# Typescript

Christine Imogu & Helen Yang

# Reminders

- Milestone 2 (MVP) is due **NEXT WEDNESDAY**, Jan 24 at 6PM sharp.
  - https://weblab.mit.edu/about/#milestones
  - ALL milestones are required to get credit or compete
- Read the pinned Piazza posts and check out weblab.is/home
  - Skeleton code (typescript version also available)
  - Lots of other useful resources!
- Friday: How to deploy
  - Essential lecture, your MVP must be deployed!

# Sponsored by

# Mobi

# Primitive Data Types

```
number
```

```
boolean
```

```
string
```

```
object
```

```
undefined
```

```
null
```

# Primitive Data Types

number

4

boolean

true

string

"bruh"

object

```
{
  property: 4;
}
```

undefined

undefined

null

null

# Javascript is dynamically typed

```
let five = 5;
five = "5";
five = undefined;

console.log(five)
// prints "undefined" 😭
```

- **Dynamic typing:** Types are only associated with values, so a variable's type can change during execution.
- Does not check types when compiling which allows for fast programming
- **How to make sure that "five" is always a number?**

# Typescript is statically typed

```
Type 'undefined' is not assignable to type 'number'. ts(2322)
View Problem (⌥F8)    No quick fixes available
```

```
let five: number = 5;
five = undefined;

console.log(five);
```

# Typescript is statically typed

```
Type 'undefined' is not assignable to type 'number'. ts(2322)
View Problem (⌥F8)    No quick fixes available
```

```
let five: number = 5;
five = undefined;

console.log(five);
```

Cannot run code
without fixing error!


Source: https://www.reddit.com/r/ProgrammerHumor/comments/a1hghl/dynamic_vs_static_typing/

# Typescript, formally

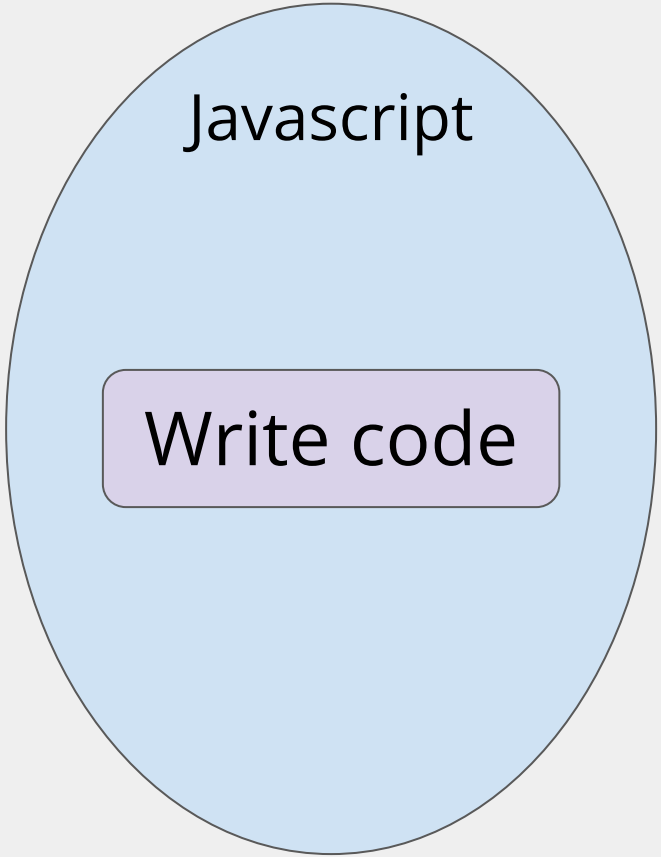
- Language built on top of “Vanilla” Javascript that enforces **static typing**
- Validates that your code works at compile-time
- Will save your life when debugging

