

# Principles of Software Construction: Objects, Design, and Concurrency

## Basic GUI concepts, HTML

Christian Kästner

Vincent Hellendoorn



# HW3 Reflections

# Today

- GUI Design
  - Concepts, strategies
  - Practical application in HTML, CSS, JS
- Dynamic Web Pages
  - Client/Server communication
  - Backend architecture

# How To Make This Happen?

17-214 Fall 2021

Course calendarScheduleSyllabusPiazza

- Have experience testing and analyzing your software
- Understand principles of concurrency and distributed systems

See a more detailed list of [learning goals](#) describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.

**Coordinates**

Tu/Th 11:50 - 1:10 p.m. in DH 2315

**Christian Kaestner**, [kaestner@cs.cmu.edu](mailto:kaestner@cs.cmu.edu), TCS 345, office hours Friday 11:30-1pm (see calendar)

**Vincent Hellendoorn**, TCS 320, office hours Tuesdays 9am-11am (see calendar)

Our TAs also provide an additional 18h of office hours each week, usually in TCS 310, see details in the calendar.

The instructors have an open door policy: If the instructors' office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.

**Course Calendar**

17214 F21

Today

October 2021

PrintWeekMonthAgenda

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26 14:00 Ye OH (Online)	27 15:00 Kevin OH	28 09:00 Vincent OH 11:50 Lecture	29 09:05 Recitation A (J) 10:10 Recitation B (T) 11:15 Recitation C (J) <a href="#">+4 more</a>	30 11:50 Midterm 14:00 Canceled: Oliv 17:00 Canceled: Kati	1 Oct 11:30 Christian OH 16:00 Zhifeng OH	2 10:00 Esther OH
3 14:00 Ye OH (Online)	4 13:30 Christian OH 15:00 Kevin OH	5 09:00 Vincent OH 11:50 Lecture	6 09:05 Recitation A (J) 10:10 Recitation B (T) 11:15 Recitation C (J) <a href="#">+4 more</a>	7 10:00 Sophie OH 14:00 Olivia OH 17:00 Katrina OH	8 16:00 Zhifeng OH	9 10:00 Esther OH
10 14:00 Ye OH (Online)	11 13:30 Christian OH 15:00 Kevin OH	12 09:00 Vincent OH 11:50 Lecture	13 09:05 Recitation A (J) 10:10 Recitation B (T) 11:15 Recitation C (J) <a href="#">+4 more</a>	14 <b>Midsemester Break</b> 10:00 Sophie OH 14:00 Olivia OH 17:00 Katrina OH	15 16:00 Zhifeng OH	16 10:00 Esther OH
17	18	19	20	21	22	23

# Why not plaintext?

```
17-214 Fall 2021 Untitled-1 •
1 17-214 Fall 2021
2 Principles of Software Construction
3 Objects, Design, and Concurrency
4 Overview
5 Software engineers today are less likely to design data structures and algorithms from scratch and more likely to build systems from library and framework components. In this course, students engage
  with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer
  structures. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object oriented programming, (3) static and dynamic analysis for programs, and (4)
  concurrent and distributed software. Student assignments involve engagement with complex software such as distributed massively multi-player game systems and frameworks for graphical user
  interaction.
6
7 Update for Fall 2021: We are planning several changes to the course for the fall 2021 semester. A key change is that we will teach the course with multiple programming languages. We will cover
  multiple languages in the lecture, but will expect students to focus on one language in assignments. When signing up, please chose a section for Java or JavaScript/TypeScript.
8
9 After completing this course, students will:
10
11 Be comfortable with object-oriented concepts and with programming in the Java or JavaScript language
12 Have experience designing medium-scale systems with patterns
13 Have experience testing and analyzing your software
14 Understand principles of concurrency and distributed systems
15 See a more detailed list of learning goals describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through
  assignments and exams.
16
17 Coordinates
18 Tu/Th 11:50 - 1:10 p.m. in DH 2315
19
20 Christian Kaestner, kaestner@cs.cmu.edu, TCS 345, office hours Friday 11:30-1pm (see calendar)
21
22 Vincent Hellendoorn, TCS 320, office hours Tuesdays 9am-11am (see calendar)
23
24 Our TAs also provide an additional 18h of office hours each week, usually in TCS 310, see details in the calendar.
25
26 The instructors have an open door policy: If the instructors' office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email
  us for appointments.
27
28 Course Calendar
29
30 Schedule
31 We are planning significant changes to the course this semester. The schedule below is a rough draft of our plans, but likely to change.
32
33 TUE, AUG 31
34 Intro
35 WED, SEP 1
36 rec 1 Introduction to Git
37 THU, SEP 2
38 OO basics, Dynamic dispatch, Encapsulation
39 TUE, SEP 7
40 IDEs, Build system, Continuous Integration, Libraries
41 Required: Effective Java, Items 15 and 16
42 WED, SEP 8
43 rec 2 IDEs, Build systems, Libraries, CI
44 THU, SEP 9
45 Specifications and unit testing, exceptions
```

# Why not a Doc?

AutoSave Off 17.asd: 10/11/2021 9:49 AM (Unsaved File) - Read-Only Vincent Hellendoorn

File Home Insert Draw Design Layout References Mailings Review View Help

Clipboard Font Paragraph Styles Editing Voice Editor Reuse Files

RECOVERED UNSAVED FILE This is a recovered file that is temporarily stored on your computer. Save As

Navigation

Search document

Headings Pages Results

- Principles of Software Construction Objects, Design,...
- Overview
  - Coordinates
  - Course Calendar
- Schedule
- Course Syllabus and Policies
  - Prerequisites
  - Grading
  - Attendance and remote participation
  - Textbooks
  - Time management
  - Late work policy
  - Collaboration policy
  - Accommodations
  - Your health matters
  - Informal feedback on this course

Have experience designing medium-scale systems with patterns

Have experience testing and analyzing your software

Understand principles of concurrency and distributed systems

See a more detailed list of [learning goals](#) describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.

**Coordinates**

Tu/Th 11:50 - 1:10 p.m. in DH 2315

[Christian Kaestner](#), kaestner@cs.cmu.edu, TCS 345, office hours Friday 11:30-1pm (see calendar)

[Vincent Hellendoorn](#), TCS 320, office hours Tuesdays 9am-11am (see calendar)

Our TAs also provide an additional 18h of office hours each week, usually in TCS 310, see details in the calendar.

