Type Inference and Async Programming

Please seat at tables with at least 3 people, ideally 4!
Only use Zones ABC. No DEF!

Kris Jordan / August 26, 2024 / COMP423 / Class 03

Structural Typing

As opposed to *reified, nominal* typing, the *shape* of values determines valid uses.

- Functions can conform to function type interfaces *without declaring they do!*
    - Contrast this with Java's `class/implements` relationship: just defining the methods of an interface is not enough, a class must also state it implements an interface in order to be used where the interface is expected.
- This is very powerful for enabling common usage scenarios in TypeScript that exploit anonymous functions.
    - As seen on the previous slide!
- An **interface *specifies the shape* of a value** and *any value* that conforms to that shape is automatically considered to be an implementation of the **interface**.

Type Inference

TypeScript can use context to infer types without explicit declarations.

- What is the type of x in the following statement:
  ```
  let x = "hello, world";
  ```
- What is the return type of the following function:
  ```
  let f = (x: number) => { return x * 2; }
  ```

Type Inference (2/2)

Consider the following function type interface and function signature:

```
interface Transform {
    (num: string): number;
}

function map(xs: string[], f: Transform): number[] { /** Elided */ }
```

- In the bolded anonymous arrow function, given the above definitions, what is `x`'s type and what is the arrow function definition's return type?

  ```
  map(["A","B","C"], (x) => { return parseInt(x, 16); });
  ```

The Right-hand Example is Idiomatic Thanks to Inference

```
interface Predicate {
    (num: number): boolean;
}

function filter(xs: number[], test: Predicate) {
    let result: number[] = [];
    for (let x of xs) {
        if (test(x)) {
            result.push(x);
        }
    }
    return result;
}

console.log(
    filter(
        [1, -2, 3, -4, 5],
        (x: number): boolean => {
            return x < 0;
        }
    )
);
```

```
interface Predicate {
    (num: number): boolean;
}

function filter(xs: number[], test: Predicate) {
    let result: number[] = [];
    for (let x of xs) {
        if (test(x)) {
            result.push(x);
        }
    }
    return result;
}

console.log(
    filter(
        [1, -2, 3, -4, 5],
        (x) => {
            return x < 0;
        }
    )
);
```

A note on syntactical sugar...

- TypeScript (and JavaScript) offer syntactical sugar for writing arrow functions definitions more concisely. Consider this arrow function:

```
const t: Transform = (x: string): int => { return parseInt(x, 16); }
```

- If TypeScript has context to infer types, you can omit param and return types:

```
const t: Transform = (x) => { return parseInt(x, 16); }
```

We recommend this syntax as you write your first TS/JS programs.

- When the *body* of an arrow function contains *only one statement*, and it is a return statement, then you can omit the curly braces and return keyword:

```
const t: Transform = (x) => parseInt(x, 16);
```

- When an arrow function has only one parameter and you do not need to specify its type, you can omit the parameter list's parentheses.

```
const t: Transform = x => parseInt(x, 16);
```

Software Engineering Lessons in TypeScript

- Better tools, **better teams**!
  - Adding static type annotations to the JavaScript language, alongside tools for checking, compiling, and niceties in IDEs like VSCode, the TypeScript language enabled software engineering teams to collaborate on large code bases more confidently and productively.
- Better automated verification, **better user experiences**!
  - 2017 study out of University College London and Microsoft Research found **15% of JavaScript bugs that made it to production systems would have been found at compile time if using TypeScript**, or equivalent*
- **Layered system design can add stronger properties above more primitive layers**
  - TypeScript is a superset of JavaScript! (All programming languages ultimately transform down to machine code... most add semantic properties stronger than assembly's ability to.)
  - Before building a new system ask: **can we build a new layer on an existing system instead?**

* https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/gao2017javascript.pdf

Runtime Models

- **Single Thread Blocking Sequential Model** - Each operation "blocks" progress in the thread of execution until it completes.
    - This is the runtime model you are most comfortable with and currently your default mental model.
- **Multithreaded Model**- Expensive operations, in terms of time complexity, are moved to separate threads, as you saw in 301. Each Thread is blocking/ sequential and maintains its own call stack, but they share the same heap. Synchronization and memory safety is a real challenge, as you saw in 301!
- **Asynchronous Event Model** - Expensive operations are added to background queues that do not block execution in the main thread. These backgrounded tasks register **callback functions** that are called sometime after the operation completes

Example Async Function: setTimeout

- Schedules a callback function to run after a specified delay in milliseconds.
- After the delay has passed, the callback function is called.
- `setTimeout` is a **non-blocking** function!

```
interface TimeoutCallback {
    (): void;
}

const setTimeout = (cb: TimeoutCallback, ms: number) => {
  /** Implementation elided. */
};

// Example Usage: Print "Hello!" 1 second from now...

setTimeout(() => { console.log("Hello!"); }, 1000);
```

What is the printed output?

