BOOTS: Beginner Object-Oriented TypeScript

By Greg Silber, Austin Cory Bart, and James Clause

Generated on Fri Aug 30 2024

Table of Contents

0) Introduction

0.1) Welcome

- What is this text about?
    - In this text we will continue to study the field of Computer Science delving deeper into core concepts and introducing new concepts and paradigms for producing well engineered software solutions
    - Specifically, we will study ***Objects and Object-Oriented programming*** techniques as well as various structures and algorithms to promote good software design.

Details

This text uses the language *Typescript*. *Typescript* is a free and open-source programming language that adds static typing and type annotations to Javascript.

Typescript is used widely and has become the most common language for developing applications for the web.

Through this text and its associated materials you will become familiar with the *Typescript* language and developing well engineered software solutions.

ChatGPT and Co-Pilot

- For this text, use of these tools is not recommended.
- This is not an arbitrary statement:
    - These tools, while impressive, are imperfect and often generate poor, inefficient, or downright incorrect code. In order to use these tools, one must already know how to program well in order to be certain that the generated code is correct.
    - In some cases, these tools may not be available, and thus it is important to learn to work without them.
- Once you achieve mastery, you will be able to use these tools in the future.

When used correctly they are powerful, but incorrectly they are dangerous.

Final Thoughts before we begin

Computer Science is hard until it is not. Be patient with yourself and be persistent. You are at the beginning of the journey, and that is the hardest part. As you work through this text, try to grasp the underlying concepts.

DON'T
PANIC!

0.2) Setup

This section helps you figure out how to setup your environment, install everything, and make sure your environment is correct.

If everything goes well, this will only take you about 20 minutes. But it is very normal to encounter issues if you are not used to this workflow. Don’t worry, you will be an expert soon!

Making web applications is super complicated, so we are going to be really pushy about your environment’s setup and the eventual structure of our web application. If this seems limiting, that’s the idea.
Please try to stay within the bounds we give you, as you experiment and try things out!

Do not skip steps.

Read error messages, and ask questions. Talk to humans as needed to get help, and use google intelligently.

Get Software

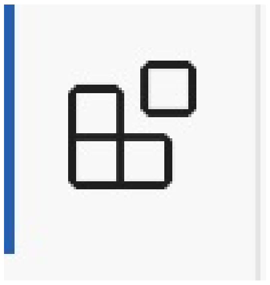
Get VSCode

Download [Visual Studio Code (https://code.visualstudio.com/download)](https://code.visualstudio.com/download)

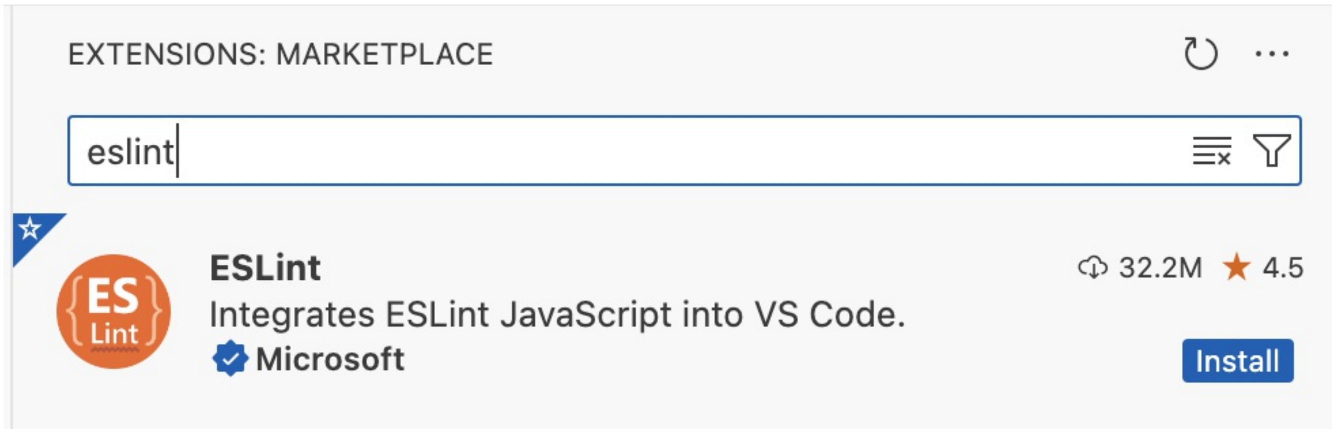
VS Code is an IDE (Integrated Development Environment) that you will program in.

When you have VS Code downloaded, open the application. You will need to install two extensions.

To open the extension menu, you can type `Ctrl+Shift+X` (Windows) or `Cmd+Shift+X` (Mac). There is also a navigation bar on the left side of your screen and you can click the extensions menu that looks like this:

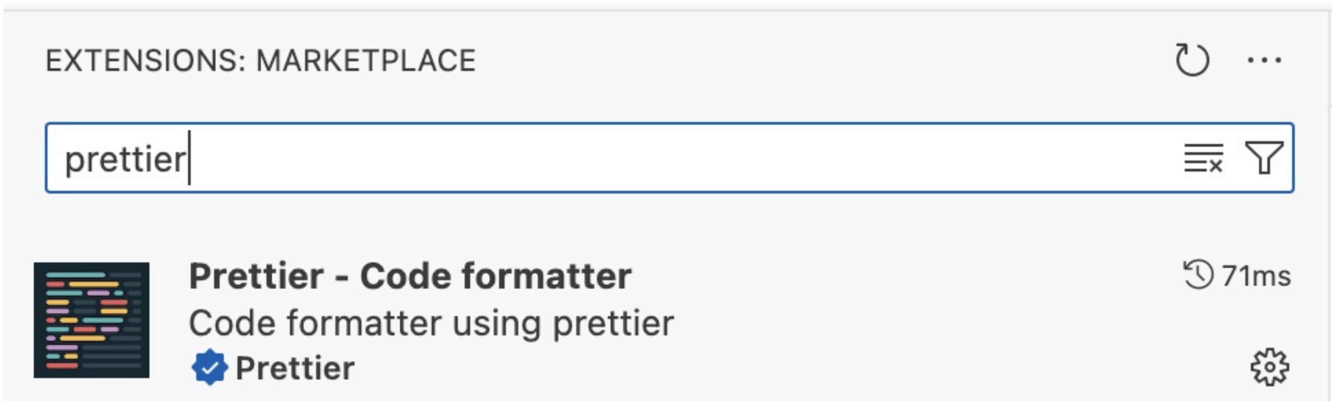


A search bar will appear at the top of the menu. Type `ESLint` and click install:



ESLint
Integrates ESLint JavaScript into VS Code.
Microsoft

Then, search for `Prettier` and click install:



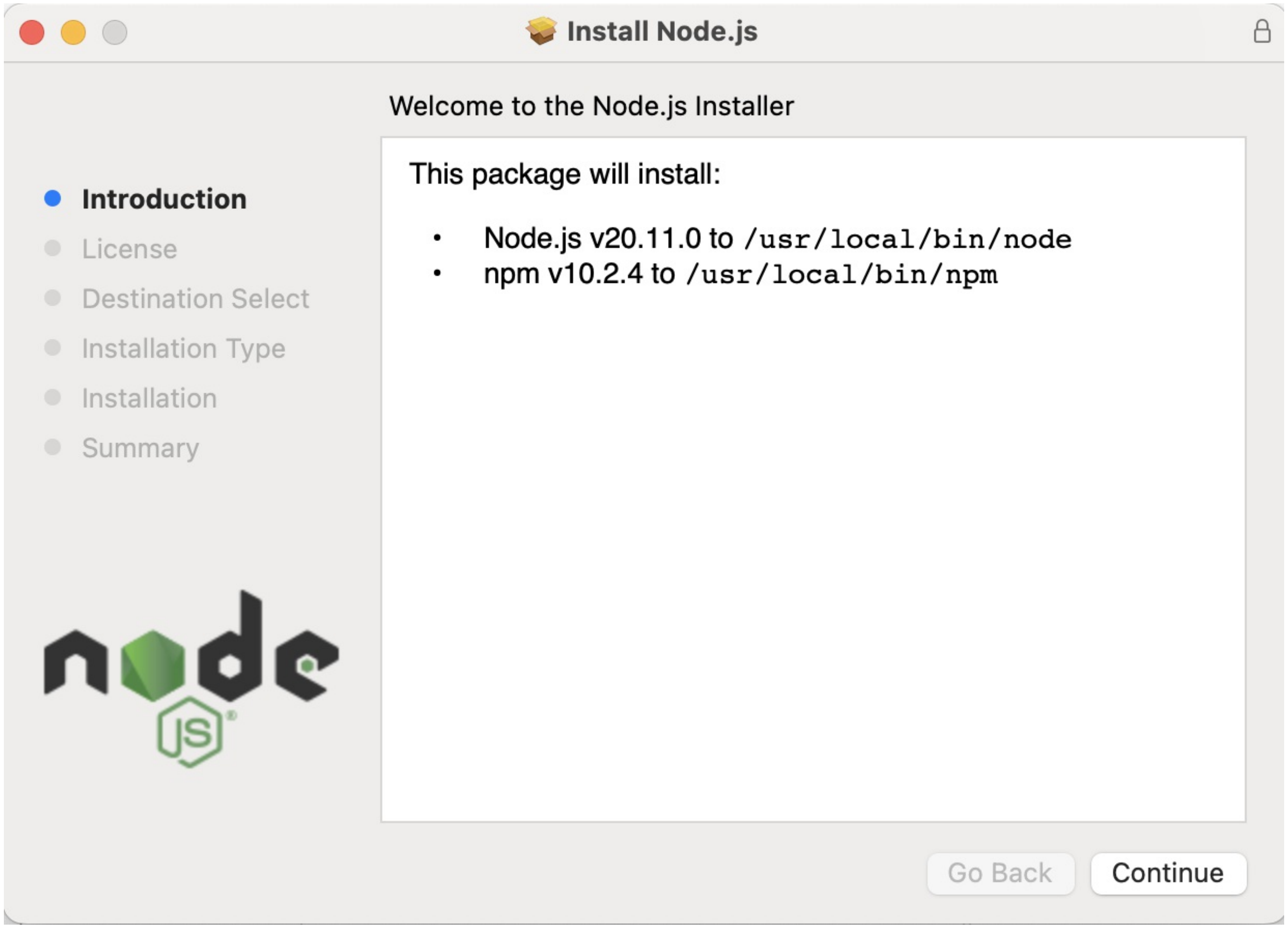
Make sure you have installed the extensions that are in the images above. These are the correct versions!

Get Node

Next, Download and Install [Node (https://nodejs.org/en/download/)](https://nodejs.org/en/download/)

You should use the installer for the most recent **LTS** version. The link will take you to the correct download page.

Once you have downloaded the installer, open it.