The instructors have an [open door](#) policy: If the instructors' office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.

**Course Calendar**

7214 F21

Today October 2021

Print Week Month Agenda

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	1 Oct	2

14:00 Ye OH 15:00 Kevin OH 16:00 Vincent OH 17:00 Recitation A (Java) 18:00 Midterm 19:00 Christian OH 20:00 Esther OH

11:30 Lecture (TypeScript) 12:00 Canceled: Olivia OH 13:00 Canceled: Olivia OH 14:00 Canceled: Olivia OH 15:00 Canceled: Olivia OH 16:00 Canceled: Olivia OH 17:00 Canceled: Olivia OH 18:00 Canceled: Olivia OH 19:00 Canceled: Olivia OH 20:00 Canceled: Olivia OH

11:15 Recitation C 12:00 Canceled: Katrina OH

Page 2 of 21 3693 words

# GUI Design: what do we want?

- Nested Elements
- Style Vocabulary
- Interactivity

# GUI Design: what do we want?

- Nested Elements
  - HTML
- Style Vocabulary
  - CSS
- Interactivity
  - JavaScript



# Anatomy of an HTML Page

## Predefined elements

The diagram illustrates the structure of an HTML document. Three boxes labeled 'Root\*', 'Header', and 'Body' are shown on the left. Arrows point from these boxes to the corresponding elements in the HTML code on the right. 'Root\*' points to the root of the document tree, 'Header' points to the <head> element, and 'Body' points to the <body> element. The code on the right is a snippet of HTML showing the document structure.

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body> == $0
    <nav id="navigation" class="hidden">...</nav>
    <header id="top" class="container">...</header>
    <div id="main" class="container">...</div>
  </body>
</html>
```

Overview

Software engineers to... data structures... systems from... library and framework... components in the course, students engage with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer structures. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object oriented programming, (3) static and dynamic analysis for programs, and (4) concurrent and distributed software. Student assignments involve engagement with complex software such as distributed massively multi-player game systems and frameworks for graphical user interaction.

Update for Fall 2021: We are planning several changes to the course for the fall 2021 semester. A key change is that we will teach the course with multiple programming languages. We will cover multiple languages in the lecture, but will expect students to focus on one language in assignments. When signing up, please chose a section for Java or JavaScript/TypeScript.

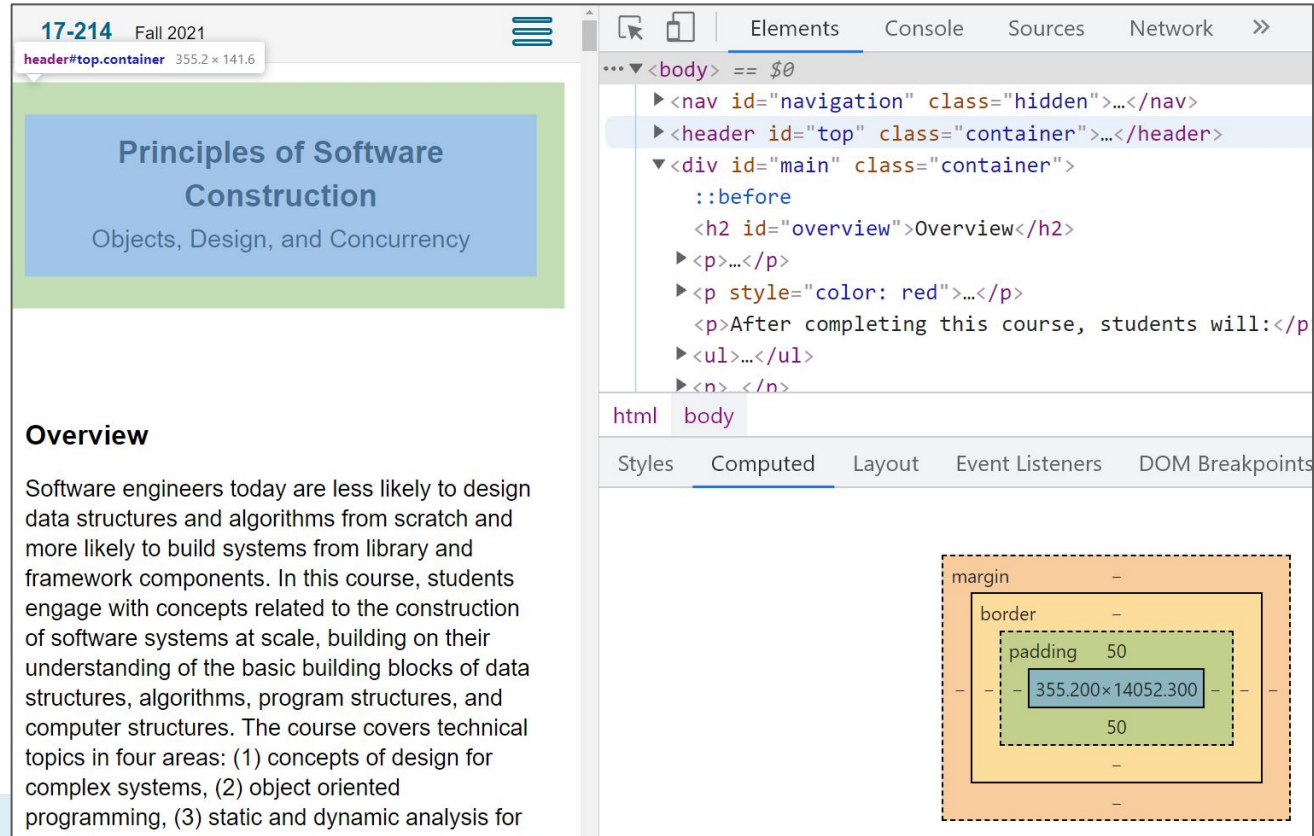
After completing this course, students will:

- Be comfortable with object-oriented concepts and with programming in the Java or JavaScript language
- Have experience designing medium-scale systems with

# Anatomy of an HTML Page

## Nested elements

- Sizing
- Attributes
- Text



17-214 Fall 2021

header#top.container 355.2 x 141.6

**Principles of Software Construction**  
Objects, Design, and Concurrency

**Overview**

Software engineers today are less likely to design data structures and algorithms from scratch and more likely to build systems from library and framework components. In this course, students engage with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer structures. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object oriented programming, (3) static and dynamic analysis for

