



Software Development with Agentic Coding Tools

Learning a Codebase in a New Language

Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

Week 1 at a New Startup: Into the Deep End!

The tech lead points you to a **pull request** in GitHub, with a few lines of code. The codebase has 100s of files and no docs.

It's in *TypeScript*. Where do you start?

Commit 6751174

TheArcaneBrony committed last week

Member list kinda works now?

1 parent 7837a03 commit 6751174

10 files changed +235 -71

Filter files... Search within code

```
.gitignore
@@ -9,6 +9,7 @@ assets/cache
9 9 .env
10 10 config.json
11 11 assets/cacheMisses
12 12 + http-client.private.env.json
13 13
14 14 .vscode/settings.json
15 15

extra/admin-api/DataMappings/Spacebar.DataMappings.Generic/Member.cs
@@ -1,3 +1,4 @@
1 + using System.Text.Json;
2 using Spacebar.Models.Generic;
3 3 namespace Spacebar.DataMappings.Generic;
4 4
13 14 @@ -13,10 +14,10 @@ public static Member ToPublicMember(this Models.Db.Models.Member member, Partial
14 14 Avatar = string.IsNullOrWhiteSpace(member.Avatar) ? null : member.Avatar,
15 15 Banner = string.IsNullOrWhiteSpace(member.Banner) ? null : member.Banner,
16 16 Collectibles = member.Collectibles,
17 17 DisplayNameStyles = member.DisplayNameStyles,
18 18 DisplayNameStyles = JsonSerializer.Deserialize<DisplayNameStyle>(member.DisplayNameStyles ?? "null"),
19 19 Bio = string.IsNullOrWhiteSpace(member.Bio) ? null : member.Bio,
20 20 Nick = string.IsNullOrWhiteSpace(member.Nick) ? null : member.Nick,
```



Week 1 at a New Startup: Into the Deep End!

The tech lead points you to a **pull request** in GitHub, with a few lines of code. The codebase has 100s of files and no docs.

It's in *TypeScript*. Where do you start?

Commit 6751174

TheArcaneBrony committed last week

Member list kinda works now?

1 parent 7837a03 commit 6751174

10 files changed +235 -71

Filter files... Search within code

```
.gitignore
@@ -9,6 +9,7 @@ assets/cache
9 .env
10 config.json
11 assets/cacheMisses
12 + http-client.private.env.json
13
14 .vscode/settings.json
15

extra/admin-api/DataMappings/Spacebar.DataMappings.Generic/Member.cs
@@ -1,3 +1,4 @@
1 + using System.Text.Json;
2 using Spacebar.Models.Generic;
3
4 namespace Spacebar.DataMappings.Generic;
5
6 public static Member ToPublicMember(this Models.Db.Models.Member member, Partial
13 Avatar = string.IsNullOrWhiteSpace(member.Avatar) ? null : member.Avatar,
14 Banner = string.IsNullOrWhiteSpace(member.Banner) ? null : member.Banner,
15 Collectibles = member.Collectibles,
16 DisplayNameStyles = member.DisplayNameStyles,
17 + JsonSerializer.Deserialize<DisplayNameStyle>(member.DisplayNameStyles ?? "null"),
18 Bio = string.IsNullOrWhiteSpace(member.Bio) ? null : member.Bio,
19 Nick = string.IsNullOrWhiteSpace(member.Nick) ? null : member.Nick,
```



Week 1 at a New Startup: Into the Deep End!

The tech lead points you to a **pull request** in GitHub, with a few lines of code. The codebase has 100s of files and no docs.

It's in *TypeScript*. Where do you start?

Commit 6751174

TheArcaneBrony committed last week

Member list kinda works now?