Welcome to the Node.js Installer

This package will install:

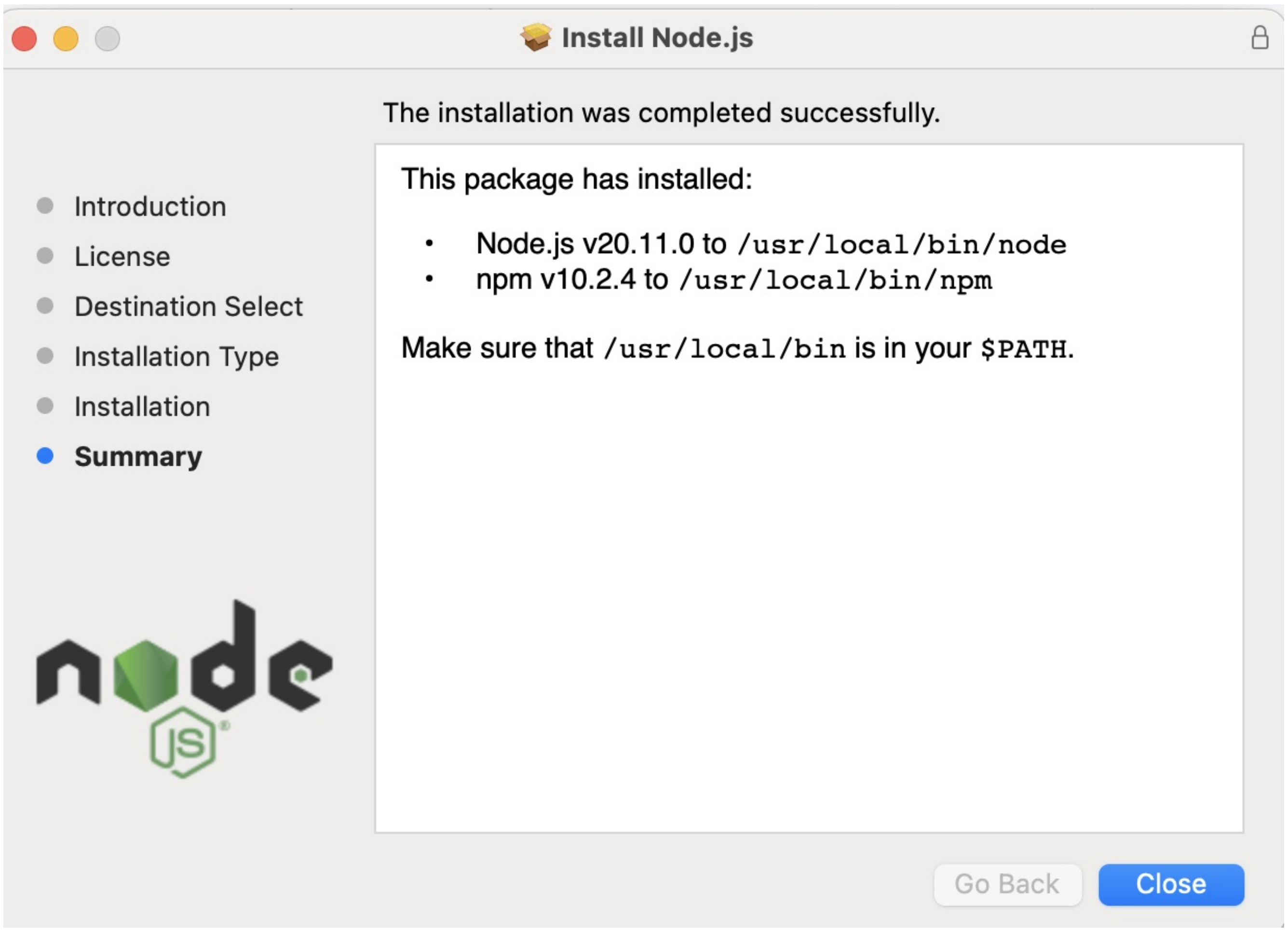
- Node.js v20.11.0 to `/usr/local/bin/node`
- npm v10.2.4 to `/usr/local/bin/npm`

You may notice that it is installing both Node and something called npm. Node Package Manager (npm) will make it easier
to manage, install, and update node packages. You need both of these!

The installer will ask you to select where you want to install the package; keep the default location that already appears.

- For Windows: `C:\Program Files\nodejs`
- For Mac: `/usr/local/bin/node`

Once the installer is finished, you should see this screen:



The installation was completed successfully.

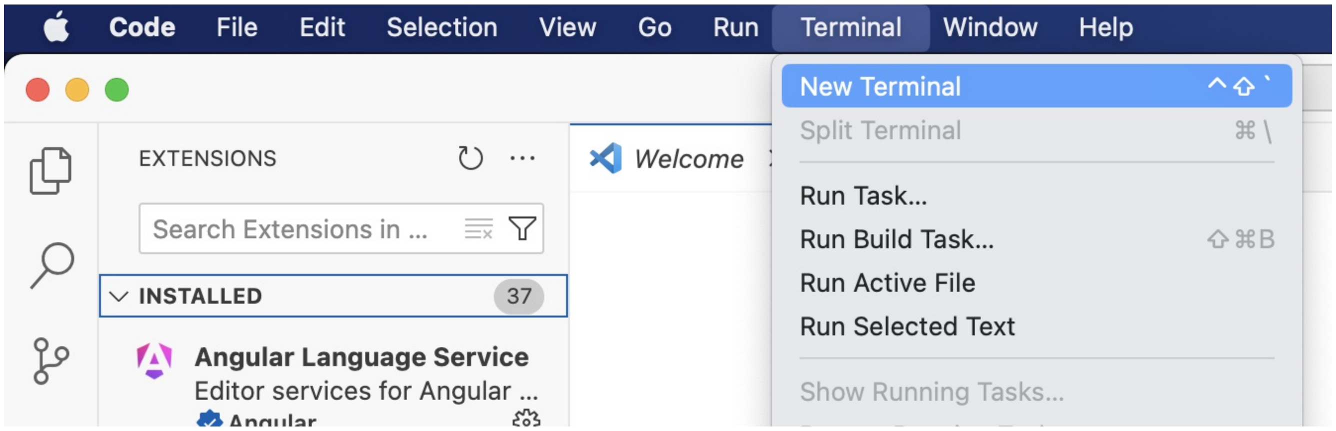
This package has installed:

- Node.js v20.11.0 to `/usr/local/bin/node`
- npm v10.2.4 to `/usr/local/bin/npm`

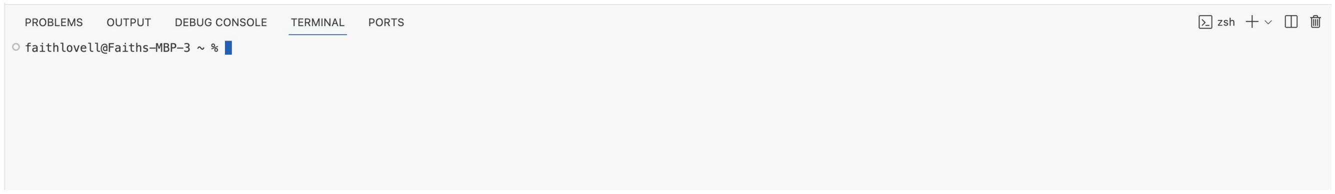
Make sure that `/usr/local/bin` is in your `$PATH`.

Now we need to verify that the installation was successful. Navigate back to VS Code and open a Terminal.

At the top of your VS Code window, click Terminal and then click New Terminal:



A new terminal will appear at the bottom of your screen. Its appearance can vary depending on your platform, but you might see something like this:



```
faithlovell@Faiths-MBP-3 ~ %
```

You should not be in any folders for this step! This should not be an issue if you have not opened a project in VS Code yet.

The blue box is the cursor where I can type commands. In the future, we will give instructions on what to write by writing boxes like this:

```
$ node --version
```

Note that you do not write the dollar sign (`$`); that just indicates the start of a new command. Sometimes folks will write an angle bracket (`>`) or some other symbol.

In this case, you need to type `node --version` and then press enter. The version that should appear is `v20.11.0` or newer. If an older version appears, you need to go back and install the **LTS version** ; some packages may only support the latest
LTS version of Node, so it’s better to fix it now.

If `node: command not found` appears, it means something went wrong with your installation. Check that the installer is properly finished. If it has, open the installer again and verify that the destination of your installation matches the ones listed above.

Once you have verified that Node was installed, enter the command (without the dollar sign):

```
$ npm --version
```

You should see `10.2.4` or later appear if everything is installed correctly.

Note: If Node and/or Git appear to not be working correctly or do not seem installed, completely quit and re-open
VS Code before troubleshooting. Sometimes VS Code will not recognize the install immediately.

{: .warning-title}

Get Git

Next, Download and Install Git (https://git-scm.com/downloads)

For Windows: You can download and use the installer. It should be straightforward, and you can move onto the next section once it finishes.

For Mac: There are a few options you can choose from on the download page.
We recommend installing Homebrew (https://brew.sh/), which you can use to install Git. There are several ways to install Homebrew, but here is the current easiest way that we know about. You will need to open up a Terminal, and then copy this (without the dollar sign!) and hit enter:

```
$ /bin/bash -c "$(curl-
fsSLhttps://raw.githubusercontent.com/Homebrew/install/HEAD
/install.sh)"
```

When it asks for password, type yours in. It'll output a lot of text, and then beneath a large list of tags, you see a purple arrow with `"Next Steps"`.

Then type `brew help` and hit enter, to confirm that Homebrew was correctly installed. Wait a second for it to spew words and then, when able, type:

```
$ brew install git
```

After that you should be done. Once you have verified that everything has been installed correctly, you are ready to move on!

Create GitHub Account

Next, [Create a GitHub Account (https://github.com/signup)](https://github.com/signup) (if you don’t already have one).

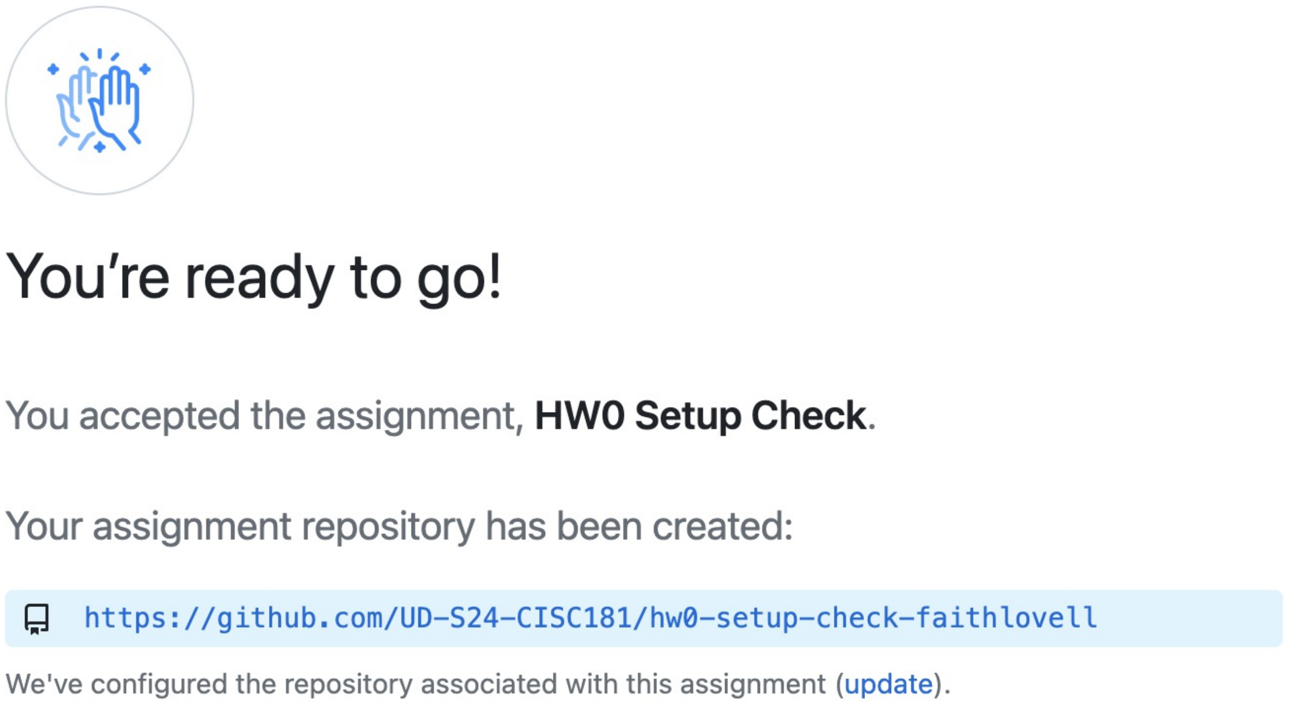
This account is an important part of your professional identity. You will use it to store your code, collaborate with others, and show off your work to potential employers. Choose a username that is professional and easy to remember. You are likely to use this account for a long time, so choose wisely!

Clone Assignment

Now that you have a programming environment, it is time to complete the first coding exercise to test everything.

On the original Canvas Assignment page that took you to this page, there should be a link to the GitHub Classroom assignment. Click on that link to go to the starter assignment on Github Classroom.

You may need to reload the page manually. The process should not take long. When the repository is ready, you should see
a new link:


You're ready to go!