```
... <body> == $0
  ▶ <nav id="navigation" class="hidden">...</nav>
  ▶ <header id="top" class="container">...</header>
  ▼ <div id="main" class="container">
    ::before
    <h2 id="overview">Overview</h2>
    ▶ <p>...</p>
    ▶ <p style="color: red">...</p>
    <p>After completing this course, students will:</p>
    ▶ <ul>...</ul>
    ▶ <n> </n>
```

html body

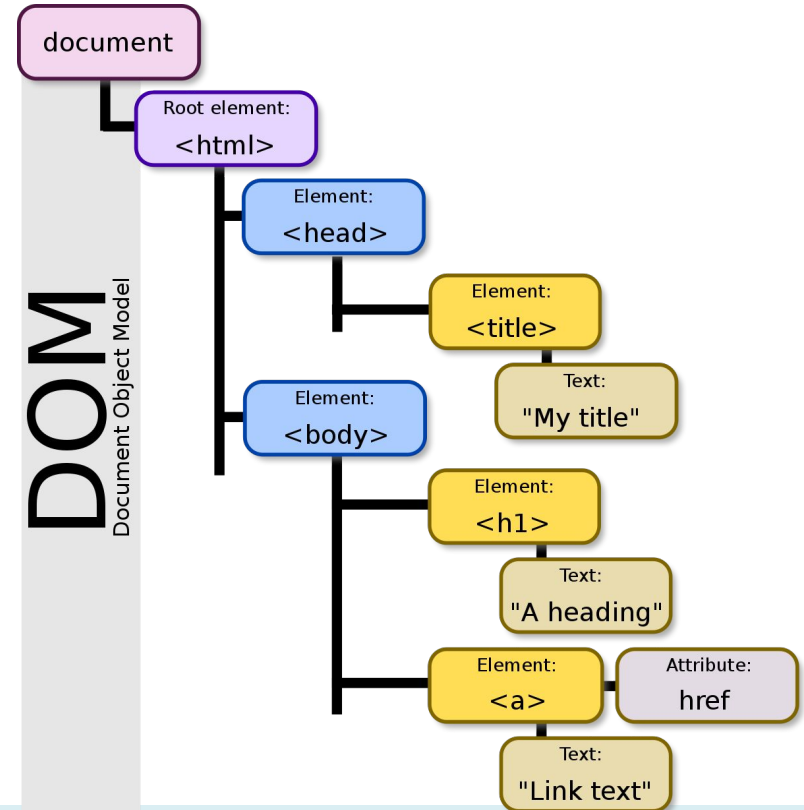
Styles Computed Layout Event Listeners DOM Breakpoints

margin -  
border -  
padding 50  
355.200 x 14052.300  
50

# Anatomy of an HTML Page

Many GUIs are trees

- Nested elements, recursively
- Some fixed positions (html, body)

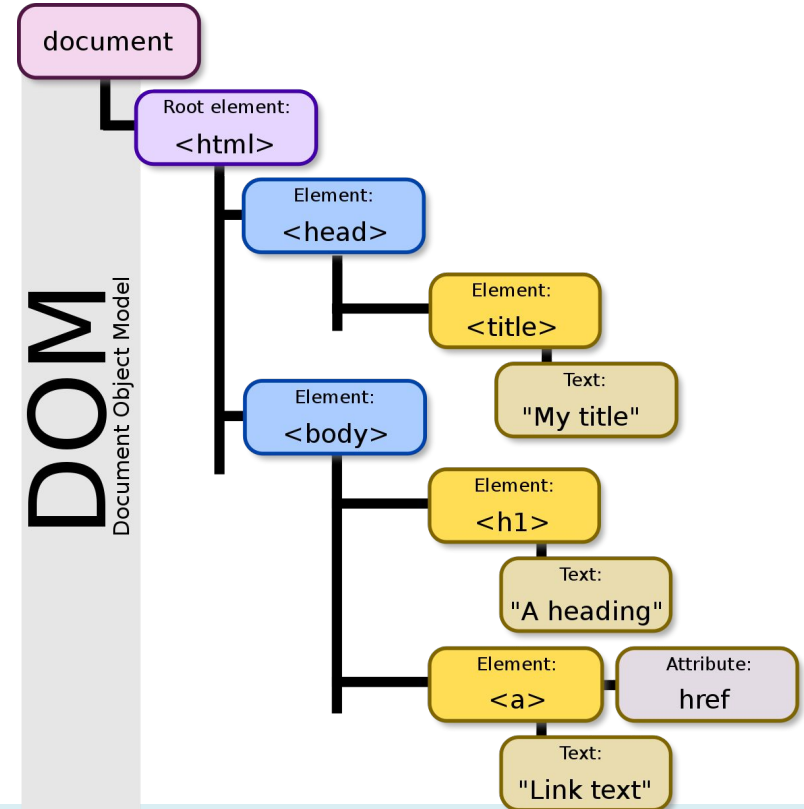


# Anatomy of an HTML Page

Many GUIs are trees

- Nested elements, recursively
- Some fixed positions (html, body)

How to implement this?

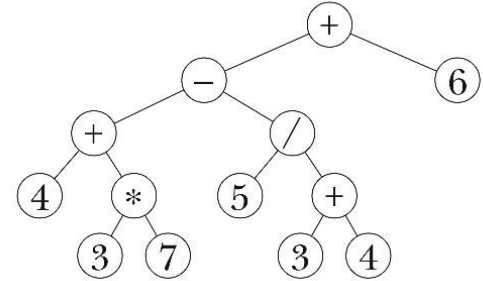


# The composite pattern

- Problem: Collection of objects has behavior similar to the individual objects
- Solution: Have collection of objects and individual objects implement the same interface
- Consequences:
  - Client code can treat collection as if it were an individual object
  - Easier to add new object types
  - Design might become too general, interface insufficiently useful

# Another composite pattern example

```
public interface Expression {  
    double eval();    // Returns value  
}
```



```
public class BinaryOperationExpression implements Expression {  
    public BinaryOperationExpression(BinaryOperator operator,  
        Expression operand1, Expression operand2);  
}
```

