COMP1531

✅ Correctness - Static Verification

Lecture 3.3

Author(s): Hayden Smith

(Download as PDF)

In This Lecture

- **Why? 🤔**
  - The best time to improve safety safety is before the code runs
- **What? 📰**
  - Type Safety
  - Typescript
  - Examples

🌳 Disclaimer: Environment Change

Beginning from lecture 3.3, we will be working inside the `env2` folder with the lecture code. To "run" code from lectures slides further on you will need to ensure you have a similar environment.

Don't stress, though! For your labs in week 4 onwards, along with iteration 2 & 3, we have setup your project to contain everything you need.

😬 Unexpected Input

Sometimes we write a really nice function like this. And everyone uses it correctly.

```
1 function manyString(repeat, str) {
2   let outString = '';
3   for (let i = 0; i < repeat; i++) {
4     outString += str;
5   }
6   return outString;
7 }
8 console.log(manyString('hello ', 5));
```

3.3_many_string_rude.js

😬 Unexpected Input

Sometimes we write a really nice function like this. And everyone uses it correctly.

```
1 function manyString(repeat, str) {
2   let outString = '';
3   for (let i = 0; i < repeat; i++) {
4     outString += str;
5   }
6   return outString;
7 }
8 console.log(manyString('hello ', 5));
```

3.3_many_string_rude.js

...right? 😳

😬 Unexpected Input

Wrong! Users of your functions will often make mistakes using them.

```
1 function manyString(repeat, str) {
2   let outString = '';
3   for (let i = 0; i < repeat; i++) {
4     outString += str;
5   }
6   return outString;
7 }
8 console.log(manyString('hello ', 5));
```

3.3_many_string_rude.js

This program prints out nothing... **the worst mistake a program makes is one that does not cause an error.**

😬 Unexpected Input

How can we protect against this?

😬 Unexpected Input

We'll add a type check in during runtime to check the type being passed in.

```
1  function manyString(repeat, str) {
2    if (repeat instanceof 'number') {
3      console.error('repeat argument is not a number');
4      return undefined;
5    }
6    if (str instanceof 'string') {
7      console.error('str argument is not a string');
8      return undefined;
9    }
10   let outString = '';
11   for (let i = 0; i < repeat; i++) {
12     outString += str;
13   }
14   return outString;
15 }
16 console.log(manyString('hello ', 5));
```

3.3_many_string_runtime.js

 Type Safety

- Preventing mismatches between the actual and expected type of variables, constants and functions
- C is type-safe*, as types must be declared and the compiler will check that the types are correct
- Javascript, on its own, is not type-safe. Everything has a type, but that type is not known till the program is executed.

🔫 Type Safety In Javascript

- The solution we saw previously is what we would refer to as improving software safety **dynamically** - by "catching" issues at runtime.
- However, rather than dynamically checking for certain errors, it is always better if errors can be detected **statically**.

We need a way to check for correct types statically in Javascript.

🔥 Typescript

Typescript is a language built on top of Javascript. It's job is to check the types in your program and outputs Javascript that is then run with node.

```
1 function sum(a: number, b: number) {
2   return a + b;
3 }
4 console.log(sum(1, 2));
```

3.3_mycode.ts

But how do I run this code?

🔥 Typescript

Typescript is another dependency we need to install:

```
npm install --save-dev typescript ts-node
```

🔥 Typescript

Running node With Typescript

Once this is installed, we can run our typescript code (e.g. `3.3_mycode.ts`) with the following command:

`node_modules/.bin/ts-node 3.3_mycode.ts`.

🔥 Typescript

Type Checking With Typescript

Whilst `ts-node` does some type checking, it also runs the code. It's useful to have a way to "type check without running" that also checks a bit more strictly.

`node_modules/.bin/tsc --noImplicitAny 3.3_mycode.ts`.

🔥 Typescript

In reality you would normally add both of these commands to your `package.json`

```
"scripts": {
  "ts-node": "ts-node",
  "tsc": "tsc --noImplicitAny"
}
```

🔥 Typescript

Now let's try and use `tsc` on a program that has type errors in it! Like our original program. But let's write it in **typescript**.

```ts
function manyString(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
}
console.log(manyString('hello ', 5));
```

3.3_mycode_broken.ts

`node_modules/.bin/tsc 3.3_mycode_broken.ts`.

Let's see what it outputs!

🔥 How To Typescript

Types are added to programs typically by putting the type name after a colon. We've seen that in our first example.

```
1 function sum(a: number, b: number) {
2   return a + b;
3 }
4 console.log(sum(1, 2));
```

3.3_mycode.ts

🔥 How To Typescript

Typescript doesn't require you to put types on **everything**. It will **infer types** that it can, but sometimes it's unable to.

Typescript doesn't know what `name` is, you need to give it a type!

```
1 function hello(name: string) {
2   console.log(name);
3 }
```

Typescript doesn't need to be told `name`'s type. It will figure it out:

```
1 const name = 'Hayden'
```

Examples Of Typing

Basics & Functions

The most basic 3 types in Typescript are `string`, `number`, and `boolean`. Sometimes we want functions to accept multiple of these.