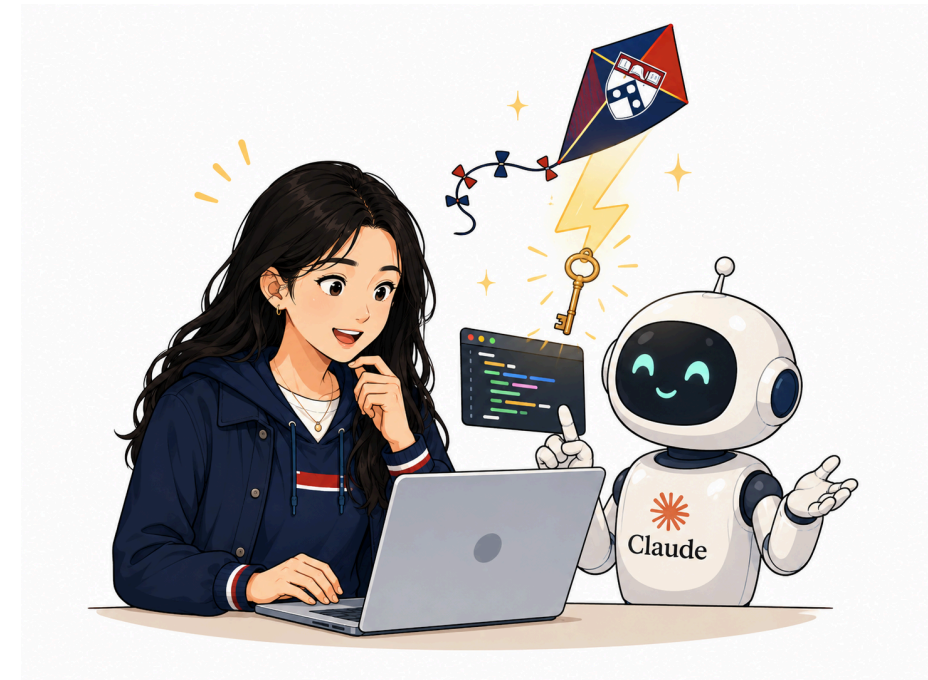
1 parent 7837a03 commit 6751174

10 files changed +235 -71

Filter files... Search within code

```
.gitignore
@@ -9,6 +9,7 @@ assets/cache
9 9 .env
10 10 config.json
11 11 assets/cacheMisses
12 12 + http-client.private.env.json
13 13
14 14 .vscode/settings.json
15 15

extra/admin-api/DataMappings/Spacebar.DataMappings.Generic/Member.cs
@@ -1,3 +1,4 @@
1 1 + using System.Text.Json;
2 2 using Spacebar.Models.Generic;
3 3 namespace Spacebar.DataMappings.Generic;
4 4
13 14 @@ -13,10 +14,10 @@ public static Member ToPublicMember(this Models.Db.Models.Member member, Partial
14 14 Avatar = string.IsNullOrWhiteSpace(member.Avatar) ? null : member.Avatar,
15 15 Banner = string.IsNullOrWhiteSpace(member.Banner) ? null : member.Banner,
16 16 Collectibles = member.Collectibles,
17 17 DisplayNameStyles = member.DisplayNameStyles,
18 18 DisplayNameStyles = JsonSerializer.Deserialize<DisplayNameStyle>(member.DisplayNameStyles ?? "null"),
19 19 Bio = string.IsNullOrWhiteSpace(member.Bio) ? null : member.Bio,
20 20 Nick = string.IsNullOrWhiteSpace(member.Nick) ? null : member.Nick,
```



Week 1 at a New Startup: Into the Deep End!

Expect little help and guidance. But you won't be lost!

- **git** helps you see the past and manage your own development
- **Google**, our old friend
- your **CIS undergraduate education**
- and, recently, **agentic tools**.

With *the right questions* and some *best practices*, you can deal with anything!

Learning to Swim

Today, we'll:

- Develop skills to **read, verify, and direct** code modifications in an unfamiliar language.
- Learn to **ask questions** and **use tools** to fill the gaps and build understanding.

A Senior Engineer Knows What They *Should* Know

Senior engineers move between teams and languages regularly. They develop **goal-directed learning** skills, e.g.:

- Identify what is different
- Ground everything in the actual codebase and with examples
- Use a knowledgeable "colleague" to fill gaps on demand

Identifying what is (and isn't!) different

```
export interface Event {  
  guild_id?: string;  
  user_id?: string;  
  session_id?: string;  
  channel_id?: string;  
  created_at?: Date;  
  event: EVENT;  
  // eslint-disable-next-line @t ...  
  data?: any;  
  reconnect_delay?: number;  
  origin?: string;  
  transaction_id?: string;  
}
```

If your PR is about updating how events are handled in Spacebar, we might start by looking at the `Event` interface.

- What looks basically familiar from Java or Python?
- What is foreign, scary, and unusual here?

Grounding Something in the Codebase

`event: EVENT` as a field in an interface called `Event` is... intriguing.

- `Event` is an object "shape"—a type!—created by this interface with several fields
- `event`, apparently, is the name of a field. (Is it... recursive?)
- `EVENT` is acting like a type, presumably different from `Event`

This is confusing, so let's go look at `EVENT`'s definition to see if it gives better context...

Grounding Something in the Codebase

```
export type EVENT =  
  | "READY"  
  | "CHANNEL_CREATE"  
  | "CHANNEL_UPDATE"  
  | "CHANNEL_DELETE"  
  | "CHANNEL_PINS_UPDATE"  
  | "CHANNEL_RECIPIENT_ADD"  
  | "CHANNEL_RECIPIENT_REMOVE"  
  | "GUILD_CREATE"  
  ...
```

Ah, OK! `EVENT` is a union type (remember OCaml???) that's like a Java `enum` that represents the kind of action that the `Event` is signaling.

...Right?

Here's that last 'asking a knowledgeable "colleague" part:

`</> Event.ts`

How does TS distinguish between the type(?) of `Event` created by the `interface Event` and the enumerated(?) `type EVENT`