Form pairs (or trios at tables of 3) and whiteboard...

```
const one = (): void => {
  console.log(`T-1 seconds`);
  setTimeout(zero, 1000);
};

const zero = (): void => {
  console.log(`T-0 seconds`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log("BOOM!");
```

Async Event Loop

Pseudo-code Intuition

- The JavaScript run-time's main thread has event loop logic like the code to the right:
- Your program's initial execution as the first "task" that's added to the "Task Queue"
- When async operations are encountered, they are handled by a background system until completed ("ready" for further processing).
- Completed/Ready operation callback "tasks" are added to the task queue.

```
while (true) {
    // 1. Check for tasks to run
    if (taskQueue is not empty) {
        // 2. Run the next task
        task = taskQueue.dequeue();
        execute(task);
    }

    // 3. Check for timed tasks (e.g., setTimeout)
    if (any timed tasks are ready) {
        // 4. Move ready tasks to the task queue
        taskQueue.enqueue(ready tasks);
    }

    // 5. Wait briefly if nothing to do
    if (taskQueue is empty and no events) {
        sleep(very_briefly);
    }
}
```

```
const one = (): void => {
  console.log(`T-1`);
};

conso                 ..");
setTi
conso
```

Associate one with function object on heap.

Call Stack

Heap

Globals

one | id:0 | id:0 - fn lines 1-3

Output

Event/Task Queue

Timer Management Subsystem

Current Time: 5:00:00pm

```
const one = (): void => {
  console.log(`T-1`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log( );
```

Output sent to stdout...

Call Stack

Globals

one | id:0

Heap

id:0 - fn lines 1-3

Output

Launch in...

Event/Task Queue

Timer Management Subsystem

Current Time: 5:00:00pm

```
const one = (): void => {
  console.log(`T-1`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log("BOOM!");
```

Timer established in Timer Management System

Call Stack

Globals

one | id:0

Heap

id:0 - fn lines 1-3

Output

Launch in...

Event/Task Queue

Timer Management Subsystem

At 5:00:01pm call function id:0

Current Time: 5:00:00pm

```
const one = (): void => {
  console.log(`T-1`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log("BOOM!");
```

Output sent to stdout...

Call Stack

Globals

one | id:0

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

Event/Task Queue

Timer Management Subsystem

At 5:00:01pm call function id:0

Current Time: 5:00:00pm

```
const one = (): void => {
  console.log(`T-1`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log("BOOM!");
```

Our program's code evaluation has completed and the frame of the call stack goes away!

Call Stack

Heap

id:0 - fn lines 1-3

Notice, though, our function definition lives on because it's still referenced by a Timer

Output

Launch in...

BOOM!