JS + 

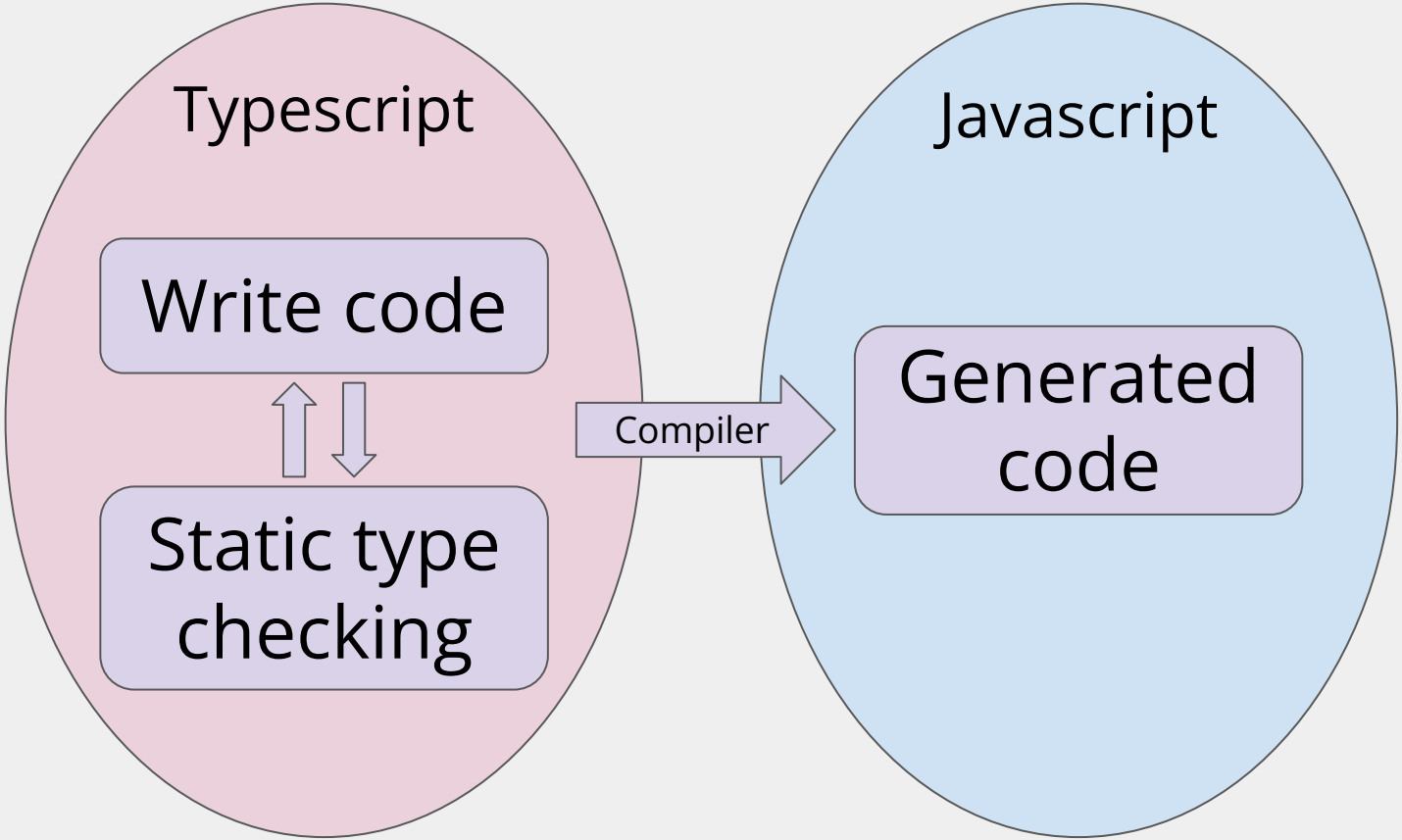
- **Interfaces**
- **Static Typing**
- **Generic Types**
- and more...

= TS

# Javascript offers no static type checking



# Typescript adds a cushion to your code



# How to type

```
let x: string = "henlo";
let y: string;
let z = "hello";
```

# How to type

```
let x: string = "henlo";
let y: string;
let z = "hello";

x = 4;
y = 4;
z = 4;
```

# Why do we care about static typing?

Here are some examples of issues that static typing can catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior - the "billion dollar mistake"
- Overloaded operators (e.g. addition, comparison)

# Why do we care about static typing?

Here are some examples of issues that static typing can catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior - the "billion dollar mistake"
- Overloaded operators (e.g. addition, comparison)

```
type ProfileProps = {
  userId: string;
};

const Profile = (props: ProfileProps) => {
  props.password;
```

# Why do we care about static typing?

Here are some examples of issues that static typing can catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior - the "billion dollar mistake"
- Overloaded operators (e.g. addition, comparison)

```
function callString(): string {
  return "hi";
}

const calledString = "hello";

let newString: string = callString;
```

# Why do we care about static typing?

Here are some examples of issues that static typing can catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior - the “billion dollar mistake”
- Overloaded operators (e.g. addition, comparison)

```
function printString(s: string | null) {
  console.log(s ?? "no string!");
}
```

# Why do we care about static typing?

Here are some examples of issues that static typing can catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior - the [“billion dollar mistake”]
- Overloaded operators (e.g. addition, comparison)

```
'5' + 4 == "54";
```

```
let sum: number = "5" + 4
    Type 'string' is not assignable to type 'number'. ts(2322)
    let sum: number
    View Problem (⌥F8)    No quick fixes available
```

# Why do we care about static typing?

Here are some examples of issues that static typing can catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior - the "billion dollar mistake"
- Overloaded operators (e.g. addition, comparison)

These problems may seem "obvious", but more complex codebases = higher chance of introducing bugs!