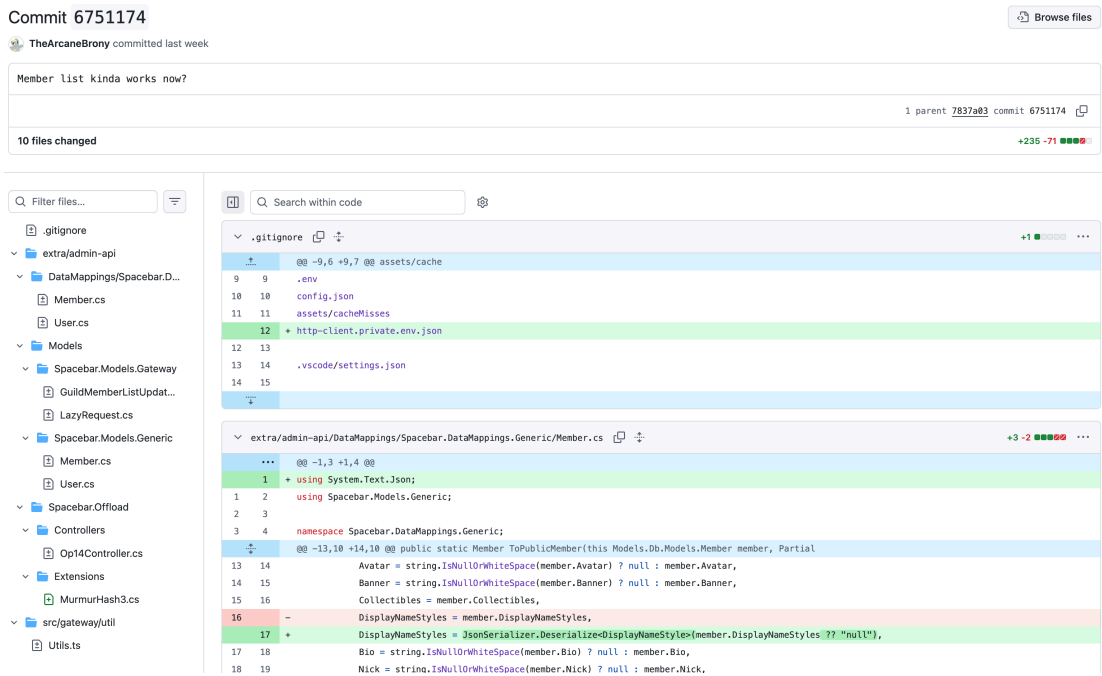
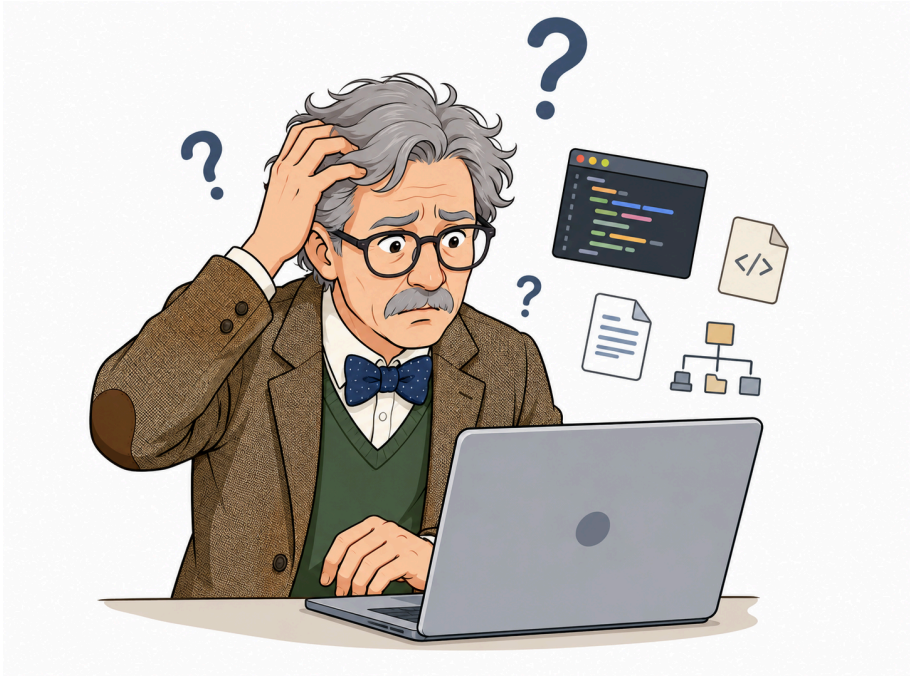
- Thought for 10s
- They're just two different identifiers — TypeScript distinguishes them the same way it distinguishes any two names: by exact, case-sensitive spelling. `Event` and `EVENT` are not the same symbol; the fact that one looks like an all-caps version of the other is a human naming convention, not something the type checker treats specially.

Concretely, in this file ([Event.ts](#)):

- `interface Event { ... }` at line 37 declares an **object shape** — a structural type describing what properties an `Event` value must have (`guild_id?`, `data?`, `event: EVENT`, etc.).
- `type EVENT = "READY" | "CHANNEL_CREATE" | ...` at line 776 declares a **union of string literal types** — a closed set of allowed string values.

A Senior Engineer Knows What They *Should* Know

Start by figuring out **which questions to ask!**



A Senior Engineer Knows What They *Should* Know

Start by figuring out **which questions to ask!**

Commit 6751174

TheArcaneBrony committed last week

Member list kinda works now?

1 parent [7837a83](#) commit 6751174

10 files changed

+235 -71

.gitignore

extra/admin-api

DataMappings/Spacebar.D...

Member.cs

User.cs

Models

Spacebar.Models.Gateway

GuildMemberListUpdat...

LazyRequest.cs

Spacebar.Models.Generic

Member.cs

User.cs

Spacebar.Offline

Controllers

Opt14Controller.cs

Extensions

MurmurHash3.cs

src/gateway/util

Utils.ts

Search within code

.gitignore

+1

```

1 @@ -9,6 +9,7 @@ assets/cache
9 9 .env
10 10 config.json
11 11 assets/cacheMisses
12 + http-client.private.env.json
12 13
13 14 .vscode/settings.json
14 15

```

extra/admin-api/DataMappings/Spacebar.DataMappings.Generic/Member.cs

+3 -2

```

... @@ -1,3 +1,4 @@
1 + using System.Text.Json;
1 2 using Spacebar.Models.Generic;
2 3
3 4 namespace Spacebar.DataMappings.Generic;
4
5 @@ -13,10 +14,10 @@ public static Member ToPublicMember(this Models.Db.Models.Member, Partial
13 14 Avatar = string.IsNullOrEmpty(member.Avatar) ? null : member.Avatar,
14 15 Banner = string.IsNullOrEmpty(member.Banner) ? null : member.Banner,
15 16 Collectibles = member.Collectibles,
16 - DisplayNameStyles = member.DisplayNameStyles,
17 + DisplayNameStyles = JsonSerializer.Deserialize<DisplayNameStyle>(member.DisplayNameStyles ?? "null"),
17 18 Bio = string.IsNullOrEmpty(member.Bio) ? null : member.Bio,
18 19 Nick = string.IsNullOrEmpty(member.Nick) ? null : member.Nick,

```

Good starting points in order to feel in control:

- What (language) am I looking at?
- What does the software do?
- What does this module do, within the context of the software?
- What background do I need to ask the right questions?

Big Question

We hope much of your learning in this class will be *directed inquiry*... after a bit of prep!