```
public class NumberExpression implements Expression {  
    public NumberExpression(double number);  
}
```

# Composite

- Elements can contain elements
  - With restrictions
  - Need to deal with style, interaction
- In JS: HTMLElement
  - With child-classes e.g. HTMLDivElement, HTMLBodyElement
  - Navigation:
    - getElement\*: locate by tag name, id, class, etc.
    - next/prev(Element)Sibling
    - childNodes, parent

# A few Tags

- `<html>`
  - The root of the visible page
- `<head>`
  - Stores metadata, imports
- `<p>`
  - A paragraph
- `<button>`
  - Attributes include `name`, `type`, `value`
- `<div>`
  - Generic section -- very useful
- `<table>`
  - The obvious
- Many more; dig into a real page!



# Style

Not only leaf-nodes have an appearance

The screenshot displays a web application titled "Course Calendar" for "17214 F21". It shows a calendar for "October 2021" with a table of events. A tooltip indicates a table row with dimensions "784 x 18". To the right, a browser's developer tools window shows the "Elements" panel with the following HTML structure:

```
<div class="month-row" style="top:16.666666666666668%;height:17.666666666666668%">...</div>
<div class="month-row" style="top:33.333333333333336%;height:17.666666666666668%">
  <table cellpadding="0" cellspacing="0" class="st-bg-table">...</table>
  <table cellpadding="0" cellspacing="0" class="st-grid">
    <tbody>
      <tr>
        <td class="st-dtitle st-dtitle-fc">...</td>
        ...
        <td class="st-dtitle st-dtitle-today">...</td> == $0
        <td class="st-dtitle st-dtitle-next">...</td>
        <td class="st-dtitle">...</td>
        <td class="st-dtitle">...</td>
        <td class="st-dtitle">...</td>
        <td class="st-dtitle">...</td>
      </tr>
      <tr>...</tr>
      <tr>...</tr>
      <tr>...</tr>
    </tbody>
  </table>

```

The browser's breadcrumb at the bottom shows the path: `... 2.mv-event-container div.month-row table.st-grid tbody tr td.st-dtitle.st-dtitle-today ...`.

# Style

Tags come with inherent & customizable style

- Inherent:
  - `<div>` is a `block` (full-width, with margin)
  - `<span>` is in-line
  - `<h1>` is large
- Customizable: add and override styles
  - Change font-styles, margins, widths
  - Modify groups of elements

# Style: CSS

- Cascading Style Sheets
  - Reuse: styling rules for tags, classes, types
  - Reuse: not just at the leafs!

```
<span style="font-weight:bold">Hello again!</span>
```

VS.

```
<style type="text/css">
  span {
    font-family: arial
  }
</style>
```

# Style: CSS

- Cascading Style Sheets
  - Reuse: styling rules for tags, classes, types
  - Reuse: not just at the leafs!
- What if there are conflicts?

```
<div style="font-weight:normal">  
  <span style="font-weight:bold">Hello again!</span>  
</div>
```

- Lowest element wins\*

\*Technically, there's a whole scoring system

# Style: CSS

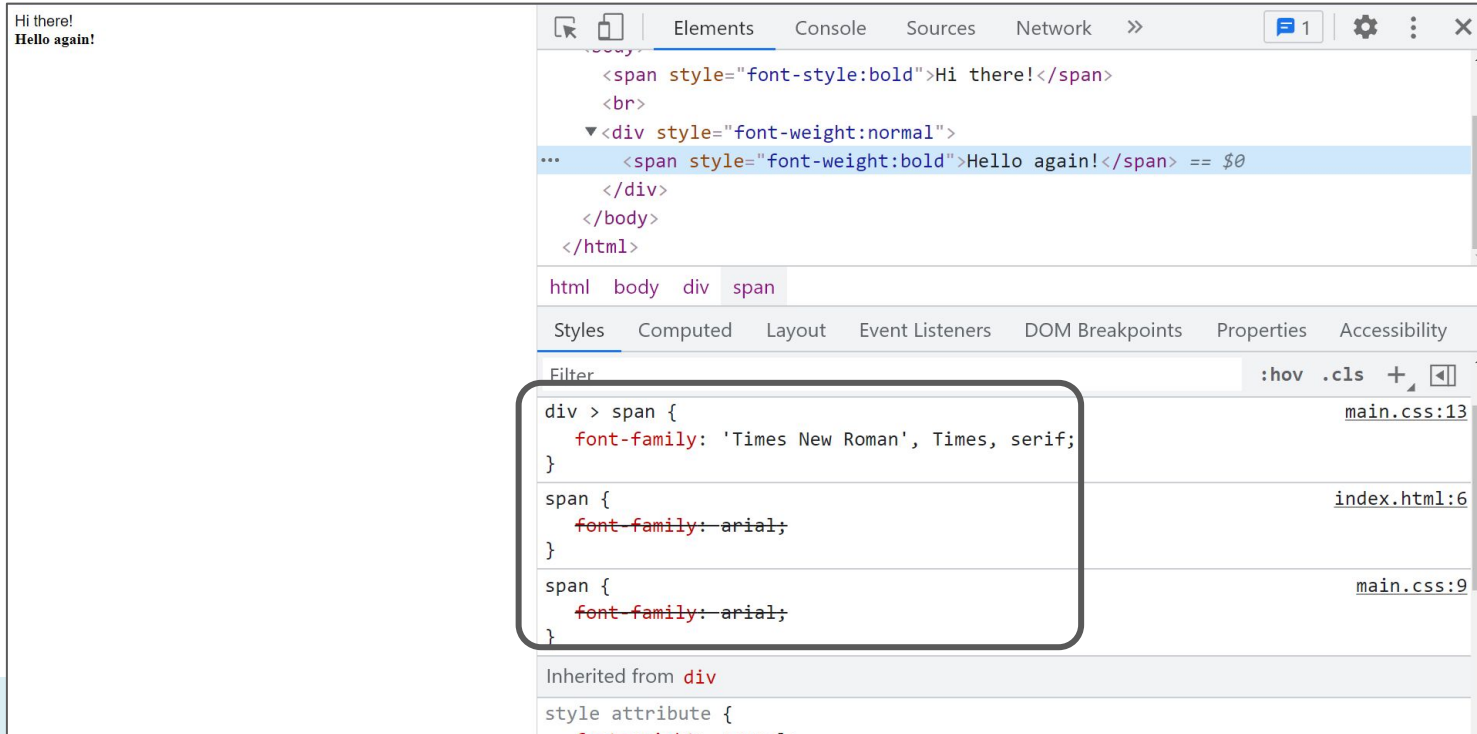
- Cascading Style Sheets
  - Reuse: styling rules for tags, classes, types
  - Reuse: not just at the leafs!
- What if there are no conflicts?

