // step 2
var Ticker = function() {
  var self = this;
  setInterval (function() {
      self.emit('tick'); // step 3
    }, 1000) ;
};
util.inherits (Ticker, eventEmitter); // step 4
module.exports = Ticker;
//Note: Use “class Ticker extends EventEmiiter” instead of util.inherits which is deprecated
```

Util module provides access to some utility functions.

With ”**events**” you can create-, fire-, and listen for- your own events.

can be used to execute the code multiple times

Inherits methods from one function into another

Creating an event emitter – Example (using Ticker)

```
// testing Ticker
var Ticker = require("./myEmitter");

var ticker = new Ticker();

ticker.on ('tick', function() { // handler for 'tick' event
   console.log("Tick");
});
```

Servers

Simple servers

require('net')

createServer()
listen(port#)
'error'
'connection'
'data'
'close'

HTTP servers

require('http')

createServer()
listen(port#)
'request'
req.on 'data'

- net module provides an asynchronous network API for creating stream-based TCP servers

Servers – Example

- Get the data by listening to the stream data events
- When the data ends the stream end event is called:

```
const server = http.createServer((req, res) => {
  // we can access HTTP headers
  req.on('data', chunk => {
    console.log(`Data chunk available: ${chunk}`)
  });
  req.on('end', () => {
    //end of data
  })
})
```

Source: https://nodejs.org/en/docs/guides/getting-started-guide/

Database server – Node.js MySQL (1)

- Module to manipulate the MySQL database

```
var mysql = require('mysql');
```

- Creating a connection to the database

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
});
```

Database server – Node.js MySQL (2)

- Create a database named "mydb"

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  con.query("CREATE DATABASE
mydb", function (err, result) {
    if (err) throw err;
    console.log("Database
created");
  });
});
```

Database server – Node.js MySQL (3)

- Create a table in “mydb” database
- Use the "CREATE TABLE" statement

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "CREATE TABLE
customers (name VARCHAR(255),
address VARCHAR(255))";
  con.query(sql, function (err,
result) {
    if (err) throw err;
    console.log("Table created");
  });
});
```

Database server – Node.js MySQL (4)

- Create primary key when creating the table
- A column with a unique key for each record

```
var sql = "CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255), address VARCHAR(255))";
```

- A column as "INT AUTO_INCREMENT PRIMARY KEY" which will insert a unique number for each record.
  - Starting at 1, and increased by one for each record.

Database server – Node.js MySQL (5)

- Insert a record in the table

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "INSERT INTO customers (name, address) VALUES ('Company Inc', 'Highway 37')";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("1 record inserted");
  });
});
```

Database server – Node.js MySQL (6)

- Insert a record in the table – results