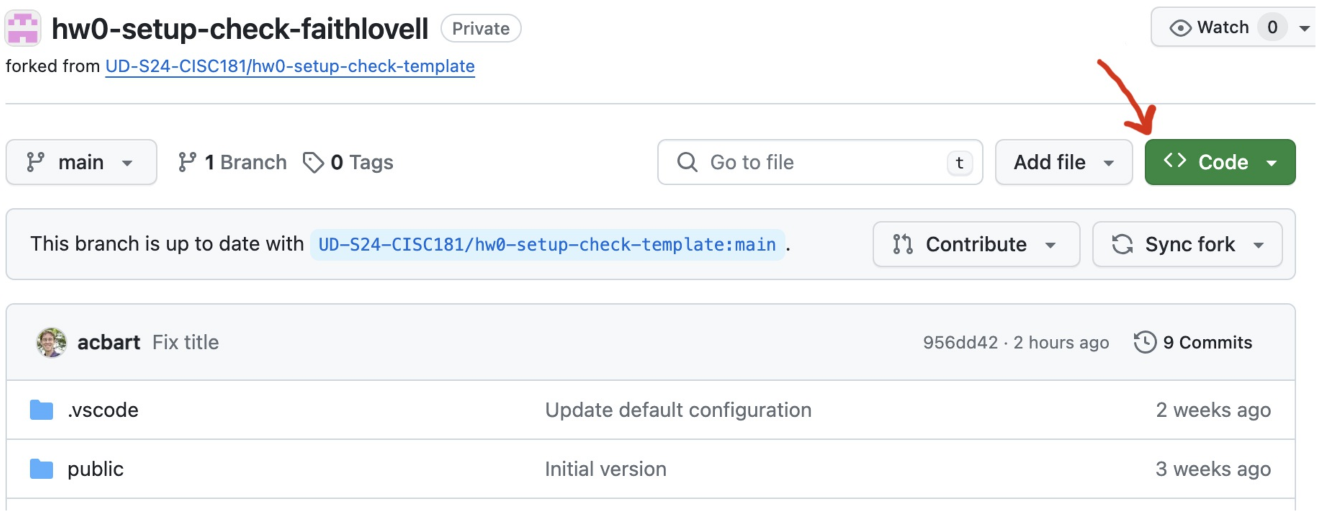
You accepted the assignment, **HW0 Setup Check**.

Your assignment repository has been created:

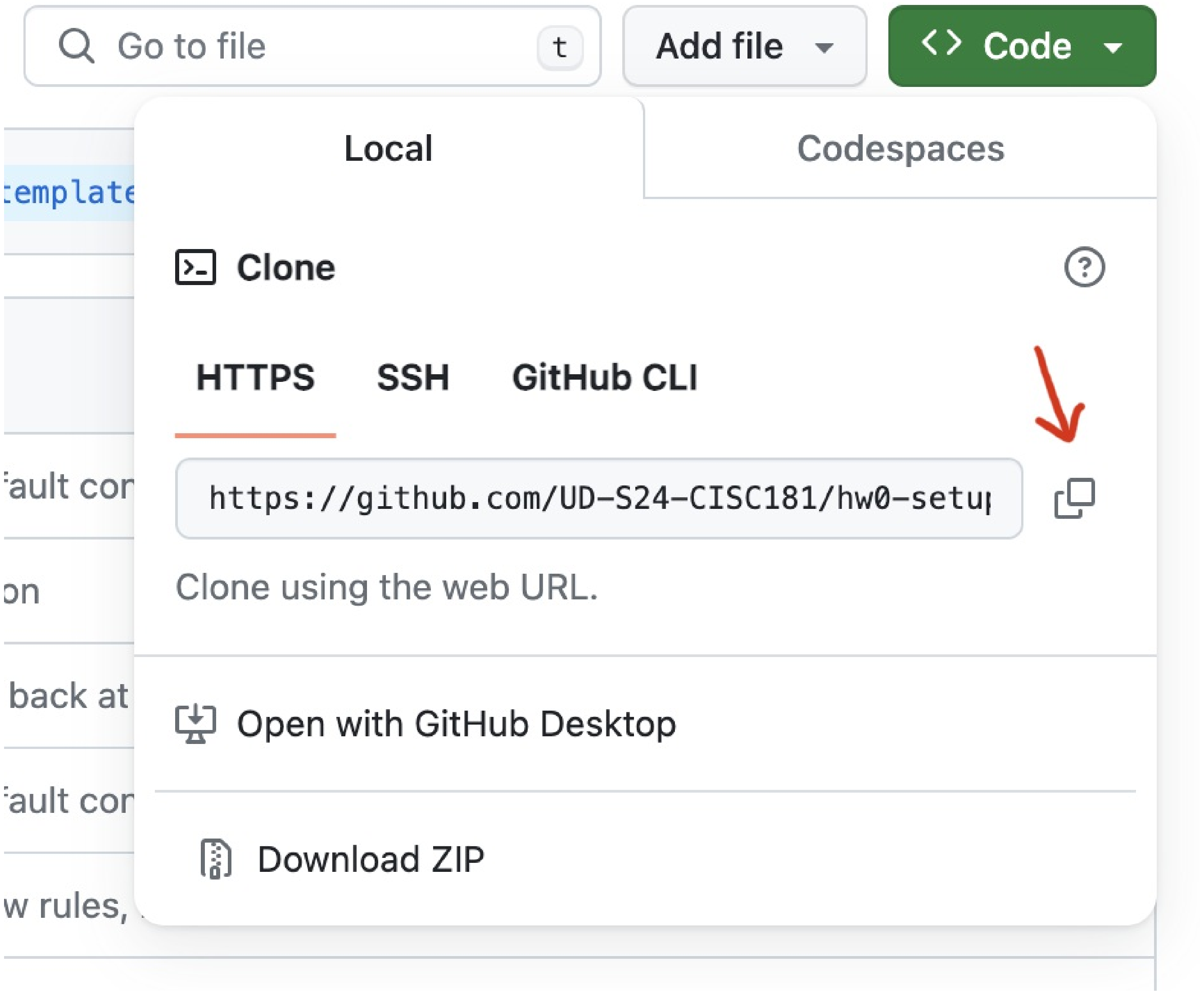
https://github.com/UD-S24-CISC181/hw0-setup-check-faithlovell

We've configured the repository associated with this assignment (update).

Click the URL for the repository (e.g., “https://github.com/UD-S24-CISC181/hw0-setup-check-acbart”) to access your
repository.


hw0-setup-check-faithlovell
Private
Watch 0
forked from UD-S24-CISC181/hw0-setup-check-template
main
1 Branch
0 Tags
Go to file
t
Add file
Code
This branch is up to date with UD-S24-CISC181/hw0-setup-check-template:main.
Contribute
Sync fork
acbart Fix title
956dd42 · 2 hours ago
9 Commits
.vscode
Update default configuration
2 weeks ago
public
Initial version
3 weeks ago

Click the green "< > Code" button and a menu will pop out.



Click the copy button to get the URL of the repository. You will **clone** this repository in VS Code, in order to get a local copy of the repository that you can freely edit.

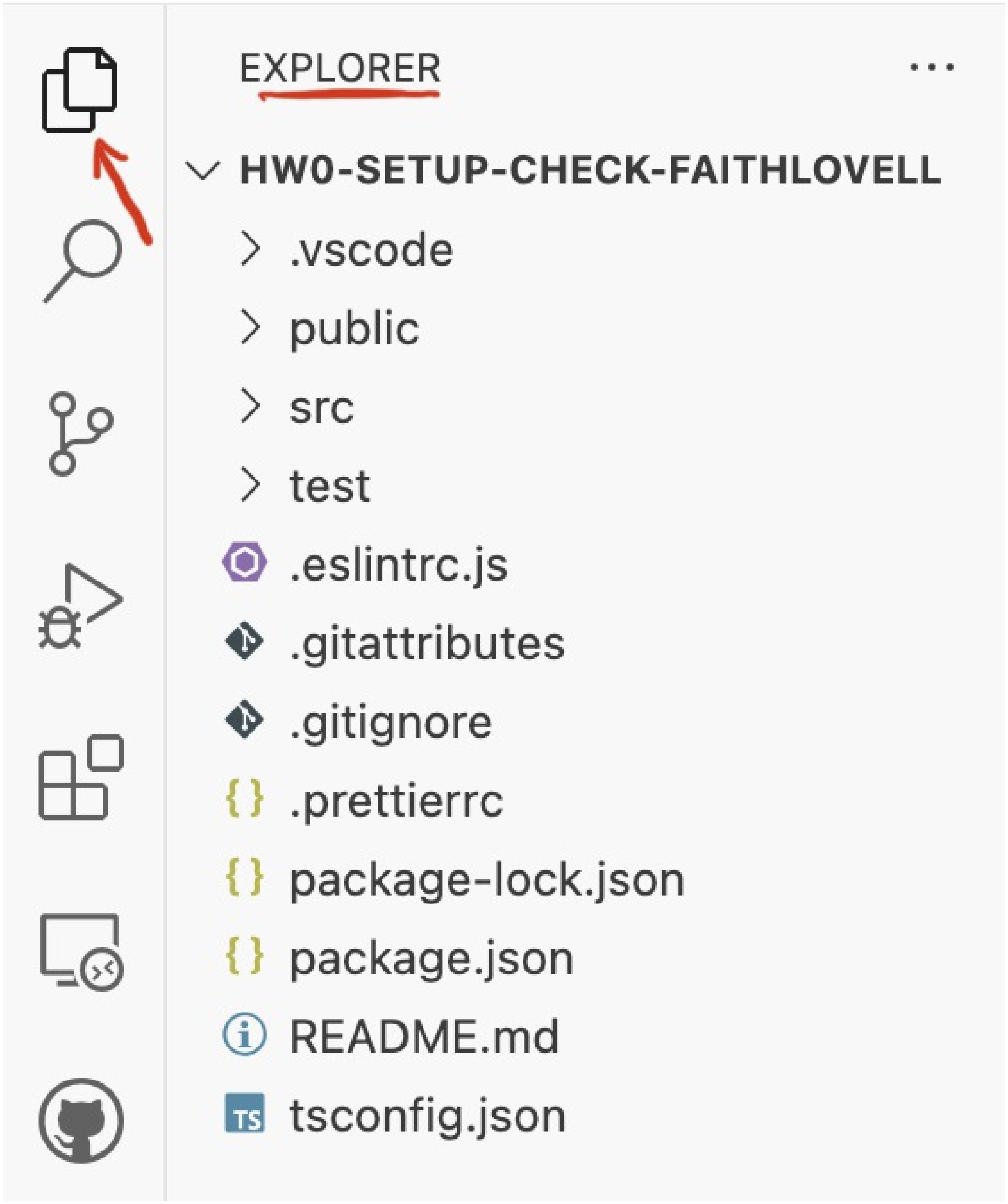
You will need to run the “Git: Clone” command in VS Code:

- Type `Ctrl+Shift+P` (Windows) or `Cmd+Shift+P` (Mac) to bring up the Command Palette
- Type `Git: Clone` and press enter
- Type `Ctrl+V` (Windows) or `Cmd+V` (Mac) to paste the previously copied link and press enter
- You may be asked to authenticate on GitHub; do so.
- A folder select window will pop up and ask "Choose a folder to clone into". We recommend that you create a `CISC-181` folder in your User directory, and store all your assignments in there. If you select that `CISC-181` folder, then a new folder will be created there for this assignment.
- When completed, it will ask if you would like to "open the cloned repository". Click “Open” to open the repository in the current window.

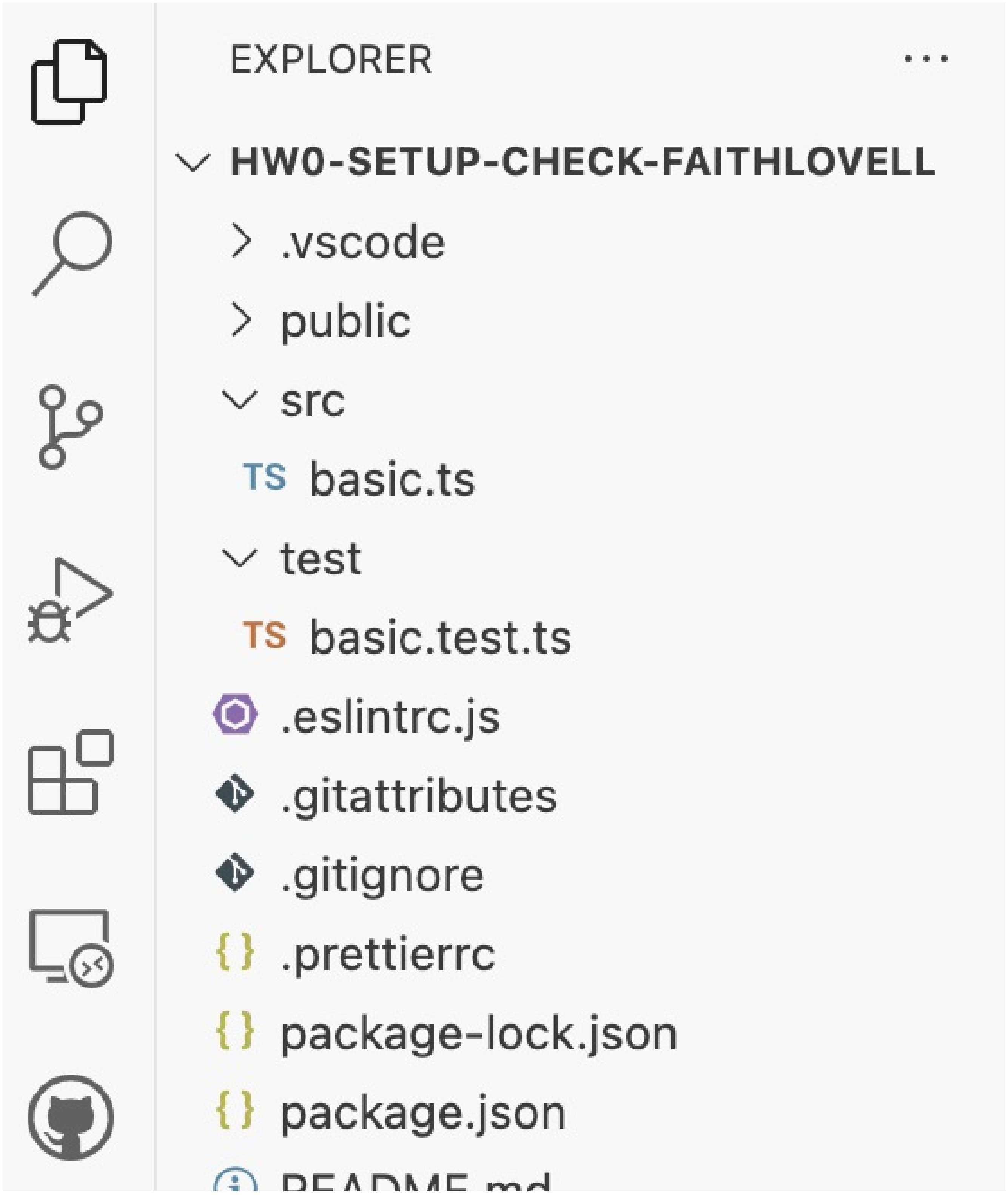
Inspecting the Project

If everything went well in the previous step, you now have the repository downloaded locally and open in VS Code.