```
<div style="font-family:arial">  
  <span style="font-weight:bold">Hello again!</span>  
</div>
```

- How would you implement this?

# Style: CSS

What is happening here?



The screenshot shows a web browser's developer tools interface. On the left, a preview of the page content is visible, showing the text "Hi there!" and "Hello again!". The main panel displays the HTML structure, with the following code snippet highlighted:

```
<span style="font-style:bold">Hi there!</span>
<br>
<div style="font-weight:normal">
...
  <span style="font-weight:bold">Hello again!</span> == $0
</div>
</body>
</html>
```

The breadcrumb below the HTML shows the path: `html > body > div > span`. The right panel shows the "Styles" tab, with a filter set to `:hov .cls`. The following CSS rules are listed:

- `div > span {`
  - `font-family: 'Times New Roman', Times, serif;`
  - `}`
- `span {`
  - `font-family: arial;`
  - `}`
- `span {`
  - `font-family: arial;`
  - `}`

The source for each rule is indicated on the right: `main.css:13`, `index.html:6`, and `main.css:9`. A box highlights the first two rules. At the bottom, it shows "Inherited from `div`" and "style attribute {".

# Decorator

What is happening here?

- To compute the style of an element:
  - Apply its tag-default style
  - **Wrap** in added style rules (tag-specific or general)
    - Text: font-family, weight, etc.
  - Inherit parents' style
    - Conflicts lead to overrides
- Makes *themes* really powerful

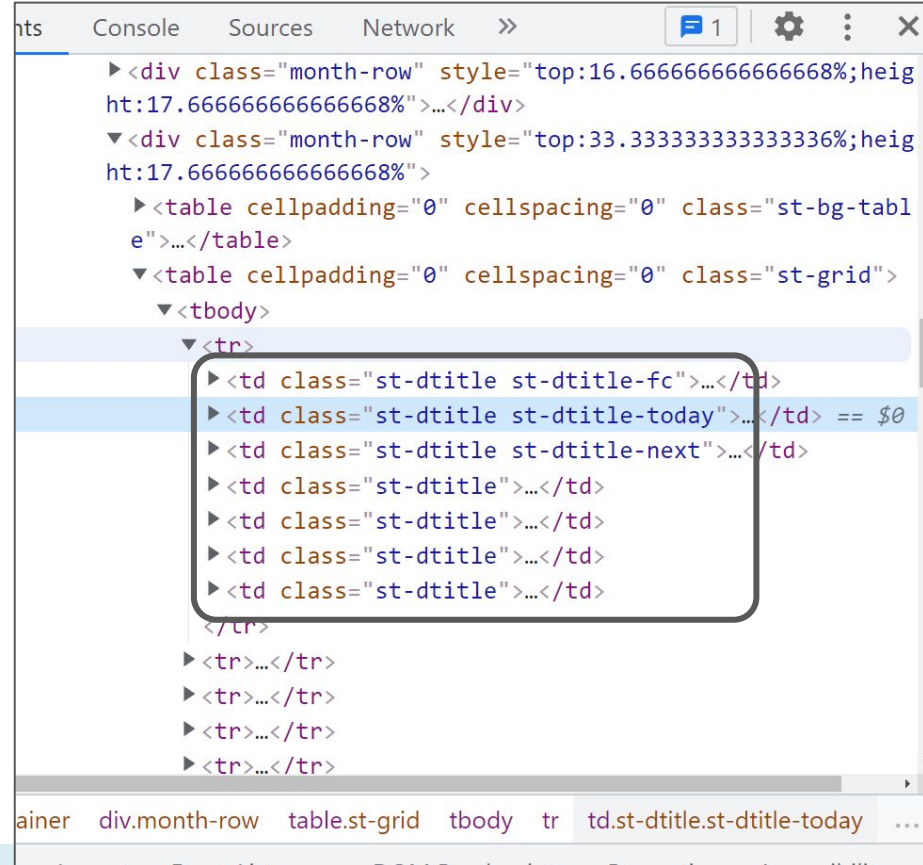
Technically, HTML is streamed top-to-bottom; CSS works bottom-up

# CSS: classes

Let's not repeat custom style

- Use any nr. of class label(s)
- Class styles get added
- Facilitates reuse

How would you implement this?



```
<div class="month-row" style="top:16.66666666666668%;height:17.66666666666668%">...</div>
<div class="month-row" style="top:33.33333333333336%;height:17.66666666666668%">
  <table cellpadding="0" cellspacing="0" class="st-bg-table">...</table>
  <table cellpadding="0" cellspacing="0" class="st-grid">
    <tbody>
      <tr>
        <td class="st-dtitle st-dtitle-fc">...</td>
        <td class="st-dtitle st-dtitle-today">...</td> == $0
        <td class="st-dtitle st-dtitle-next">...</td>
        <td class="st-dtitle">...</td>
        <td class="st-dtitle">...</td>
        <td class="st-dtitle">...</td>
        <td class="st-dtitle">...</td>
      </tr>
      <tr>...</tr>
      <tr>...</tr>
      <tr>...</tr>
      <tr>...</tr>
    </tbody>
  </table>
</div>
```



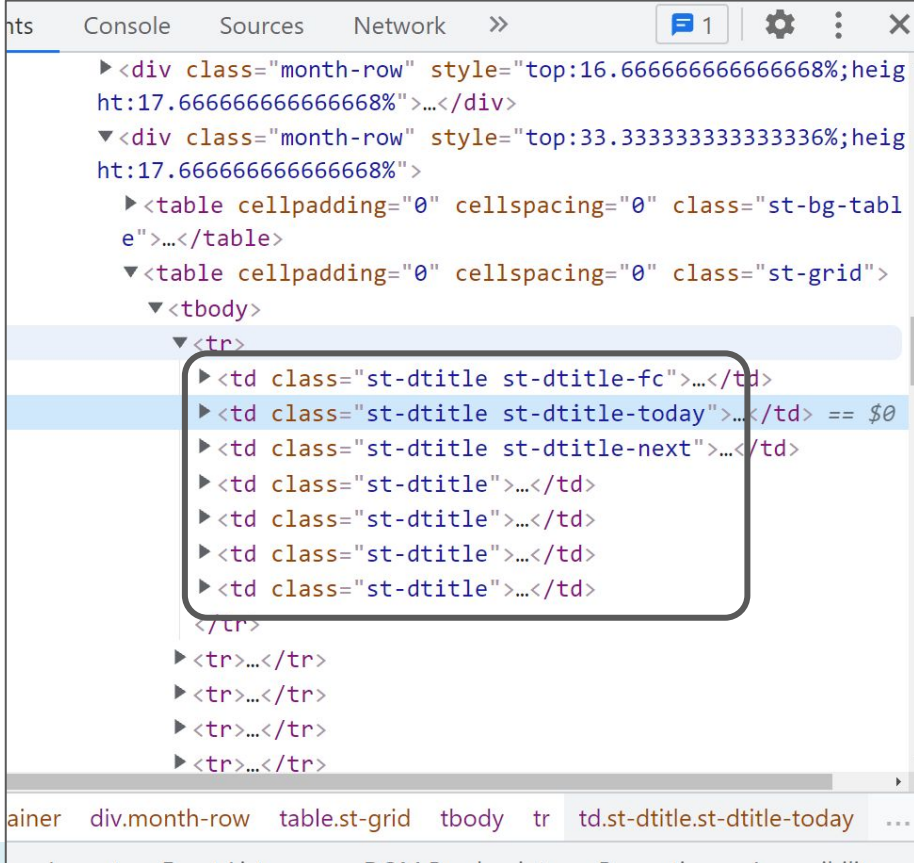
# Strategy or Observer?