Functions typically require all parameters to be explicitly typed. You can also type return types if needed, though usually Typescript will infer.

```
1 function hello(name: string): string {
2   return `Hello ${name}!`;
3 }
```

3.3_example_functions.ts

Examples Of Typing

Unions

```
1  function printIfReady(ready: boolean | number) {
2    if (ready === true || ready !== 0) {
3      console.log('Ready!');
4    }
5  }
6  printIfReady(1);
7  printIfReady(2);
8  printIfReady(0);
9  printIfReady(true);
10 printIfReady(false);
```

3.3_example_unions.ts

Examples Of Typing

Lists

```ts
function create10List(item: string | number) {
  const arr: Array<string | number> = [];
  for (let i = 0; i < 10; i++) {
    arr.push(item);
  }
  return arr;
}
```

3.3_example_lists.ts

Examples Of Typing

Aliases

```ts
type ListItem = string | number;

function create10List(item: ListItem) {
  const arr: ListItem[] = [];
  for (let i = 0; i < 10; i++) {
    arr.push(item);
  }
  return arr;
}
```

3.3_example_aliases.ts

Examples Of Typing

Optionals

```
// Note:
//   end?: number
// = end: number | undefined

function substring(str: string, start: number, end?: number) {
  let newString = '';
  const modifiedEnd = end || str.length;
  // ^ What about end ?? str.length
  for (let i = start; i < modifiedEnd; i++) {
    newString += str[i];
  }
  return newString;
}

console.log(substring('hayden', 0, 3));
console.log(substring('hayden', 2));
```

3.3_example_optionals.ts

Examples Of Typing

Objects

```ts
type Person = {
  name: string;
  age?: number;
  height?: number;
}

const person: Person = {
  name: 'Hayden',
};

person.age = 5;
```

3.3_example_objects.ts

Examples Of Typing

Literals

```ts
type visibility = 'Private' | 'Public';

function createChannel(name: string, visibility: visibility) {
  // Do things
}
```

3.3_example_literals.ts

Examples Of Typing

Any

any is a type that kind of makes typescript pointless. It's great for a stub and a "I will come back to this later"

```ts
// @ts-nocheck

function hello(name: any): any {
  return `Hello ${name}!`;
}

function substring(str: any, start: any, end: any) {
  return null;
}

type Person = any;
const person: Person = {
  name: 'Hayden',
};
```

3.3_example_any.ts

Examples Of Typing

A much more thorough array of types and their explanations can be found on the typescript website.

Type Safety

A summary of languages and their type safety:

- Languages with a non-optional built-in static type checking
  - C
  - Java
  - Haskell
- Languages with optional but still built-in static type checking
  - Typescript
  - Objective C
- Languages with optional external type checkers
  - Python
  - Ruby

🛠️ Migrating To Typescript

Included below are some of the steps to get typescript working:

1. Run `npm install --save-dev ts-jest @types/jest`. This allows Typescript to work with your `jest` files.
2. Run `npm install --save-dev @types/node`. This avoids complicated issues with the types of imports we do.
3. Add files `tsconfig.json` and `jest.config.js` with course-provided info in them.
4. Update `package.json` to include a script `tsc` that just runs `tsc --noImplicitAny` and `ts-node` that just runs `ts-node`.
5. You can also add `npm run tsc` as a step in pipelines (see next slide).

You aren't required to know these. You can just reference lecture code.

🛠️ Typescript With CI

Now we can add another step to our pipeline

```yaml
image: comp1531/basic:latest

stages:
  - checks

testing:
  stage: checks
  script:
    - npm run test

typecheck:
  stage: checks
  script:
    - npm run tsc
```

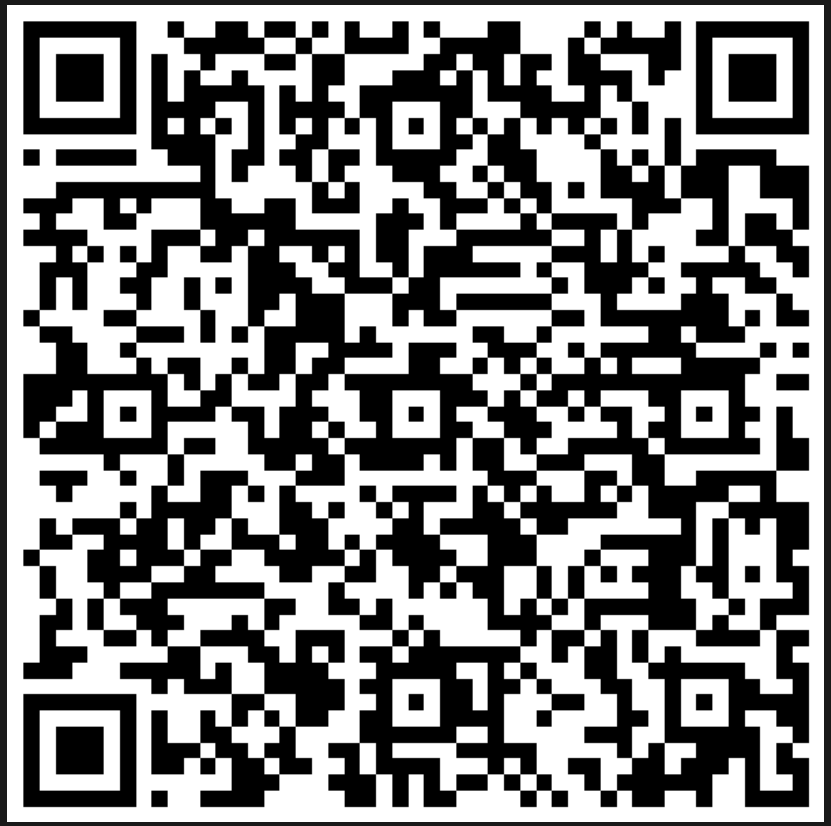
3.3_gitlab-ci_type.yml

🤗 Don't Stress!

All of the environmental setup or changes you've seen in this lecture will either be **done for you** or will be given to you with clear unambiguous instructions.

We don't expect you to all be experts in tweaking these environments.

👂 Feedback



Or go to the form here.