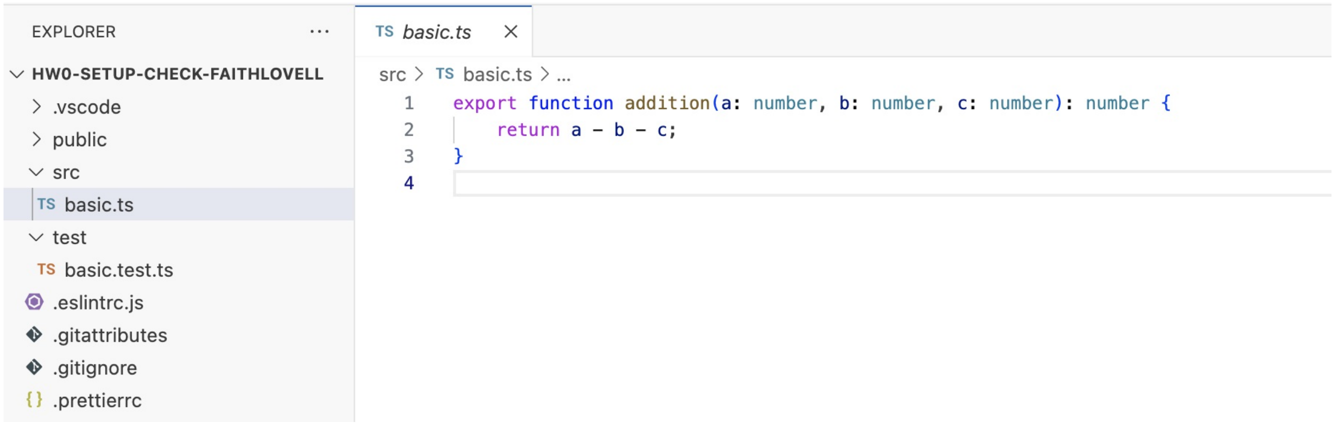
There are a lot of files already in this repository, but we only need to look closely at two of them. If the file are
not already visible, click the document icon in the topleft of the left navigation bar to see the Explorer view.


EXPLORER
...
HW0-SETUP-CHECK-FAITHLOVELL
.vscode
public
src
test
.eslintrc.js
.gitattributes
.gitignore
.prettierrc
package-lock.json
package.json
README.md
tsconfig.json

This shows all the current files in the project. We are most interested in the `src` and `test` directories, which can be expanded by clicking on them.



Click on the `basic.ts` file (NOT the `basic.test.ts` file), which is located inside of the `src` folder. This will open up the file in the editor area.

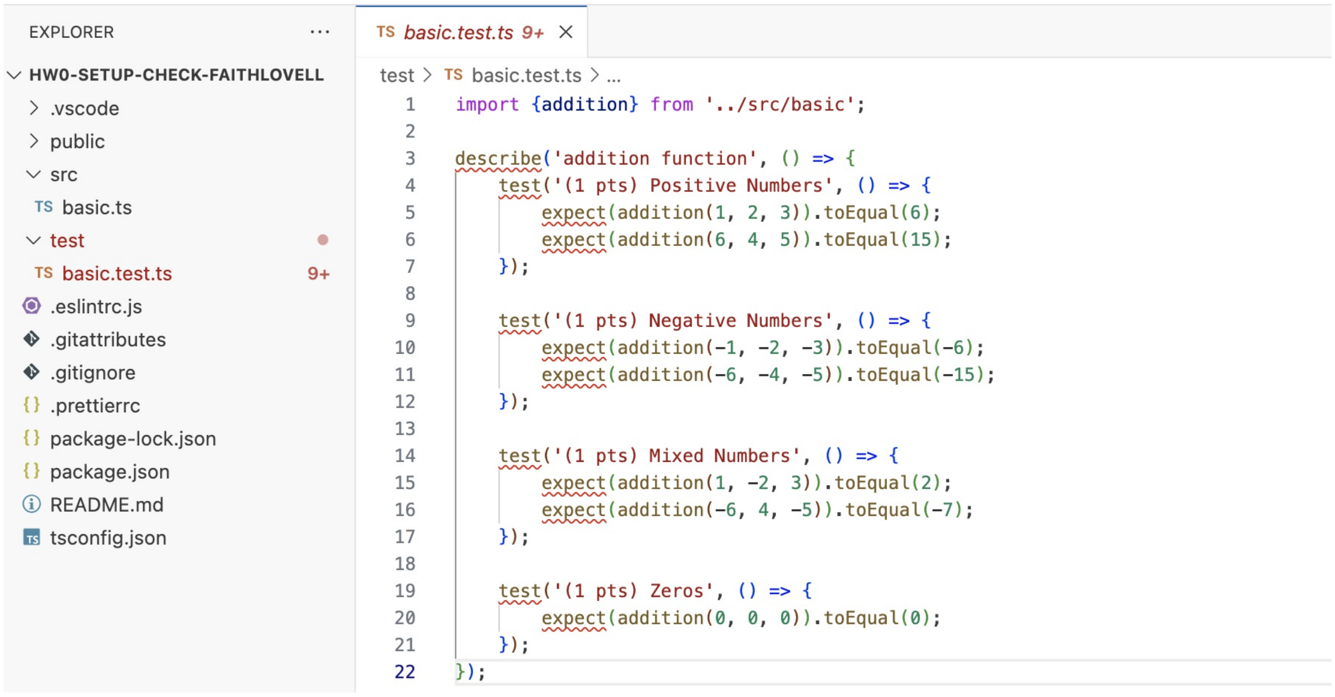


```
export function addition(a: number, b: number, c: number): number {
    return a - b - c;
}
```

It looks like someone defined and exported a function named `addition`, which takes in three `number` parameters and returns a `number`.

The code in this file is just a function, which will not do anything on its own. We could run the function definition, but we would not see anything happen since the function is not even being called. Let's try running the project's tests to see the function in action.

Click on the `basic.test.ts` file to view the file's contents:



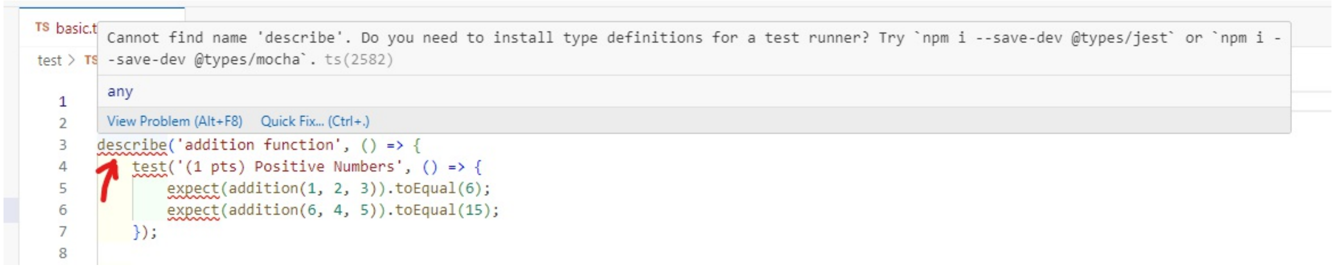
```typescript
import {addition} from '../src/basic';

describe('addition function', () => {
    test('(1 pts) Positive Numbers', () => {
        expect(addition(1, 2, 3)).toEqual(6);
        expect(addition(6, 4, 5)).toEqual(15);
    });

    test('(1 pts) Negative Numbers', () => {
        expect(addition(-1, -2, -3)).toEqual(-6);
        expect(addition(-6, -4, -5)).toEqual(-15);
    });

    test('(1 pts) Mixed Numbers', () => {
        expect(addition(1, -2, 3)).toEqual(2);
        expect(addition(-6, 4, -5)).toEqual(-7);
    });

    test('(1 pts) Zeros', () => {
        expect(addition(0, 0, 0)).toEqual(0);
    });
});
```

Note: If you single clicked on the `basic.ts`, file, then clicking `basic.test.ts` replaces the file in the current view. To keep the file open even when you click on other files, double click the filenames instead.

{: .warning-title}

Oh dear, there appear to be red squiggles in our code, the universal sign of trouble. What has gone wrong?

To find out more details, hover over the first word with the red squiggles (`describe`) and a message box will appear.



```
Cannot find name 'describe'. Do you need to install type definitions for a test runner? Try `npm i --save-dev @types/jest` or `npm i --save-dev @types/mocha`. ts(2582)
any
```

```typescript
describe('addition function', () => {
    test('(1 pts) Positive Numbers', () => {
        expect(addition(1, 2, 3)).toEqual(6);
        expect(addition(6, 4, 5)).toEqual(15);
    });
```

The interface is reporting an error: it "`Cannot find name 'describe'.`" It goes on to suggest installing "`type definitions for a test runner`" and even offers a command and some `Quick Fix` actions.

Should you take the advice of the machine? The answer will always be "it depends". If you know what you are doing and the advice is correct, then you should certainly use a tool like this. But if you are not sure, then **you should not run commands you do not understand**.

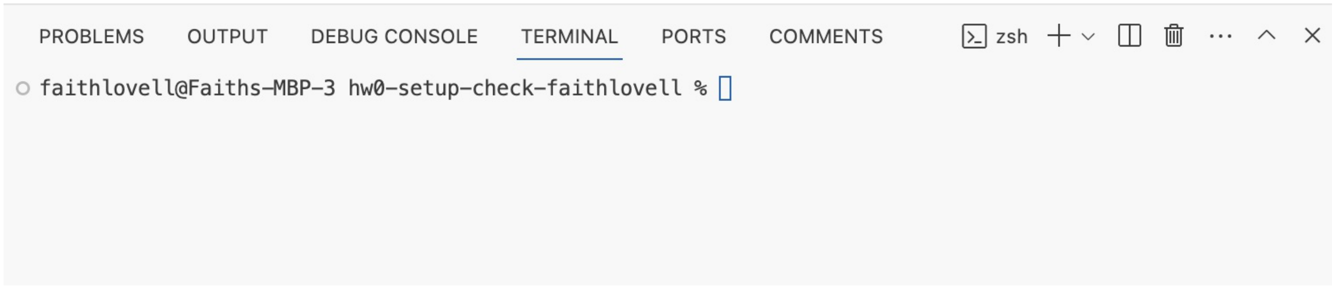
In this case, the system is letting us know that we never installed the necessary modules required to run our program.
This is one of the first steps when starting a project - to install the necessary dependencies onto your system.
We'll need to do this every time we clone a new assignment repository.

Install Modules

The next step is to open a Terminal that we can run instructions in.

At the top of your VS Code window, click Terminal and then click New Terminal. You've done this before, but this time, you need to be in the assignment directory. VS Code will do this for you!