```
<div class="month-row" style="top:16.66666666666668%;height:17.66666666666668%">...</div>
  <div class="month-row" style="top:33.33333333333336%;height:17.66666666666668%">
    <table cellpadding="0" cellspacing="0" class="st-bg-table">...</table>
    <table cellpadding="0" cellspacing="0" class="st-grid">
      <tbody>
        <tr>
          <td class="st-dtitle st-dtitle-fc">...</td>
          <td class="st-dtitle st-dtitle-today">...</td> == $0
          <td class="st-dtitle st-dtitle-next">...</td>
          <td class="st-dtitle">...</td>
          <td class="st-dtitle">...</td>
          <td class="st-dtitle">...</td>
          <td class="st-dtitle">...</td>
        </tr>
        <tr>...</tr>
        <tr>...</tr>
        <tr>...</tr>
        <tr>...</tr>
      </tbody>
    </table>
  </div>
```

# Strategy or Observer?

Either could apply

- Both involve callback
- Strategy:
  - Typically single
  - Often involves a return
- Observer:
  - Arbitrarily many
  - Involves external updates



```
>>
▶<div class="month-row" style="top:16.66666666666668%;height:17.66666666666668%">...</div>
▼<div class="month-row" style="top:33.33333333333336%;height:17.66666666666668%">
  ▶<table cellpadding="0" cellspacing="0" class="st-bg-table">...</table>
  ▼<table cellpadding="0" cellspacing="0" class="st-grid">
    ▼<tbody>
      ▼<tr>
        ▶<td class="st-dtitle st-dtitle-fc">...</td>
        ▶<td class="st-dtitle st-dtitle-today">...</td> == $0
        ▶<td class="st-dtitle st-dtitle-next">...</td>
        ▶<td class="st-dtitle">...</td>
        ▶<td class="st-dtitle">...</td>
        ▶<td class="st-dtitle">...</td>
        ▶<td class="st-dtitle">...</td>
      </tr>
      ▶<tr>...</tr>
      ▶<tr>...</tr>
      ▶<tr>...</tr>
      ▶<tr>...</tr>
    </tbody>
  </table>

```

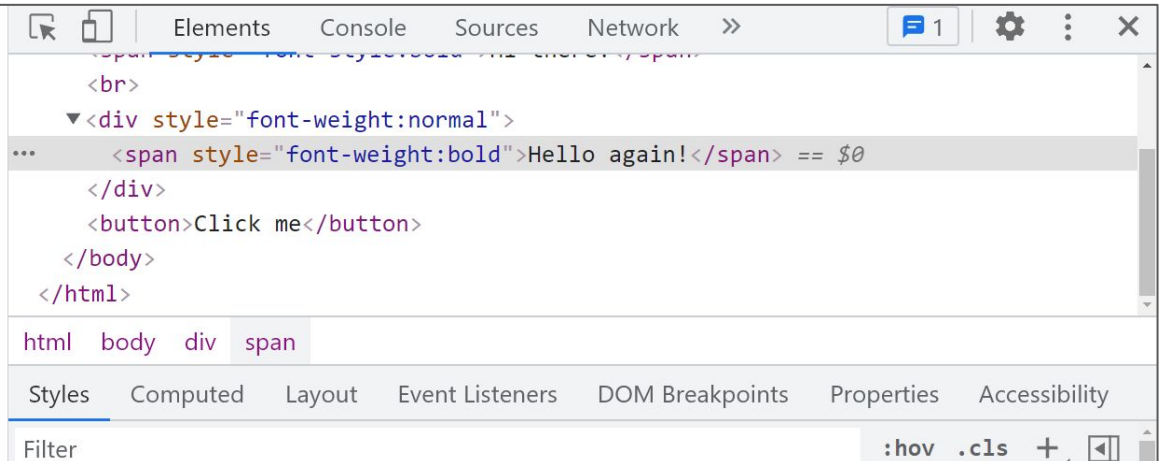
# Interactivity

A GUI is more than a document

- How do we make it “work”?

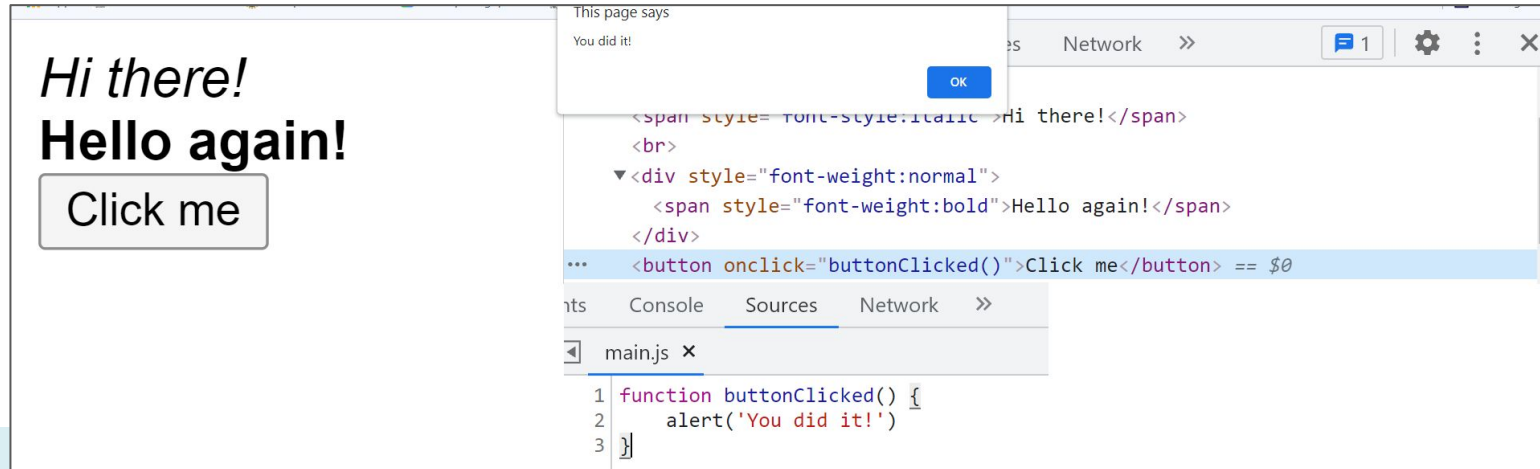
Hi there!  
**Hello again!**

Click me



# Actions: JavaScript

- Key: event listeners (what's that pattern?)
- (frontend) JS is highly event-driven
  - Respond to window `onLoad` event, content loads (e.g., ads)
  - Respond to clicks, moves



# Observer Pattern

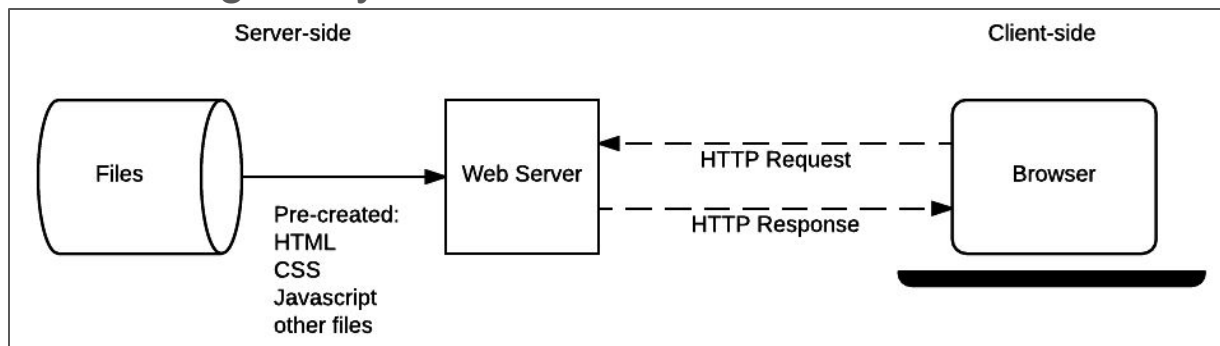
- Manages publishers and subscribers
  - Here, button publishes its 'click' events
  - `buttonClicked` subscribes to 1+ updates
- Flexibility and Reuse
  - Multiple observers per element
  - Shared observers across elements

# Step Back

- What is our website now?
  - Layout, style, interaction
  - What is missing?

# Static Web Pages

- Delivered as-is, final
  - Consistent, often fast
  - Cheap, only storage needed
- “Static” a tad murky with JavaScript
  - We can still have buttons, interaction