Coming from CIS 1200 and 1210, you will have **some but not all of the prerequisite background knowledge to ask the right questions.** We'll start with some background:

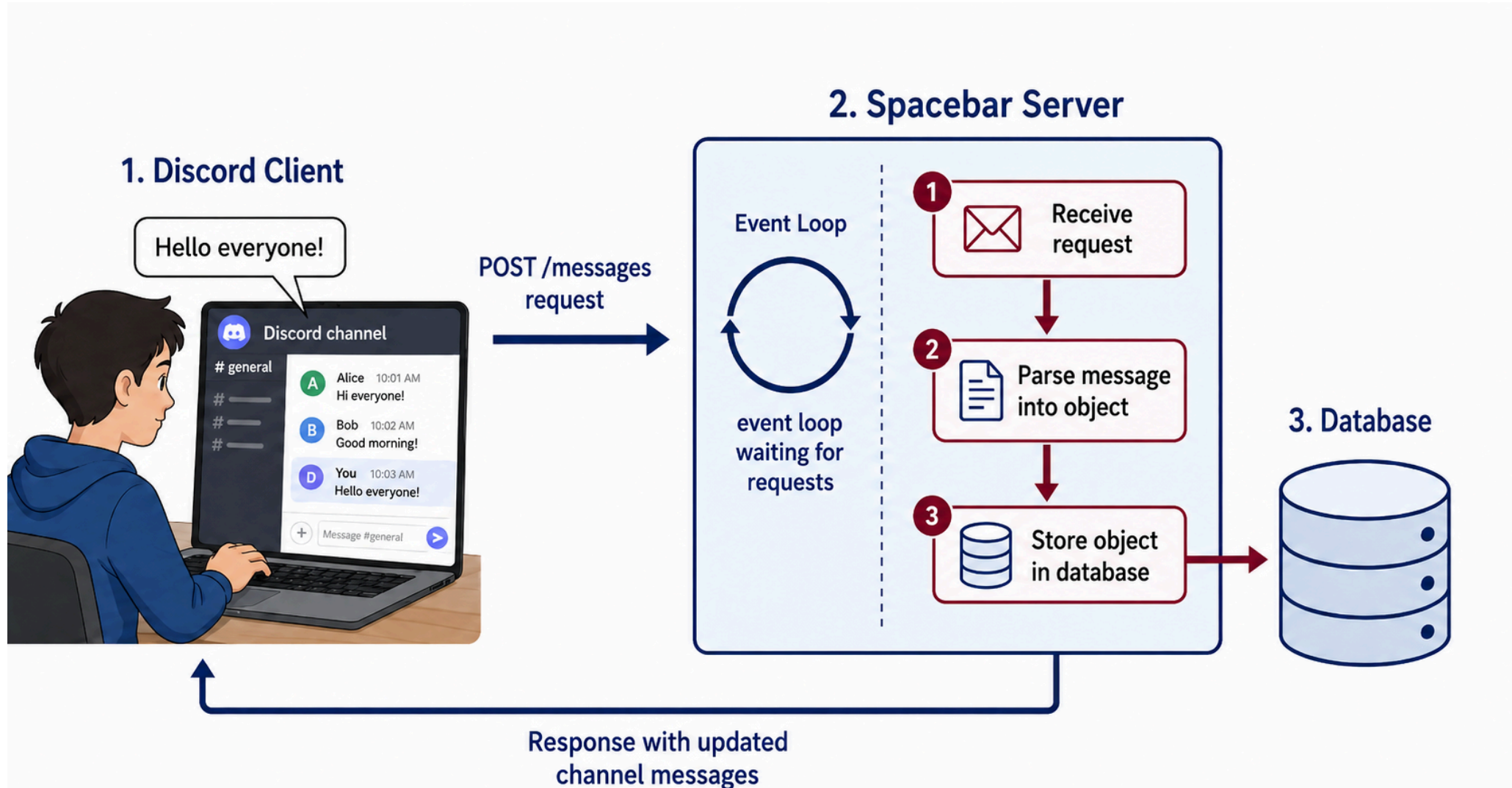
1. **The software** and what a server like Spacebar does.
2. **The language** and its novel concepts.
3. **Precise prompting** (in a live demo).

Lab 2 puts you in the driver's seat, to practice learning Spacebar with AI, yourself!

What is a Chat Server and How Does It Work?

Let's get some background in Spacebar, clients and servers, and storage

What Is a Chat Server (under the hood)?



Some Quick Building Blocks

You'll want to learn **top-down** -- figure out how to make sense of the high-level concepts, then go to details from there.

- Calling functions on a remote **server**
- TypeScript -- an *almost* familiar language
- Storing objects persistently in a **data management system**

Function Calls

Call a named *function* (procedure), send it *parameters*, and wait for it to *return a value or an exception*.

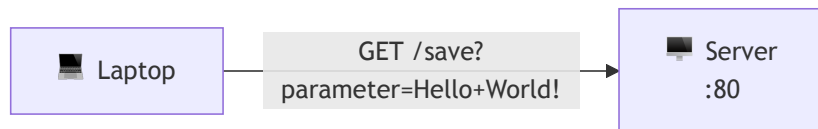
```
Worker obj = new Worker();  
obj.save("Hello World!");
```

```
class Worker {  
    public void save(String parameter) {  
        BufferedWriter bw = getWriter();  
        bw.write(parameter);  
    }  
}
```

Remote Procedure Calls: The Client Sends a Message

Call a named *function* (procedure), send it *parameters*, and wait for it to *return a value or an exception*. **Suppose the function is on another machine!**

- Messages are sent to the machine's **address**.
- Common request types are GET, POST, PUT, DELETE



Let's make a request to another **computer**:

```
HttpClient connection = HttpClient
    .newHttpClient();
String url = "http://server/api/save"
    + "?parameter=";
HttpRequest request = HttpRequest
    .newBuilder().uri(URI.create(url +
        "Hello World!")).GET();
connection.send(request.build(),
    BodyHandlers.ofString());
```

17

Remote Procedure Calls: The Server Daemon Loops

The server registers a *ServerSocket* with a numeric **port**. *Ask your Agent: Where does the port number come from? Does it need to be 80?*

The server has an **event loop**:

```
void saveHandler(String parameter) {
    BufferedWriter bw = getWriter();
    bw.write(parameter);
}

void eventDaemon() {
    ServerSocket sk = new ServerSocket(80); // port set here
    while (true) {
        Request req = getRequest(sk);
        if (req.type.equals("save")) { saveHandler(req.getParameter("parameter")); }
        ...
    }
}
```

Many Clients + Servers Are Written in TypeScript

A statically typed version of JavaScript -- which was "inspired by Java"

Goal of the modern Web: lightweight apps *in any browser*!