A new terminal will appear at the bottom of your screen, and it will look something like this (notice you are in `hw0-setup-check`):

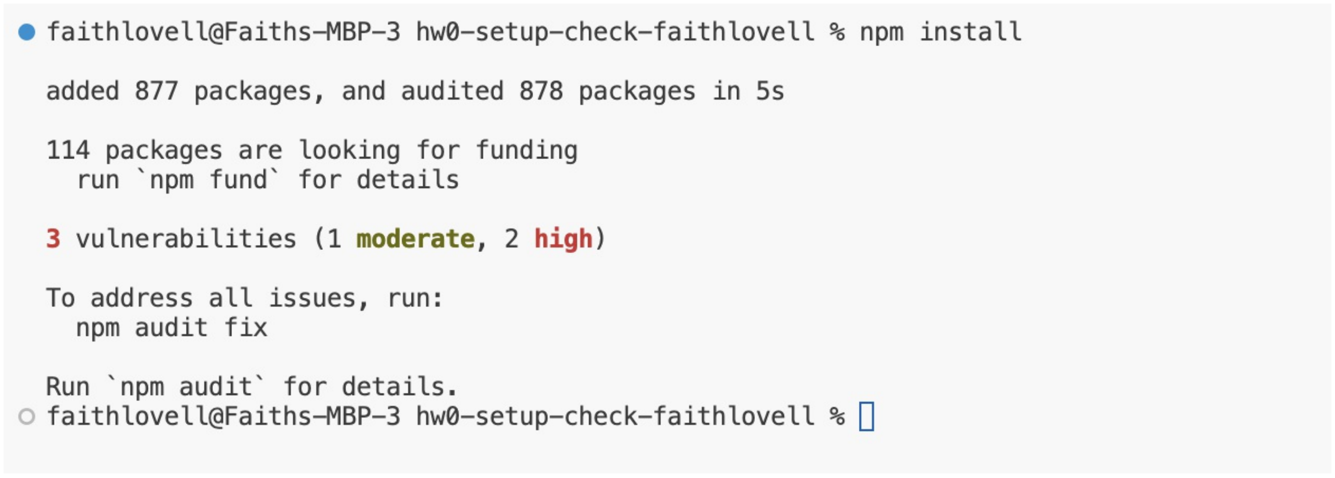


```
faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell %
```

Run the command (without the dollar sign):

```
$ npm install
```

A whole bunch of messages will appear, some of which may look alarming. Just because you see red text does not mean you
have errors, though.



```
faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell % npm install

added 877 packages, and audited 878 packages in 5s

114 packages are looking for funding
  run `npm fund` for details

3 vulnerabilities (1 moderate, 2 high)

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell %
```

Hopefully, you get a message like the one shown above. It says that it "`added 877 packages`".

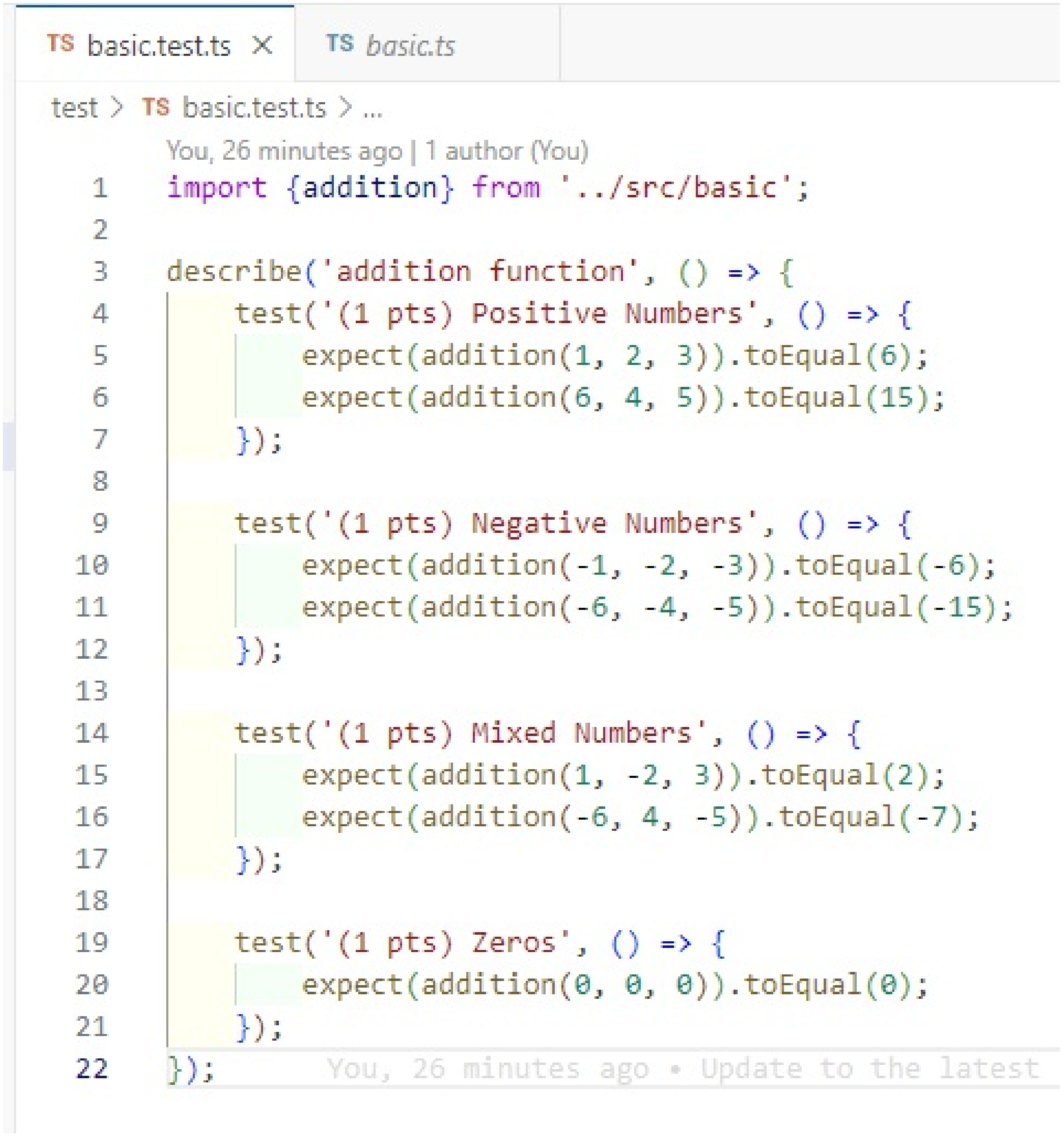
The message also says that "`3 vulnerabilities`" were detected in the packages, and offers a command to fix them. Again,
you might be wondering if you should take the advice of a machine? Remember, do not run commands you do not understand.

Modern Typescript development requires a large number of packages, even for simple projects. Often, these projects will have small vulnerabilities that are rigorously tracked by the community. If you were going to deploy a website for a
large bank or trusted government entity, then it would be very important to address these vulnerabilities.

However, you are a student learning to code. Let's not get caught up in the security ramifications of our `addition` function. At least, not yet. We will ignore these vulnerabilities and move back to the code.

Running the Tests

The `basic.test.ts` file we were looking at before no longer has red squiggles! The code is much easier to read now.


TS basic.test.ts ×
TS basic.ts
test > TS basic.test.ts > ...
You, 26 minutes ago | 1 author (You)

```typescript
import {addition} from '../src/basic';

describe('addition function', () => {
    test('(1 pts) Positive Numbers', () => {
        expect(addition(1, 2, 3)).toEqual(6);
        expect(addition(6, 4, 5)).toEqual(15);
    });

    test('(1 pts) Negative Numbers', () => {
        expect(addition(-1, -2, -3)).toEqual(-6);
        expect(addition(-6, -4, -5)).toEqual(-15);
    });

    test('(1 pts) Mixed Numbers', () => {
        expect(addition(1, -2, 3)).toEqual(2);
        expect(addition(-6, 4, -5)).toEqual(-7);
    });

    test('(1 pts) Zeros', () => {
        expect(addition(0, 0, 0)).toEqual(0);
    });
});
```

This is a test file written with a library named "**Jest** ". You have previously seen unit tests written using the Bakery’s `assert_equal` function, but Jest is a much more sophisticated testing framework. Let’s look at each part of the file in turn.

- At the top of the file (on line 1), we `import` the `addition` function from the `basic.ts` file, which is in the `src` directory. Since that function was `export`ed, we are able to `import` the function in this file.
- The next line of code (line 3) is a call to the `describe` function, which is a Jest function for organizing a suite of unit tests. It takes the `string` name of a collection of tests and then an anonymous function that has all the tests inside. Don't worry about that "anonymous function" term just yet; for now, just think of it as a block of code that Jest will run for us.
- The inside of the `describe` function call is a sequence of four calls to the `test` function (on lines 4, 9, 14, and 19). The test function is another Jest function, once again for organizing related unit tests. We give names to the tests, and sometimes we will also let you know how much that test is worth to us when we grade the assignment. Then there is another anonymous function to have the actual assertions.
- On lines 5, 6, 10, 11, 15, 16, and 20, we see the actual assertions, which are equivalent to the `assert_equal` function you saw previously. In Jest, they are written using the `expect` function, which consumes one expression (almost always a function call for the function we are testing). The result of the `expect` function is an object that has a `toEqual` method, which allows us to check the expected result. Again, don't worry about the

terms just yet, just focus on the comparable idea for writing tests between Bakery and Jest:

```
 # Bakery version in Python
assert_equal(addition(1, 2, 3), 6)
```

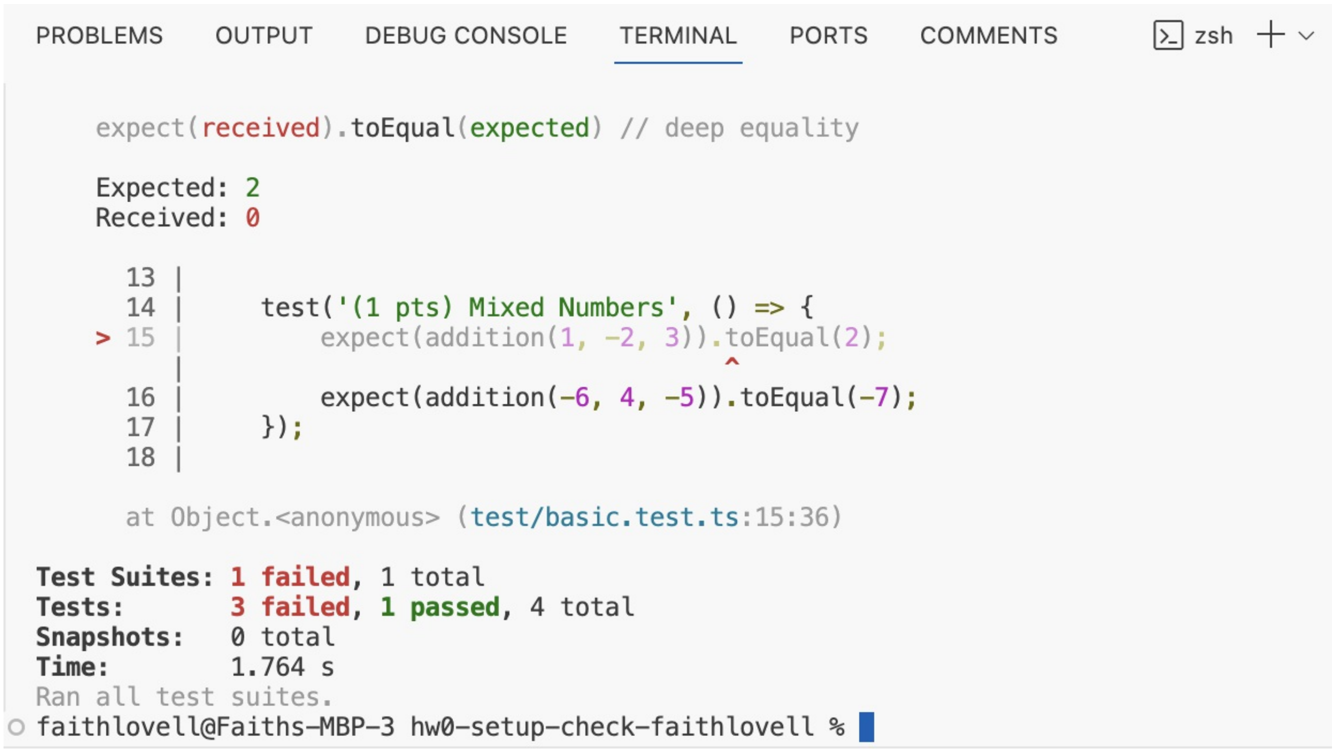
```
 // Jest Version in Typescript
expect(addition(1, 2, 3)).toEqual(6)
```

The two approaches are basically the same, but Jest has a lot of features for organizing the unit tests. Jest also has a lot of other kinds of assertions, which we might see later in this course. For now, all that matters is that we can see there are 7 tests.