  - But it won’t “go” anywhere -- the server is mum



# Static Web Pages

- Delivered as-is, final
  - Consistent, often fast
  - Cheap, only storage needed
- Maintain with *static website generators*
  - Or you'll be doing a lot of copying
  - Coupled with themes => rapid development, deployment
  - Quite popular, e.g. hosting on GH Pages



# Static Web Pages

- But ...
  - No persistence (at least, not obviously)
  - No customizability (e.g., accounts)
  - No communication (payment, chat, etc)
  - Realistically, no intensive jobs

# Dynamic Web Pages

- Client/Server
  - Someone needs to answer the website's calls
    - Doesn't need to be us!
  - Host a webserver
    - Serves pages, handles calls
    - For static pages too!
- We'll show you more tomorrow (Wednesday)

# Web Servers

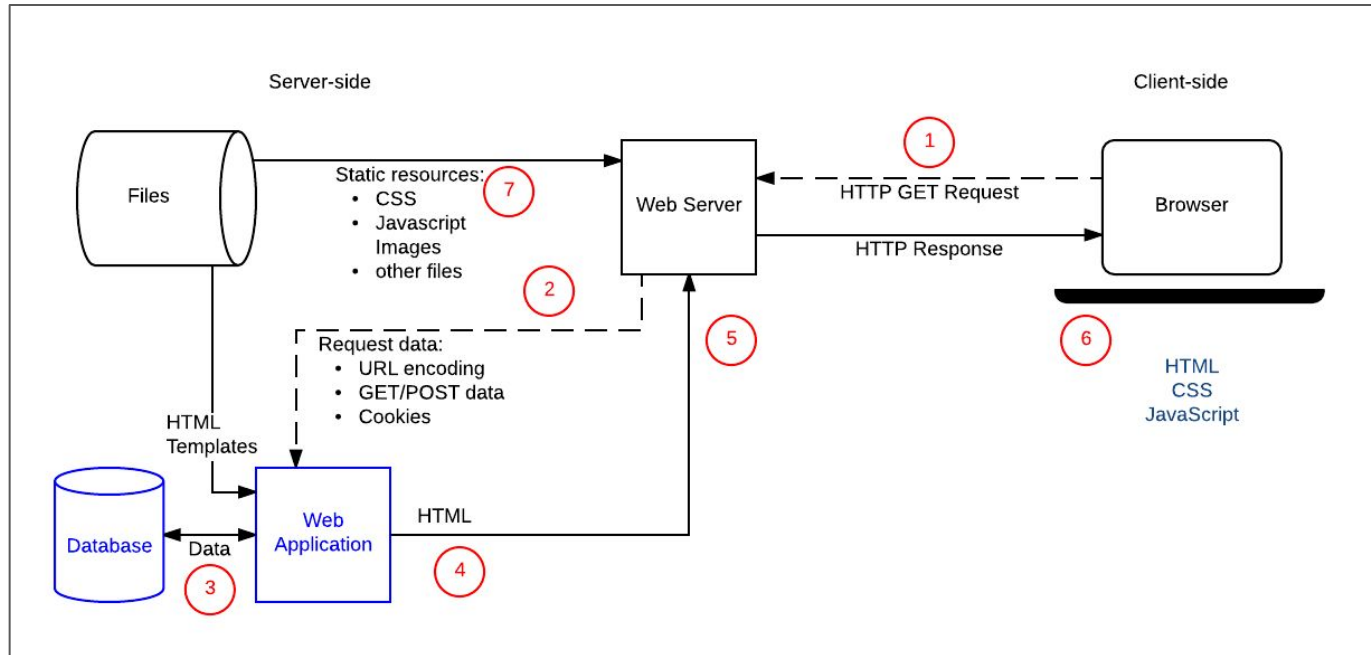
- Communicate via HyperText Transfer Protocol
  - URL (the address)
  - Method:
    - GET: retrieve data. Parameters in URL ``...?key=value&key2=value2`` and message body
    - POST: store/create data. Parameters in request body
    - Several more, rarely used
  - Responses:
    - *Status Code*. We all know 404. 2XX family is OK.
    - And possible data. E.g., entire HTML page.

# Web Servers

- Communicate via HyperText Transfer Protocol
  - URL (the address)
  - Method:
    - GET: retrieve data. Parameters in URL ``...?key=value&key2=value2`` and message body
    - POST: store/create data. Parameters in request body
    - Several more, rarely used
  - Responses:
    - *Status Code*. We all know 404. 2XX family is OK.
    - And possible data. E.g., entire HTML page.
  - POST makes no sense for static sites!
  - As do GETs with parameters

# Web Servers

Dynamic sites can do more *work*



[https://developer.mozilla.org/en-US/docs/Learn/Server-side/First\\_steps/Client-Server\\_overview#anatomy\\_of\\_a\\_dynamic\\_request](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview#anatomy_of_a_dynamic_request)

# AJAX

- Originally: “Asynchronous JavaScript and XML”
  - Updates parts of a page dynamically
  - Sends XMLHttpRequests with a callback
  - On return, check the code; handle success and failure.
  - Asynchronous, naturally decouples backend from UI

# AJAX

- Originally: “Asynchronous JavaScript and XML”
  - Updates parts of a page dynamically
  - Sends XMLHttpRequests with a callback
  - On return, check the code; handle success and failure.
  - Asynchronous, naturally decouples backend from UI
- Slowly being phased out
  - Replace with `fetch`, which uses... Promises
    - More next week

# How to Web App?

- Let's avoid generating HTML from scratch on every call
  - Map requests to handler code
    - Fetch data, process
  - Generate and return HTML
- Historically: PHP
  - Modifies HTML pages server-side on request; strong ties to SQL