- **JavaScript**: minimalist, Java-ish syntax. Extended to **TypeScript**.
- Most modern web software is built in these languages!
- Write Javascript for both the client (V8, SpiderMonkey) and server (Node.js)

To go with it: a simple data syntax, **JSON**, which lets you compose:

- Lists: `[item1, item2]`
- Dictionaries: `{key1: value1, key2: value2}`
- And basic scalars: ints, floats, strings
- Those items/values can also be lists and dictionaries!

Our Server Code in TypeScript, Step by Step

In practice, a server doesn't write its own socket loop — a **framework**, e.g. Express.js, registers a handler (function) per route:

```
import express, { Request, Response } from "express";  
const app = express();
```

Our Server Code in TypeScript, Step by Step

An application of any real size requires a persistent data storage backend, so we'll need to set up a connection there, although details are omitted for now...

```
import express, { Request, Response } from "express";  
const app = express();  
const database = // ...
```

Our Server Code in TypeScript, Step by Step

We create a handler function to represent what happens when the user wants to "save" something.

- Functions are defined with `function` keyword
- Arguments use a `name: type` syntax

```
import express, { Request, Response } from "express";
const app = express();
const database = // ...
function saveHandler(req: Request, res: Response) {
  const parameter = req.query.parameter;
  database.write(parameter);
  res.status(200).send(parameter);
}
```

22

Think: what does this suggest about responses are built and sent?

Our Server Code in TypeScript, Step by Step

Register the function as the correct behavior for the `/save` **endpoint**, or the name of the URL where the user will route their request.

```
import express, { Request, Response } from "express";
const app = express();
const database = // ...
function saveHandler(req: Request, res: Response) {
    const parameter = req.query.parameter;
    database.write(parameter);
    res.status(200).send(parameter);
}
app.get("/save", saveHandler);
```

Our Server Code in TypeScript

Finally, `app.listen(80, () => ...)` gets your app listening on port 80 of this machine.

```
import express, { Request, Response } from "express";
const app = express();
const database = // ...
// Functions are defined with the function keyword
function saveHandler(req: Request, res: Response) {
  const parameter = req.query.parameter;
  database.write(parameter);
  res.status(200).send(parameter);
}
app.get("/save", saveHandler);
app.listen(80, () => {
  console.log("Server is listening on Port 80");
});
```

24

Chat: In Essence, Dictionaries of Lists

- A chat application running on a server ("guild") has a set of users, but much of the work involves **appending posts to channels**.
- A channel is essentially a dictionary (Map) from **channel name** to a **list of posts**, chronologically ordered!
 - DMs are essentially special channels

But, unlike most applications in, say, CIS 1210 -- our dictionaries and lists need to be **persistent**. If the server restarts, they should still live... in a **database**.

Storing Chat Message Objects

How do we save it persistently? Translate it into tables in a **relational database** with an **Object Relational Mapping (ORM)**:

```
@Entity()
class Message {
  @PrimaryKey()
  id: string;
  @Column()
  channel_id: string;
  @ManyToOne(() => User)
  author: User;
  @Column()
  content: string;
  @Column()
  timestamp: Date;
```

```
chatMsg = {
  "id": "812345678",
  "channel_id": "934567890",
  "author": { "id": "112345678",
              "username": "alice" },
  "content": "hello, world!",
  "timestamp": "2026-07-12T14:32:00Z"
}
const messageStore = Message.create
  (chatMsg);
await messageStore.save();
```

26

The Story So Far

- We are building a **server** that listens to **requests** from a client, Spacebar
- The server does this with an **event loop** that listens for requests, and **dispatches** them
 - In JavaScript this is handled by **Express** running on **Node.js**
 - We register handlers for paths (like `'/save'`)
- All communication is done by **network messages** via a protocol called HTTP
 - Data can be passed as **JSON** objects
 - Object-Relational Mappings (ORMs) let us easily store collections of objects

Now we know what we're building. Let's understand a bit more about how to read TypeScript!

Learning to Read TypeScript and JavaScript... Or Any Other Language

How can we leverage what we know from OCaml and Java to understand Spacebar source code?

Four Axes for Sizing Up Any New Language

Ask these of any language, TypeScript included:

1. **Paradigm** — functional? procedural? object-oriented? A mix?
2. **Typing** — static or dynamic? by name (nominal) or by content (structural)?
3. **Runtime model** — how does the runtime schedule and run concurrent code?
4. **Structure & composition** - modules organize code into files and packages; the object model organizes data and its operations

Going from Java to TypeScript, Items 2, 3 & 4 are most important.
Going from Java to Rust or Go hits different points.

TypeScript in 10 Minutes, for Java Programmers

Much of it will be familiar enough to read!

- Basic keywords and syntax (like nesting with `{` and `}` and ignoring whitespace) carry over from Java (though ending semicolons are optional)
- Types come after the variables, and `function` is a keyword:

```
function addNumbers(a: number, b: number): number {  
    return a + b;  
}
```

- TypeScript files are modules that import/export symbols—remember *encapsulation*?
 - `export async function onIdentify(this: WebSocket, data: Payload) {}`
 - `import { PayloadSchema } from "../schemas";`

Compile-Time Typing in TypeScript

- Java determines variable types at compile time, and the compiler carries the types through its computations. It will catch typing errors based on dynamic state.
- TypeScript adds type checking at **compile time**. But type checking doesn't *additionally* happen during computation, so it doesn't reason about things like JSON arguments.

```
let x: number = 42;  
x = "hello";           // ✗ Type 'string' is not assignable to type 'number'  
x = { id: 1 };         // ✗  
x = x + JSON.parse('"3x"'); // This will not be caught at compile time
```

The critical difference from Java: Type errors surface as `undefined`, `NaN`, or wrong-shaped data propagating through the program, rather than instant crashes like `NullPointerException`.