Are we passing the tests? To find out, go back to the terminal and enter the following command (without the dollar sign):

```
 $ npm run test
```

The output might take a little while the first time, and may be so long that it scrolls offscreen. The final output might look something like this:

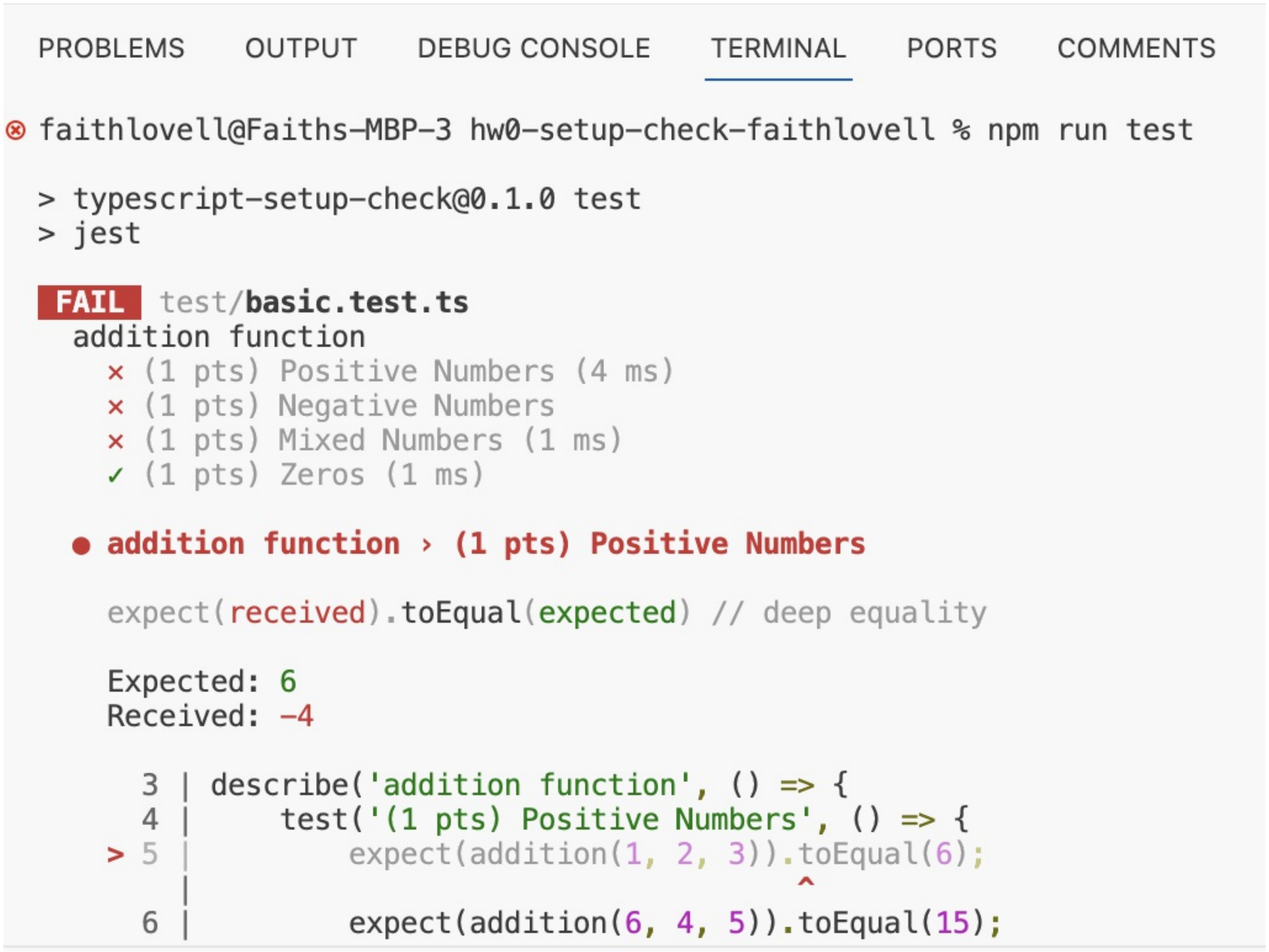


```
    expect(received).toEqual(expected) // deep equality

    Expected: 2
    Received: 0

      13 |
      14 |     test('(1 pts) Mixed Numbers', () => {
    > 15 |         expect(addition(1, -2, 3)).toEqual(2);
         |                                    ^
      16 |         expect(addition(-6, 4, -5)).toEqual(-7);
      17 |     });
      18 |

      at Object.<anonymous> (test/basic.test.ts:15:36)

Test Suites: 1 failed, 1 total
Tests:       3 failed, 1 passed, 4 total
Snapshots:   0 total
Time:        1.764 s
Ran all test suites.
faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell %
```

The bottom of the output has a summary of what happened.

- We had one test suite (“addition function”)
- With four total tests
    - Three of which failed ("Positive Numbers", "Negative Numbers", and “Mixed Numbers”)
    - One of which passed (“Zeros”)

Scrolling up through the output, you can see more details about exactly which tests failed, and the specific `expect` assertions that went wrong.

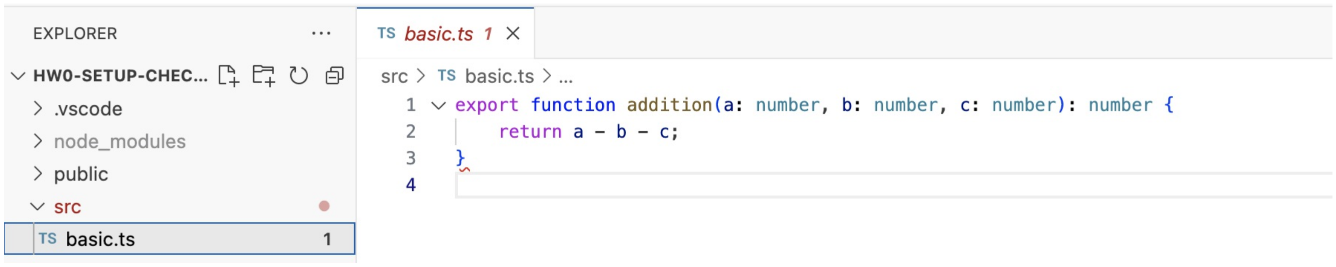


```
⊗ faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell % npm run test

  > typescript-setup-check@0.1.0 test
  > jest

  FAIL  test/basic.test.ts
    addition function
      × (1 pts) Positive Numbers (4 ms)
      × (1 pts) Negative Numbers
      × (1 pts) Mixed Numbers (1 ms)
      ✓ (1 pts) Zeros (1 ms)

    ● addition function › (1 pts) Positive Numbers

      expect(received).toEqual(expected) // deep equality

      Expected: 6
      Received: -4

        3 | describe('addition function', () => {
        4 |     test('(1 pts) Positive Numbers', () => {
      > 5 |         expect(addition(1, 2, 3)).toEqual(6);
          |                                   ^
        6 |         expect(addition(6, 4, 5)).toEqual(15);
```

According to the output, the `Positive Numbers` test called the `addition` function with the arguments `1, 2, 3` and expected the result to be `6`. However, instead the result was `-4`. There seems to be an error in our code.

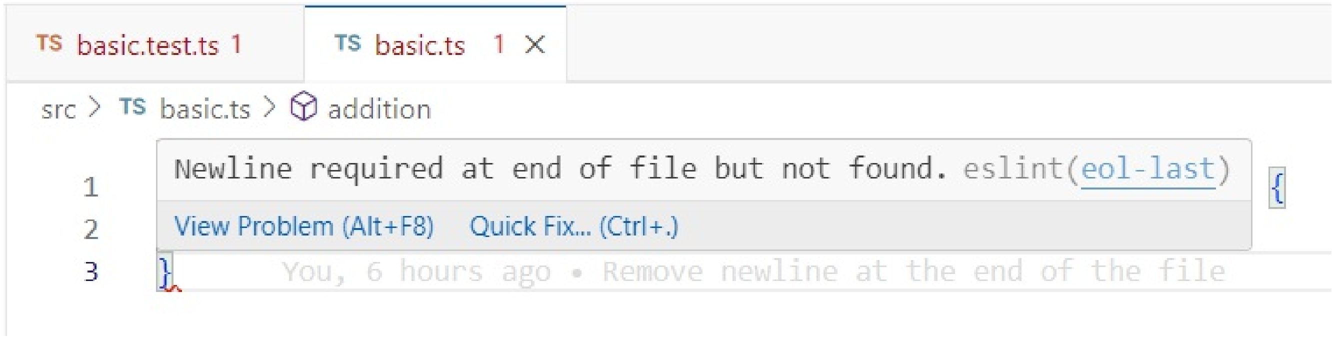
Fix the Code

Let us return to our source code file, `basic.ts`, where the addition function was defined.



```
export function addition(a: number, b: number, c: number): number {
    return a - b - c;
}

```

There is a new red squiggle waiting for us! We can hover over the squiggle to find out what it is about.



```
Newline required at end of file but not found. eslint(eol-last)
}        You, 6 hours ago • Remove newline at the end of the file
```

The feedback from the environment has nothing to do with the correctness of our code. Instead, this is the linter (**eslint**) complaining about the formatting of our file. Specifically, the system wants us to add a newline at the end of our file.

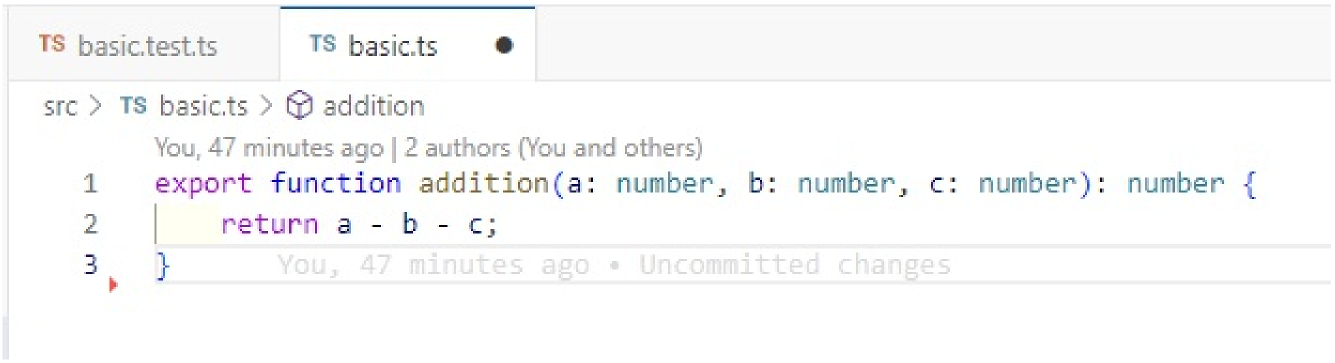
Sometimes, you may disagree with what the Linter says. There is a lot of subjective opinions about how code should be
formatted. On a good development team, all the developers will agree on a set of linting rules that they can live with. However, while you are starting out (and even sometimes when you are established), you may have to live with rules you
do not like. In this case, we need everyone to follow linting

rules to make our ability to help you more effective.
Clean, well-formatted code is much easier to read and debug!

{: .info-title}

Add a blank line at the end of the file to make the squiggle go away.

However, this does not fix the program. You should look at what the function is doing and think about it for a moment. We will not tell you the error, but you have probably already noticed it. Either way, fix the code now.