```
<?php
// The global $_POST variable allows you to access the data sent with the POST method by name
// To access the data sent with the GET method, you can use $_GET
$say = htmlspecialchars($_POST['say']);
$to  = htmlspecialchars($_POST['to']);

echo $say, ' ', $to;

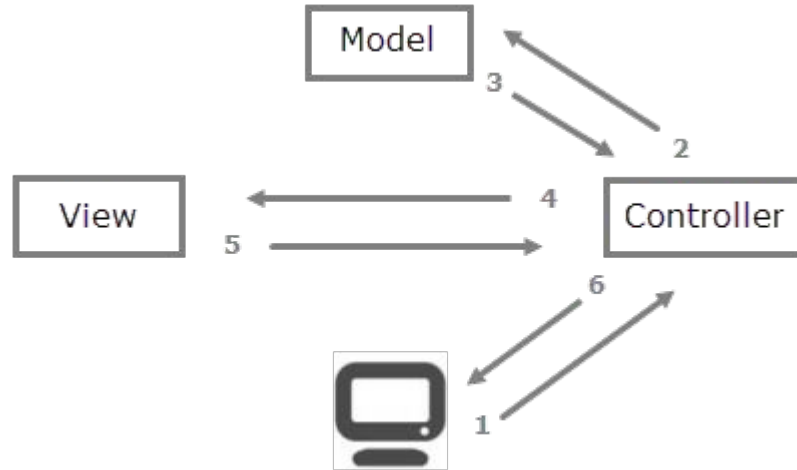
?>
```



# How to Web App?

- Let's avoid generating HTML from scratch on every call
  - Map requests to handler code
    - Fetch data, process
  - Generate and return HTML
- Or use a framework
  - Python: Flask, Django
  - NodeJS: Express
  - Spring for Java
  - [Many others](#), differences in **weight**, features

# Model-View-Controller (MVC)



<https://overiq.com/django-1-10/mvc-pattern-and-django/>

# MVC is ubiquitous

Separates:

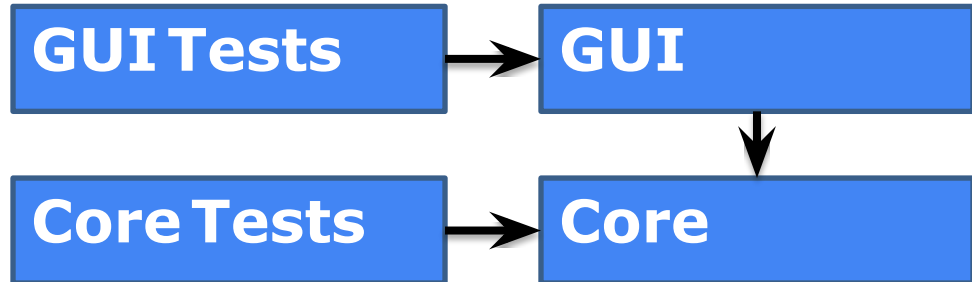
- Model: data organization
  - Interface to the database
- View: data representation (typically HTML)
  - Often called *templates* in web-dev; “view” is a bit overloaded
- Controller: intermediary between client and model/view
  - Typically asks model for data, view for HTML

# Core implementation vs. GUI

- Core implementation: application logic
  - Computing some result, updating data
- GUI
  - Graphical representation of data
  - Source of user interactions
- Design guideline: *avoid coupling the GUI with core application*
  - Multiple UIs with single core implementation
  - Test core without UI

# Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to communicate information from the core (Model) to the GUI (View)



# Summary

- GUIs are full of design patterns
  - Helpful for reuse, delegation in complex environments
- Covered the basics of HTML, CSS, JS, servers
  - Needed for dynamic web pages
  - Decouple the GUI; architect your backend
  - A lot more to learn (security, performance, privacy), but this will do
- You will build this
  - At a small scale