31

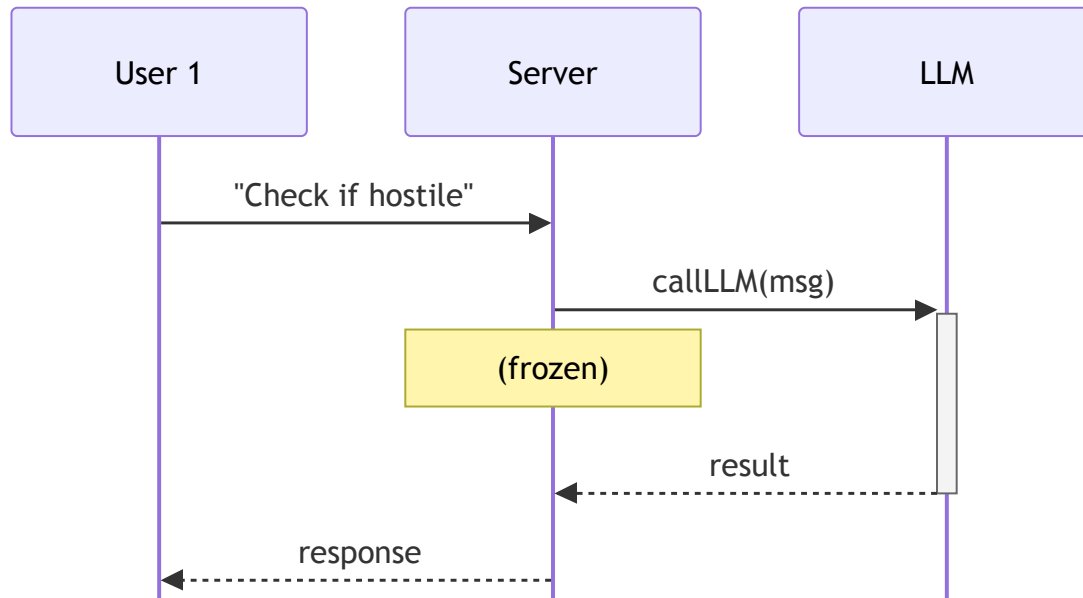
TypeScript Objects Are *Property Bags*

- **Java** objects exactly follow the blueprints (member variables, methods) of their class.
- A **TypeScript** object is a mutable *property bag*, finding the *nearest match* to a property.
 - An object has key/value pairs to its member variables and methods.
 - It also has a chain of `__proto__` links to its "prototypes."

```
class Person { constructor(public name: string) {} }
class Student extends Person { constructor(name: string, public courses: string[])
  { super(name); } }
const student1 = new Student("alice", ["CIS 1200", "CIS 1210"]);
student1.email = "alice@example.com"; // ✗ compile error: no such property
student1.name = 99; // ✗ compile error: wrong type
const bag = student1 as any; // escape the type checker
bag.email = "alice@example.com"; // ✓ legal at runtime – same object(!!)
delete bag.courses; // ✓ legal at runtime
bag.name = 99; // ✓ legal at runtime
```

Spacebar Needs to Support Concurrent Requests

Suppose our chat server filters hostile messages. It calls an LLM on each post.



```
// pseudocode
state = {}
while server_running:
    req = waitForRequest()

    state = { req, result: null }
    state.result = callLLM(req.msg)
    #           ^ blocks here
    #           nothing else happens

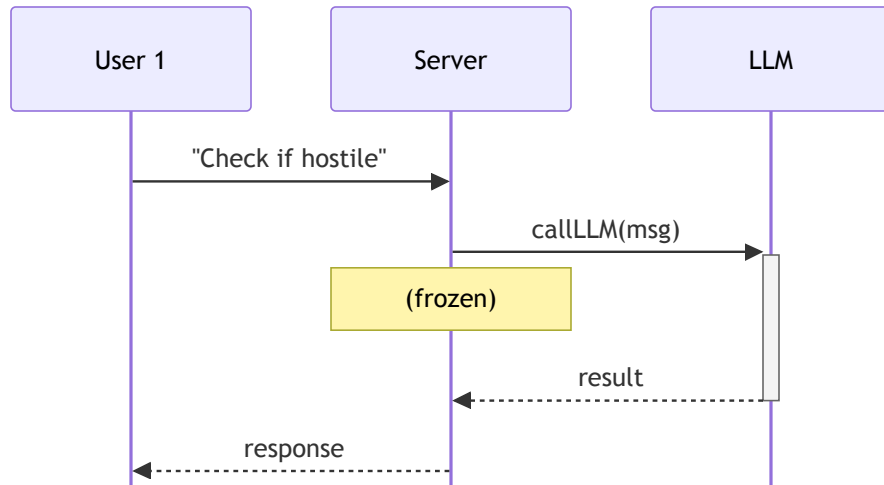
    sendResponse(state)
```

`callLLM()` takes 2 seconds.

33

Spacebar Needs to Support Concurrent Requests

Suppose our chat server filters hostile messages. It calls an LLM on each post.



User 2's request (a new message) arrives during the LLM call. Impatient User 2 waits in frustration.

```
// pseudocode
state = {}
while server_running:
    req = waitForRequest()

    state = { req, result: null }
    state.result = callLLM(req.msg)
    #           ^ blocks here
    #           nothing else happens

    sendResponse(state)
```

`callLLM()` takes 2 seconds.