TS basic.test.ts
TS basic.ts ●
src > TS basic.ts > addition
You, 47 minutes ago | 2 authors (You and others)

```
export function addition(a: number, b: number, c: number): number {
    return a - b - c;
}       You, 47 minutes ago • Uncommitted changes
```

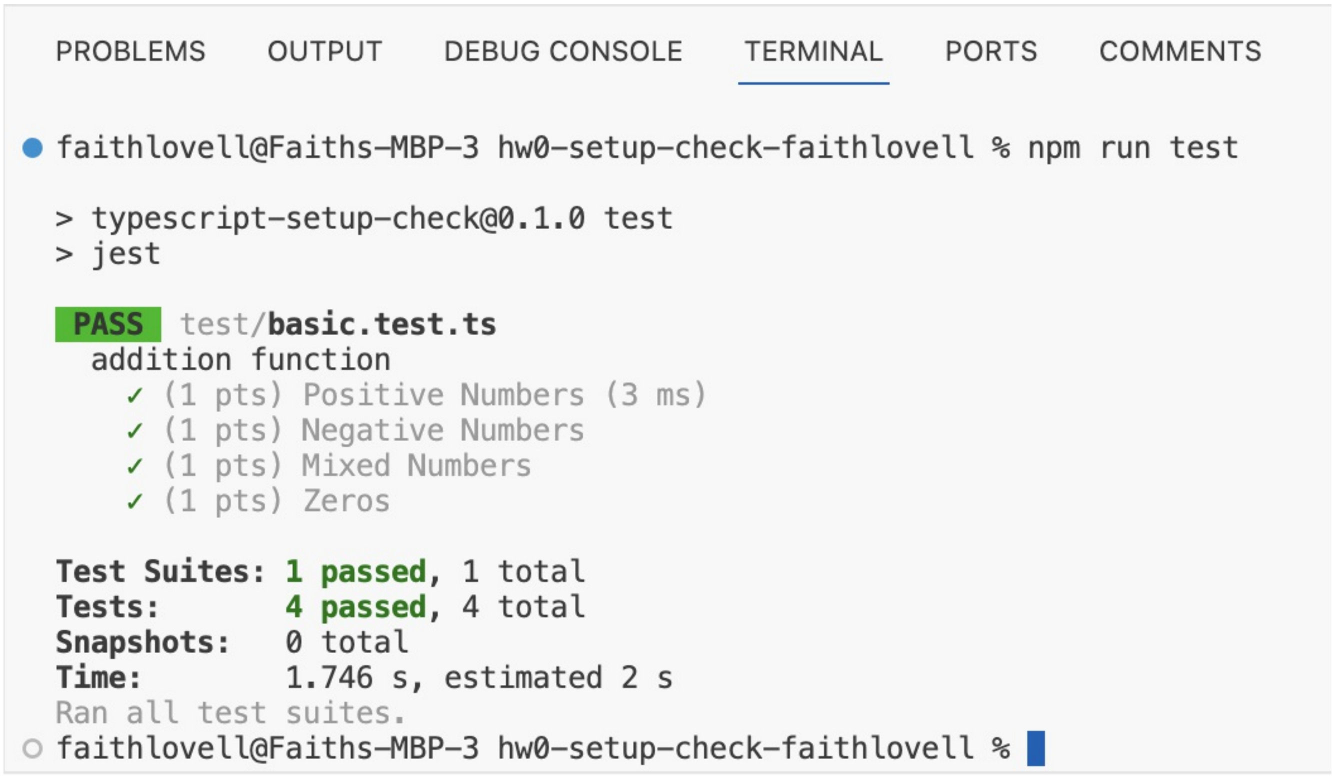
You may notice a black dot next to the name of the `basic.ts` file in the tab. This indicates that the file has not been saved. Save the file now (either using the appropriate keyboard shortcut or the File menu). If you do not fix AND save
the file, then the next step will not work.

Run the Tests Again

Now that you've fixed the code and saved the file, you can return to the terminal to run the tests again:

```
$ npm run test
```

And the resulting output this time should look a lot happier.



```
● faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell % npm run test

  > typescript-setup-check@0.1.0 test
  > jest

  PASS  test/basic.test.ts
    addition function
      ✓ (1 pts) Positive Numbers (3 ms)
      ✓ (1 pts) Negative Numbers
      ✓ (1 pts) Mixed Numbers
      ✓ (1 pts) Zeros

  Test Suites: 1 passed, 1 total
  Tests:       4 passed, 4 total
  Snapshots:   0 total
  Time:        1.746 s, estimated 2 s
  Ran all test suites.
○ faithlovell@Faiths-MBP-3 hw0-setup-check-faithlovell % 
```

With the code fixed, we are now ready to save our work back to our remote repository.

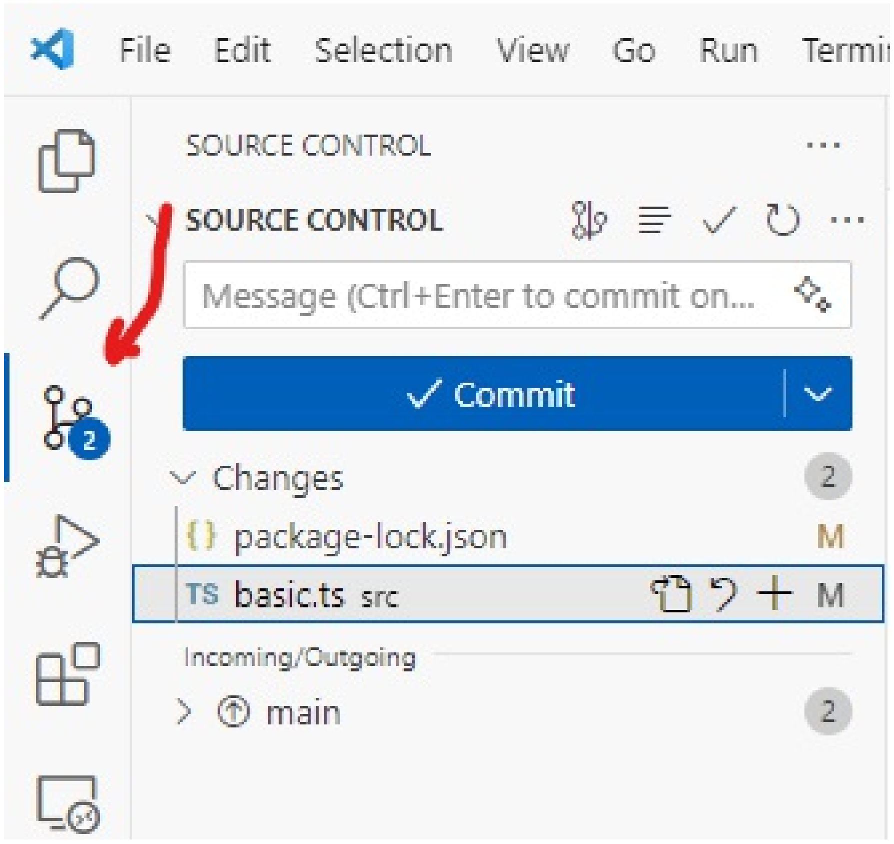
Stage/Commit/Push to GitHub

Periodically, as you complete portions of assignments, you should **stage** and **commit** your work in your local Git repository. This makes a backup of your work locally and also will give us a clear indication of your work timeline. When you are done the assignment, you can **push** your commits to the remote repository on GitHub.

We will discuss these terms a lot more in lecture, but here are some basic definitions:

- Stage: Mark locally edited files as being ready to save.
- Commit: Save a group of files' current state along with a message describing the change made to them.
- Push: Move a bunch of local commits to a remote repository.

To stage and commit files, we will use the Source Control panel, accessible from the left navigation bar.

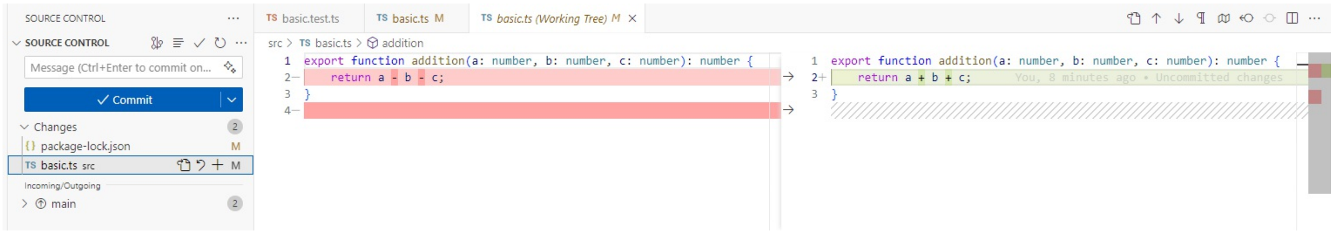


The Source Control panel gives us a graphical user interface for running Git commands. We could also run them from the terminal, but for now it will most likely be easier to use this interface.

VS Code has identified two files that we have edited: `basic.ts` and `package-lock.json`.

You hopefully remember editing the `basic.ts` file, but what about `package-lock.json`? That's a file used by the system to track the installed packages. We updated it when we ran npm install. You don't need to worry about this file.

Instead, focus on the `basic.ts` file. Click on the filename in the Source Control panel and VS Code will show you a diff (“difference”).



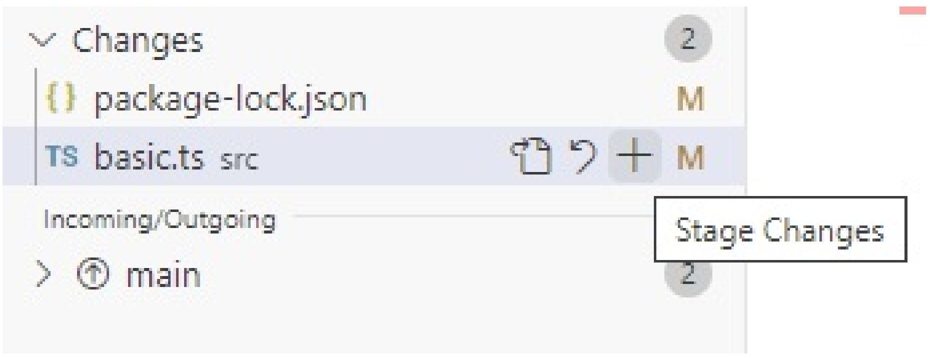
```
export function addition(a: number, b: number, c: number): number {
    return a - b - c;
}

```

```
export function addition(a: number, b: number, c: number): number {
    return a + b + c;      You, 8 minutes ago • Uncommitted changes
}
```

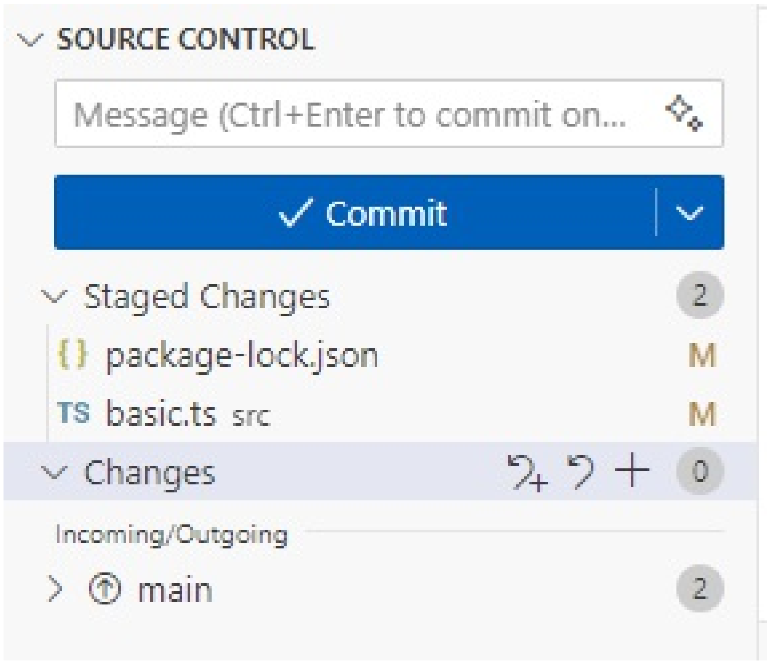
The dark red line was deleted from the left, so it shows up as a grey hatched line on the right. Similarly, the light red line was modified on the left, so it shows up as a green line on the right. This was indeed the changes we made to the file.

We’re happy with these changes, so the time has come to **stage** the files.

Click the plus button next to each file to Stage them.



When staged, the files are moved to the “Staged Changes” section, and are ready to be committed.