```
C:\Users\My Name>node demo_db_select.js
[
  { id: 1, name: 'John', address: 'Highway 71'},
  { id: 2, name: 'Peter', address: 'Lowstreet 4'},
  { id: 3, name: 'Amy', address: 'Apple st 652'},
  { id: 4, name: 'Hannah', address: 'Mountain 21'},
  { id: 5, name: 'Michael', address: 'Valley 345'},
  { id: 6, name: 'Sandy', address: 'Ocean blvd 2'},
  { id: 7, name: 'Betty', address: 'Green Grass 1'},
  { id: 8, name: 'Richard', address: 'Sky st 331'},
  { id: 9, name: 'Susan', address: 'One way 98'},
  { id: 10, name: 'Vicky', address: 'Yellow Garden 2'},
  { id: 11, name: 'Ben', address: 'Park Lane 38'},
  { id: 12, name: 'William', address: 'Central st 954'},
  { id: 13, name: 'Chuck', address: 'Main Road 989'},
  { id: 14, name: 'Viola', address: 'Sideway 1633'}
]
```

Database server – Node.js MySQL (7)

- Important SQL Commands:
  - **SELECT** - extracts data from a database
  - **UPDATE** - updates data in a database
  - **DELETE** - deletes data from a database
  - **INSERT INTO** - inserts new data into a database
  - **CREATE DATABASE** - creates a new database
  - **ALTER DATABASE** - modifies a database
  - **CREATE TABLE** - creates a new table
  - **ALTER TABLE** - modifies a table
  - **DROP TABLE** - deletes a table
  - **CREATE INDEX** - creates an index (search key)
  - **DROP INDEX** - deletes an index

Threading – Node.js

- A **single thread**, using non-blocking I/O calls
  - Support tens of thousands of concurrent connections **without** the **cost of thread context switching**
  - Building highly concurrent applications
  - A thread pool (event handlers) handles execution of parallel tasks
- Good for **horizontal scaling** (lots of requests)!
  - Highly scalable servers without using threading, by using a simplified model of event-driven architecture/programming that uses **callbacks** to signal the completion of a task

Threading – Node.js (2)

- **Drawback** of the single-threaded approach
  - **No vertical scaling** by increasing the number of CPU cores of the system
  - Not good for massive parallel computing
  - Needs additional module
    - Such as cluster, StrongLoop Process Manager, Pm2, etc.
- **Mitigation**:
  - Developers can increase the default number of threads in the thread pool
    ➔ The server OS distributes the threads across multiple cores (event handlers running on multiple cores).

Thread-based vs. Event-based

Threads (Java Threads)	Asynchronous event-driven (Node.js)
Monitor/synchronization (difficult to program, race conditions)	event handler (using queue and then processes it)
scheduling (ready, running, waiting, …)	event loop (only one thread, which repeatedly fetches an event)
exported functions (need to be thread-safe with no data race)	event types accepted by event handler
returning from a procedure (using context switching)	dispatching a reply (no contention and no context switches)
executing a blocking procedure call	dispatching a message, awaiting a reply
waiting on condition variables (synchronization)	awaiting messages

Conclusion:

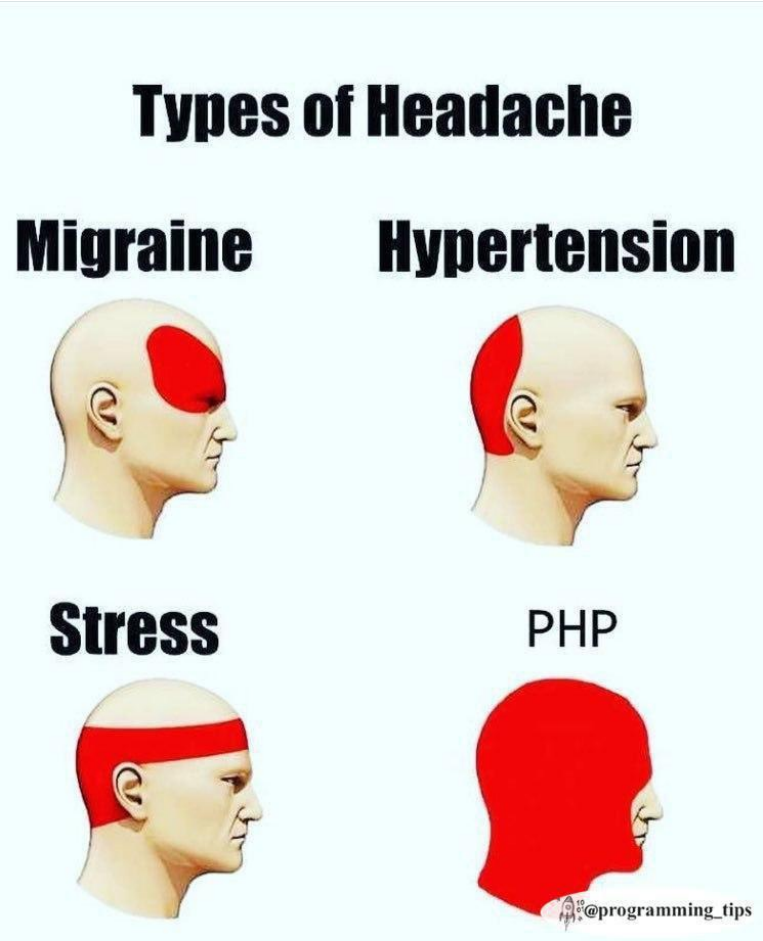
- Use threads for performance critical applications (kernels, compute-intensive)
- Use events for GUI and distributed systems (lots of requests, horizontal scaling)

Use Node.js for …

- Chat/Messaging
- Real-time Applications
- Intelligent Proxies
- High Concurrency Applications
- Communication Hubs
- Coordinators
- ….

- Web application
- Websocket server
- Ad server
- Proxy server
- Streaming server
- Fast file upload client
- Any Real-time data apps
- Anything with high I/O
- ….

Why Node.js? – Humor



Use
Node.js !!!

Literature – Node.js

- https://nodejs.org/en/
- https://www.w3schools.com/nodejs/default.asp
- https://www.tutorialspoint.com/nodejs/index.htm
- https://npmjs.org/