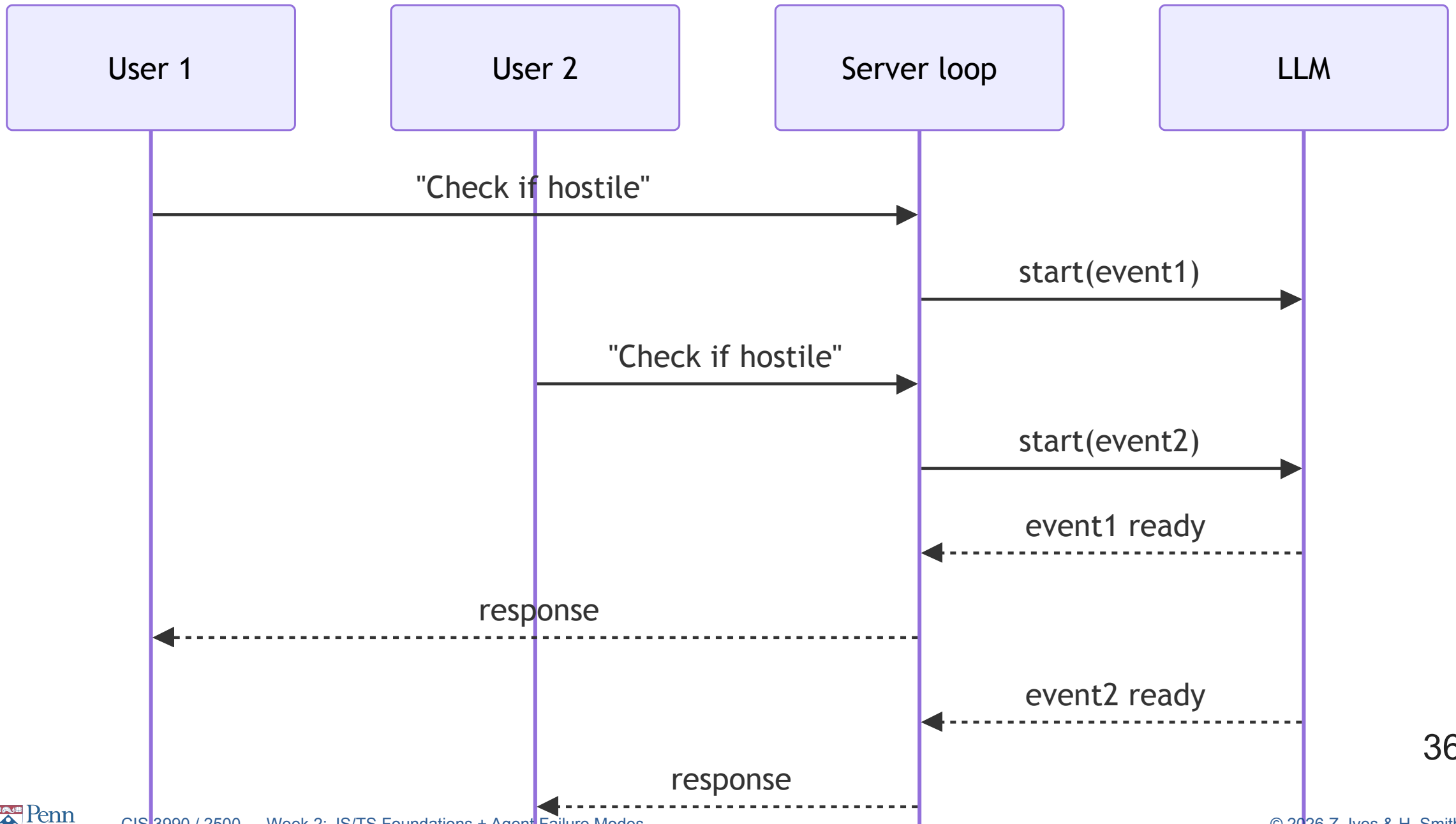
34

An Idea: Concurrency via *Event Objects*

Give each request a **finite state machine**. Store **state & data** in an event object.
Start the LLM call without waiting, and immediately loop back for the next request.

```
pending = []
while (server_running) {
  if (req = pollNewRequest()) {
    event = { req, result: null }
    startLLM(event)
    pending.add(event)
  }
  foreach (event in pending) {
    if (event.result ≠ null) { // reqs w/ LLM step done
      sendResponse(event)      // (assume this part is ~fast)
      pending.remove(event)
    }
  }
}
```

35



async/await Simplifies Event-Object Handling

Node.js runs the `pending` queue. `async/await` builds in the event-object pattern.

```
// on reaching the LLM call:
event = { req, result: null }
startLLM(event) // non-blocking
pending.add(event)
return // handler exits

// when LLM responds:
event.result = llmResult
// loop resumes handler here
sendResponse(event)
```

```
async function handleReq(req: Request) {
  const result = await callLLM(
    req.message
  );
  // save state, yield to loop;
  // resume here when LLM responds
  sendResponse(req, result);
}
```

Each `await` is where the runtime:

1. Saves state as an event object
2. Registers the pending call
3. Returns control to the caller
4. Resumes the function

37

A Tricky Execution Order

```
async function example() {  
  console.log("A");  
  await someDbQuery();      // (20 seconds)  
  console.log("B");  
}  
example();  
console.log("C");
```

Think: what gets printed here?

A Tricky Execution Order

```
async function example() {  
  console.log("A");           // runs immediately  
  await someDbQuery();        // saves state, yields to loop  
  console.log("B");           // runs when loop resumes this function  
}  
example();  
console.log("C");             // runs while example() is yielded  
// Output: A, C, B ← not A, B, C
```

Calling `example()` does *not* wait for it to finish. `example()` runs until `await`, then yields — just like a handler adding itself to `pending`. `console.log("C")` runs next because it is already on the call stack. Only when the loop has nothing more urgent does `example()` resume at `"B"`.

39

async/await in Action

JavaScript wraps the event-state object in a **Promise** and builds `async/await` over it.

```
async function sendMessage(  
  userId: string, channelId: string, text: string  
) : Promise<Message> {  
  const user      = await db.findUser(userId); // suspend here until resolved  
  const channel   = await db.findChannel(channelId);  
  const saved     = await db.insertMessage({ user, channel, content: text });  
  return saved; // wraps in Promise automatically  
}
```

- An `async` function **always returns a Promise** — a container for the future value.
- `await` is only legal **inside an async function**
- Hitting `await` suspends the function and resumes the calling event loop. **It resumes once the awaited Promise settles**

40

Closures — Functions That Remember

A **closure** is a function that *captures* variables from the scope where it was defined — keeping those variables alive even after the enclosing function has returned.