Event/Task Queue

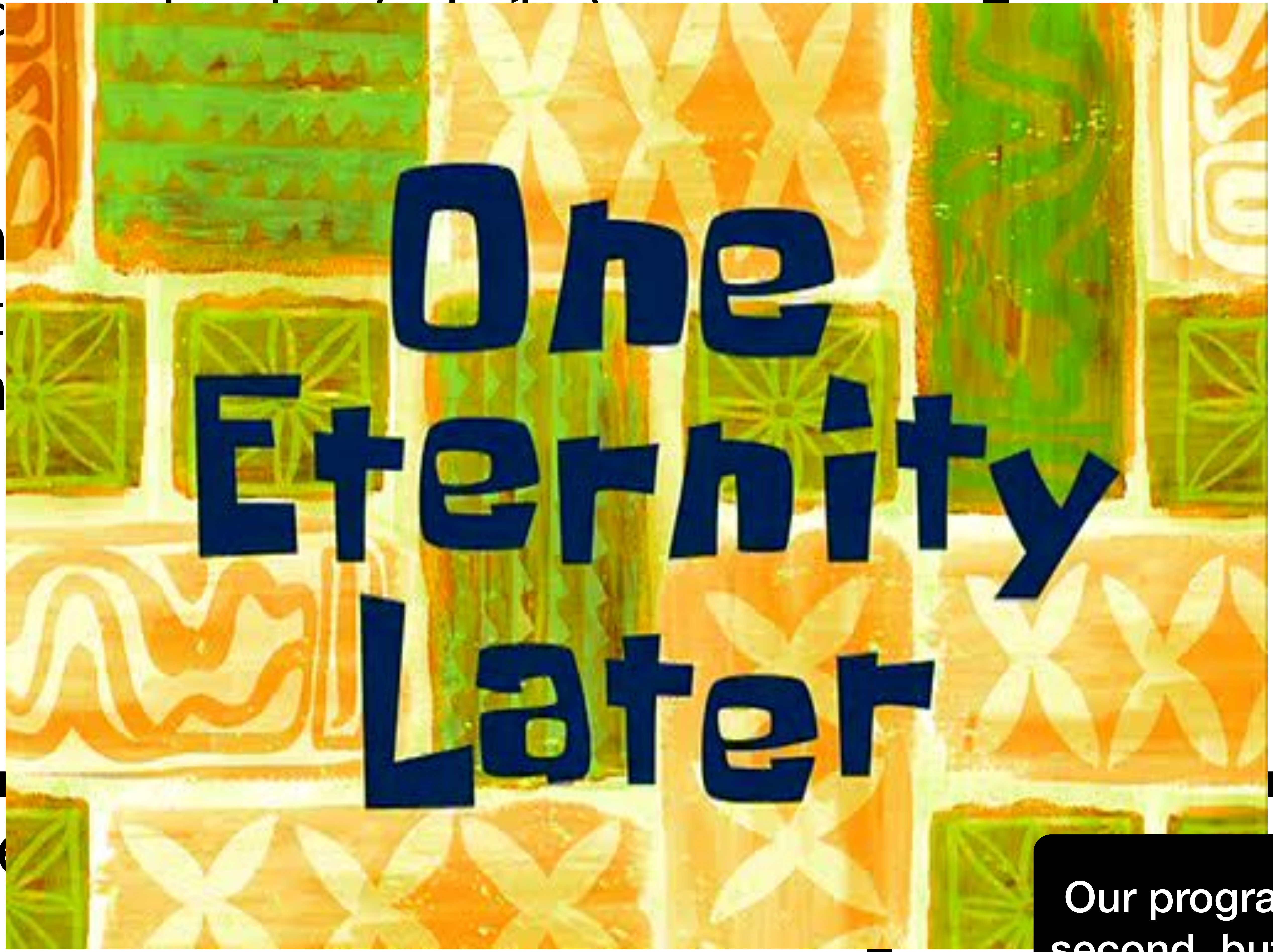
Timer Management Subsystem

At 5:00:01pm call function id:0

Current Time: 5:00:00pm

```
const one = (): void => {
  c
};

con
set
con
```



Call Stack

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

Eve

tem

Our program was "sleeping" for one second, but now we imagine the time is one second later....

Current Time: 5:00:01pm

```
const one = (): void => {
  console.log(`T-1`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log("BOOM!");
```

Call Stack

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

Event/Task Queue

`Call function id:0()`

Tim… …ystem

This timer is READY to run, so it adds a task to the queue...

At 5:00:01pm call function id:0

Current Time: 5:00:01pm

```
const one = (): void => {
  console.log(`T-1`);
};

console.log("Launch in...");
setTimeout(one, 1000);
console.log("BOOM!");
```

Call Stack

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

The Runtime Model's event loop will check this queue and see that it is not empty...

Eve

`Call function id:0()`

Timer Management Subsystem

Current Time: 5:00:01pm

```
const one = (): void => {
  console.log(`T-1`);
};

con
set
con
```

The Task's callback is processed by adding a frame to the stack and jumping into the function definition.

Call Stack

one

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

Event/Task Queue

Timer Management Subsystem

Current Time: 5:00:01pm

```
const one = (): void => {
  console.log(`T-1`);
};

con
set
con
```

This line of code is evaluated and output is logged to stdout.

Call Stack

one

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

T-1

Event/Task Queue

Timer Management Subsystem

Current Time: 5:00:01pm

```
const one = (): void => {
  console.log(`T-1`);
};

co
se
cor
```

The end of the void function is reached, so it returns. This leaves the stack empty and no outstanding tasks so the program exits.

Call Stack

Heap

id:0 - fn lines 1-3

Output

Launch in...

BOOM!

T-1

Event/Task Queue

Timer Management Subsystem

Current Time: 5:00:01pm

What is the printed output?

```
const countdown = (start: number): void => {
  console.log(`T-${start} seconds`);
  if (start > 0) {
    setTimeout(
      () => {
        countdown(start - 1)
      },
      1000
    );
  }
};

console.log("Launch in...");
countdown(1);
console.log("BOOM!");
```

Promise-based Model

Promises offer a more modern, object-based take on async callbacks

- Most modern JavaScript/TypeScript non-blocking asynchronous functions return *Promise* objects rather than expect callbacks directly
  - Promises offer looser coupling and composability versus primitive async callback APIs like setTimeout
- As you read, you can use the `.then` method to register a callback function with a Promise object.
- Let's try using `fetch`, which carries out the "expensive"/slow operation of going out to the internet and downloading a resource. Fetch returns a Promise object.

In a `node` REPL in the DevContainer Terminal:

```
let request = fetch("http://worldtimeapi.org/api/timezone/America/New_York");
```

request is a Promise that represents the async execution of an API request

```
let json = request.then((response) => { return response.json(); });
```

When the request promise resolves, response represents metadata about the HTTP request/response.

json is a Promise that represents async execution of downloading the body (data) of the API request

```
json.then((data) => { console.log(data) });
```

Finally, we can log the data out which contains the data about the current date and time.

Contrast with async/await

```
const main = async (): Promise<void> => {
  const response = await fetch("http://worldtimeapi.org/api/timezone/America/New_York");
  const data = await response.json();
  console.log(data);
};

main().then(() => {
  console.log("--- DONE ---");
});
```

Functions marked async can use the await keyword to asynchronously "await" the result of Promises. This gives the ergonomics of writing synchronous blocking code but results in asynchronous, sequential semantics. *We will dive more into this Friday!*