# Questions?

# Typescript with primitive types

```
1   let message: string = "Hello world!";
2   message = 1;
```

```
Type 'number' is not assignable to type 'string'. (2322)
Peek Problem (⌥F8)    No quick fixes available
```

- string, boolean, number

# More types

## Arrays

```
let message: string[] = ["1", "2", "3"];
```

```
let message: Array<string> = ["1", "2", "3"];
```

```
const example: Array<string | number> = [1, 2, "three"];
```

## Enums

```
type Color = "Red" | "Green" | "Blue";

let c : Color = ""
```

Blue
Green
Red

# Types in general

Declare new type “myType”

```
type myType = {
  myProperty: string;
};

const variable: myType = { myProperty: "" };
```

“myType” has a property “myProperty”

Initialize new variable of type “myType”

# Define your own type

- Define each property and its type
- Denote optional params with ?

```
type User = {
  _id: string;
  name: string;
  is_admin?: boolean;
};

const user: User = {
  _id: "555",
  name: "Kenneth",
};
```

# Use types with other types

```
type User = {
  _id: string;
  name: string;
}

type Message = {
  sender: User;
  content: string;
}

type ChatData = {
  messages: Message[];
  recipient: User;
}

export {User, Message, ChatData};
```

```
type User = {
  _id: string;
  name: string;
  is_admin?: boolean;
};

const user: User = {
  _id: "555",
  name: "Kenneth",
};

const user2: User = {
  _id: "123",
  name: "Nick",
};

const users: User[] = [user, user2];
```

# Extend types

```typescript
type User = {
    _id: string;
    name: string;
    is_admin?: boolean;
}

type UserLogin = {
    _id: string;
    name: string;
    is_admin?: boolean;
    password: string;
}
```

Here's a less repetitive way to write the left:

```typescript
type User = {
  _id: string;
  name: string;
  is_admin?: boolean;
};

type UserLogin = User & { password: string };

const userLogin: UserLogin = {
  _id: "555",
  name: "Kenneth",
  password: "yeahyouthought",
};
```

# Typed functions

- You need to declare the type for function parameters and output!
- So users know exactly what type gets passed in and what type comes out.
- Function signature looks like (paramName: **paramType**): **returnType** => { }

```
1  const getComments = (id) => {
2    return [];
3  };
```

Javascript version:
a bit ~mysterious~

```
const getComments = (id: string): Comment[] => {
  return [];
};
```

TypeScript version:
a lot clearer what's going on!

# Typed functions

- Use with built in React events as well

```
const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
  setValue(event.target.value);
}
```

# Typed async functions

JavaScript version:
again, mysterious

```
const getComments = async (id) => {
```

TypeScript version!

```
const getComments = async (id: string): Promise<Comment[]> => {
```

Tells TypeScript this is an async function

Tells TypeScript that this function takes in a string

Tells TypeScript that this function returns a Promise that resolves to a list of Comments

# First-class functions

```
type CommentProps = {
  _id: string;
  creator_name: string;
  content: string;
  handleAdd: () ⇒ void;
};
```

Functions in TypeScript are treated like any other variable! So you can add them as a property :)

# Chatbook example: Typed props and state

```
type ProfileProps = {
  userId: string;
};

const Profile = (props: ProfileProps) => {
  const [user, setUser] = useState<User | undefined>(undefined);
  const [catHappiness, setCatHappiness] = useState(0);
```

# Chatbook example: Define types for chats

Export to reuse your custom types in other files

```
type User = {
  _id: string;
  name: string;
}

type Message = {
  sender: User;
  content: string;
}

type ChatData = {
  messages: Message[];
  recipient: User;
}

export {User, Message, ChatData};
```

You can then import to use your custom types in a different file!

```
import React from "react";
import { Message } from "../pages/Chatbook";
```

# Easily integratable with your projects!

- Works well with defining React prop/state types
- Can integrate slowly/partially into your projects; you don't need to write entirely in Javascript or entirely in Typescript
- **Recall**: TypeScript is a superset of JavaScript, which means that any valid JavaScript code is also valid TypeScript code.

# Typescript Setup Info

How to add typescript to an existing React project:

- https://www.sitepoint.com/how-to-migrate-a-react-app-to-typescript/
- (or just use the Typescript skeleton we provided for you)

Typescript Config Settings:

- Can be changed in `tsconfig.json`
- Describe “how strict” Typescript should be

# Quick Look at Catbook!

https://github.com/weblab-workshops/catbook-react/tree/main

# Resource: Typescript playground

https://www.typescriptlang.org/play

# Resource: Web Lab Typescript skeleton

https://github.com/weblab-workshops/skeleton/tree/typescript
("typescript" branch in the skeleton repo)