In JavaScript, closures are everywhere: every callback, every event handler, every module pattern uses them. You will read closures every time you open a Spacebar route file.

```
function makeCounter() {  
  let count = 0;    // local variable  
  return function increment() {  
    count++; // "captures" `count`  
    return count;  
  };  
}  
  
const counter = makeCounter();  
console.log(counter());  
console.log(counter());
```

Why this matters in Spacebar: every route handler closes over the Express `app` object, the database connection, and the config.

Understanding closures is how you trace why a handler has access to variables you didn't pass it explicitly.

Recap: Key Things You Need to Know about TypeScript

Most of your knowledge from Java will transfer. Summary of key differences.

#	CONCEPT	AXIS	JAVA	JAVASCRIPT / TYPESCRIPT
1	Type system	Typing	Static, checked at compile time, enforced at runtime	JS: dynamic, no runtime types. TS: static layer that <i>erases</i>
2	Object model	Structure & composition	Class blueprint → instances	Property bags with prototype chains
3	Execution model	Runtime model	OS threads, one per request	Single thread + event loop
4	Async patterns	Runtime model	Blocking I/O; stable <code>this</code>	Non-blocking; <code>this</code> shifts by call site
5	Closures	Structure & composition	Anonymous inner classes (limited)	Ubiquitous; capture live variables

Detailed Notes for Review

We'll skip discussion of these points during lecture, but you should review them during the Lab.

Shared vs. Per-Iteration Variables

The classic trap that fools Java programmers — and comes up in Spacebar integration tests:

```
// ❌ Broken: var has function scope – one binding for the entire loop  
for (var i = 0; i < 3; i++) {  
    setTimeout(() => console.log(i), 0);  
}
```

```
// Output: 3 3 3. All three closures share the same `var i`.  
// By the time setTimeout fires, the loop has finished and i ≡ 3.
```

```
// ✅ Fixed: let has block scope – a new binding per iteration  
for (let i = 0; i < 3; i++) {  
    setTimeout(() => console.log(i), 0);  
}
```

```
// Output: 0 1 2. Each iteration gets its own `i`.
```

44

await Only Suspends the Enclosing Function

```
async function handleRequest(req, res) {  
  const user = await db.findUser(req.userId);    // suspends THIS function  
  // ...                                          // the server is NOT frozen  
}  
  
// Meanwhile, on another async function running in the same process:  
async function handleOtherRequest(req, res) {  
  // This runs while handleRequest is awaiting the database  
  // The event loop picked it up from the queue  
  const data = await db.findChannel(req.channelId);  
}
```

Two requests arrive simultaneously. Both handlers **await** a database query. Both see the same "before" state.

45

this or That?

In Java, `this` inside any method always refers to the enclosing object instance. In JavaScript, `this` is set by **how a function is called**, not where it is defined.

```
class Button {  
  label = "Submit";  
  function setupBad() { // Regular function – `this` is set by the caller  
    document.addEventListener("click", function() {  
      console.log(this.label); // ❌ `this` is the caller, not the Button  
    }); }  
  function setupGood() { // Arrow function – `this` is captured from enclosing  
    document.addEventListener("click", () => {  
      console.log(this.label); // ✅ `this` is still the Button instance  
    }); } }
```

46

The rule: use **arrow functions** for event handlers — they inherit `this` from the context.

Using an Agent to Understand Code

Being vague vs. precise

Agentic Tools: Specificity Matters!

Large language models are trained over the entire Internet

- Without specificity, they revert to doing "average" things. Sometimes that causes **hallucinations**.
- The more you can give them adequate *context*, the more likely you'll get good results!

The better you get at specifying "good prompts", the more likely you will get what you want!

Let's try this!

Vague → "How does async work in JavaScript?"

JavaScript is single-threaded, so async code lets long-running operations (network requests, file I/O, timers) happen without blocking that one thread. Three layers build on each other:

Callbacks (the original mechanism): pass a function to be invoked later, when the operation finishes.

```
fs.readFile('file.txt', (err, data) => { ... });
```

Promises: an object representing a value that will exist in the future, in one of three states — pending, fulfilled, or rejected. They chain instead of nesting.

```
fetch(url)
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

async/await: syntax sugar over promises that lets async code read like synchronous code. `await` pauses the function (not the thread) until the promise settles.

```
async function getData() {
  try {
    const res = await fetch(url);
    const data = await res.json();
    return data;
  } catch (err) {
```

Precise →

"Find a route handler in `src/api/routes/` that has multiple sequential `await` calls. For the first `await` in the handler: (1) trace exactly what happens to the function's stack frame when execution hits `await`; (2) describe what the Node.js event loop is doing while the function is suspended; (3) explain what triggers the function to resume."

Vague vs. Precise Prompts

The demos above all follow the same structure. Write it down — you'll use it for the rest of the semester.

	VAGUE PROMPT	PRECISE PROMPT
Scope	The whole language / concept	A specific file, function, or line
Output	Textbook explanation	Cited, verifiable code excerpt
Verification	You can only nod	You can open the file and check
Follow-up	"Tell me more"	"Is there a bug if this is wrong?"
Learning	Theory	Theory + consequence

Precise prompts also produce shorter outputs. When the agent's scope is bounded to one function, it can't fill space with tangential information. Shorter, cited output is easier to verify and harder to hallucinate into.

Wrap-up: What We Covered

CONCEPT	THE JAVA ASSUMPTION	THE JS/TS REALITY
Type system	Compile-time + runtime enforcement	TS erases at runtime; AJV fills the gap
Object model	Class blueprint, stable <code>this</code>	Property bags; <code>this</code> is caller-determined
Closures	Anonymous inner classes (rare)	Everywhere; <code>var</code> vs <code>let</code> scoping trap
Execution model	Thread per request	Single thread + event loop; never block
Async	Blocking I/O	<code>await</code> suspends only the enclosing function

Key skill: vague prompt → generic output; precise prompt grounded in a real file → verifiable, citeable output.

Next lecture: why agents get things wrong, and how to catch it before it ships.

Questions?

`zives@engineering.upenn.edu` · `sharry@engineering.upenn.edu`