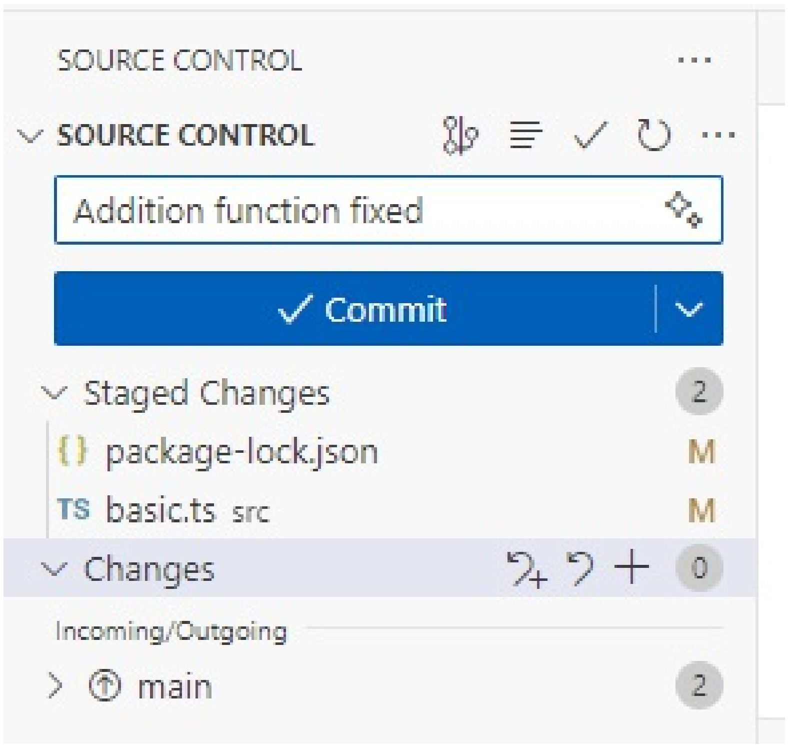
However, first we must write a commit message to explain what we have done. This message should be short, ideally fitting nicely into that box. If someone scrolls through the history of our commit messages, they should have a clear idea of what we did while writing the project.

An example commit message here might be `Addition function fixed`.

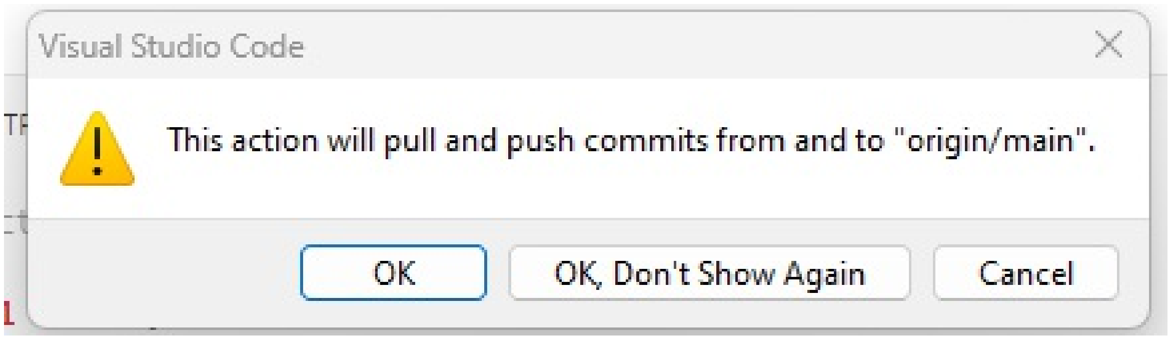
It might have been a better idea to make two separate commits, one for updating the package-lock.json file (“Modules installed”) and then one for just the basic.ts file (“Addition function fixed”). Commits don’t have to be made with all edited files; just the ones you have staged. Deciding on the granularity of your commits is a personal decision, but we encourage you to be fine-grained!

{: .info-title}

Once you have typed your message, click the Commit button to **commit** your staged changes.



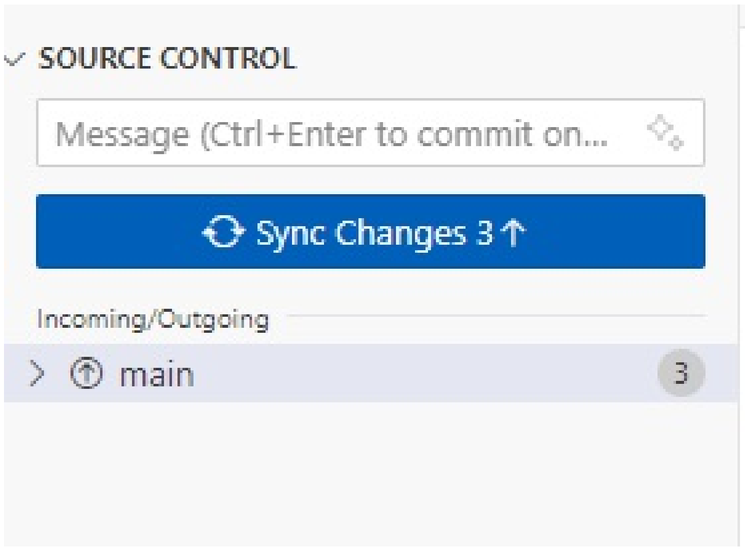
After committing, the button will change to "Sync Changes", allowing you to **push** your commits to GitHub.


Visual Studio Code

This action will pull and push commits from and to "origin/main".

Click the Sync Changes button, and you will be told "This action will pull and push commits from and to "origin/main"".

Click "OK", because that is exactly what we want to do.



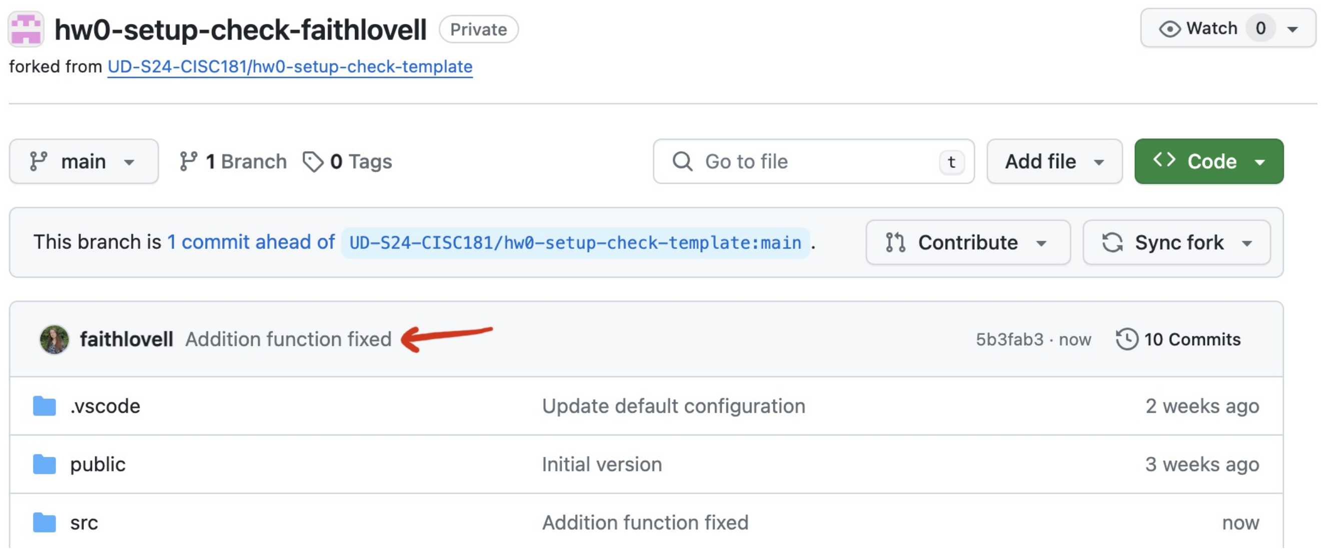
The Source Control panel will now be partially greyed out since you have nothing left to commit.

If you encounter an error like "need to configure git" before you can push, then you can run the following commands in the Terminal, substituting your email address and name.

```
 git config --global user.email "YOUREMAIL@udel.edu"
git config --global user.name "YOUR NAME"
```

Make sure you replace `YOUREMAIL` with your UD Email, and `YOUR NAME` with your name (e.g., "Austin Bart").
{: .info-warning}

If everything went well, you should be able to see your new commit on the GitHub repository website.


hw0-setup-check-faithlovell Private

forked from UD-S24-CISC181/hw0-setup-check-template

This branch is 1 commit ahead of UD-S24-CISC181/hw0-setup-check-template:main.

faithlovell Addition function fixed 5b3fab3 · now 10 Commits

.vscode	Update default configuration	2 weeks ago
public	Initial version	3 weeks ago
src	Addition function fixed	now

We’re almost done. The time has come to submit!

Submitting on GradeScope

At the bottom of the assignment page on Canvas, you will see a box with GradeScope embedded inside (just like BlockPy!). GradeScope is a
platform for running student code through instructor unit tests, which will give you automatic feedback and score you.

For this assignment, GradeScope will run the same tests that we gave you. But in future assignments, we may have hidden tests. This helps make sure that you are fulfilling all the

parts of the assignment, and not just coding directly
against the tests we gave you. Make sure you follow all the instructions!

In the box below, click Submit and then choose “GitHub” as the submission method.

The first time you submit your repository, you will need to authorize Gradescope to access your git repository.

When you click to authorize GitHub with Gradescope, the embedded page may fail to load. If this happens, just open
[https://gradescope.com (https://gradescope.com)](https://gradescope.com), go into the course and assignment (`"Homework 0- Setup Check"`), and authorize there. The
permissions should work fine in a separate browser tab.

Type the name of your repository and choose it from the dropdown. It should start with `hw1`.

From the branch dropdown, choose the `main` branch.

You can submit multiple times before the deadline. Your last submission will determine your grade. For many assignments,
we will give you additional feedback beyond what the autograder will give you, so do not assume that your grade will
remain as it is. However, if the autograder reports any issues, you should definitely handle them now!

Summary

Let us review all the steps we took in this assignment:

- Created a fork of the assignment on GitHub Classroom
- Cloned the repository onto your computer
- Installed the project’s modules using npm install
- Ran the project’s tests using npm run test
- Edited the Typescript source code files for the project in the src/ directory
- Reran the tests to make sure everything worked
- Staged, Commited, and Pushed the changes to your repository
- Submitted the repository below to GradeScope
- Confirmed that we passed all the autograder tests

This will be the workflow for the rest of the semester, so get used to it!

1) Introduction To Typescript

1.1) Variables

A variable is a named container for some unknown value. We can use variables to create generic code that works on different values.

Motivation

Simple Math Example

Consider a simple math expression:

```
3+4
```

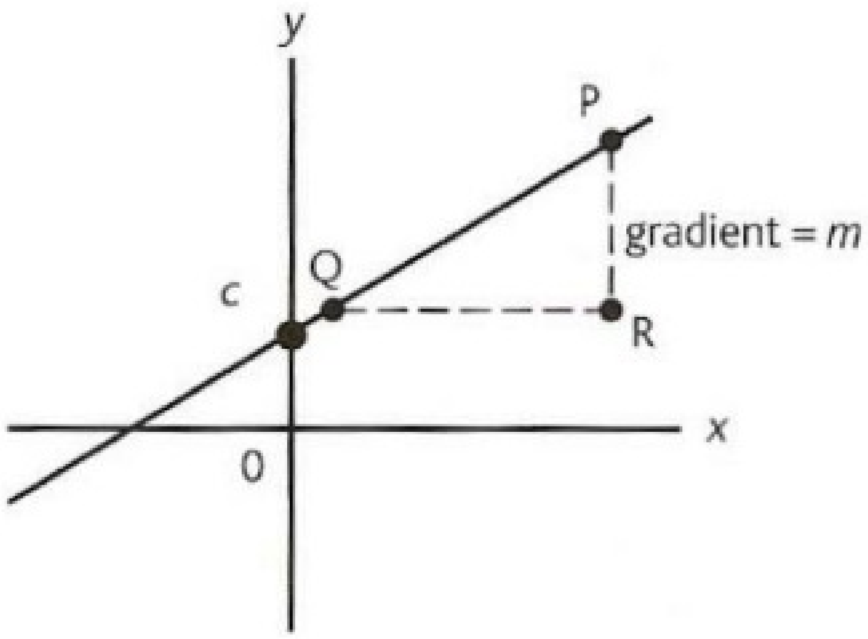
This is useful in computing this specific **value** (`7`), but is only useful in that one particular case.
On the other hand:

```
X+4
```

This would compute the same value if `X=3`, but would also compute a correct value for any other value of `X`.
This is the basic idea of why we use variables. We can write

a single expression that computes a correct answer for many possible values of `X` (the **variable**).

Another Math Example



Using variables we can represent concepts like the equation of a line. In the visualization shown:

- `m` is the slope of the line (change in `y` over change in `x`) and
- `c` is where the line intersects the `y` axis.

The equation `y = mx + c` represents every possible straight line.

`y = 2x + 4` represents a specific line. By assigning a value to the variable `x` we can compute the appropriate `y` for this line.

Just like we can use variables in math to create an expression that represents a line, in Computer Science we can use the same idea to create code that computes the correct answer for a variety of input values.

Variables Have Types

But what happens if we do this?

```
 x = "hello"
y = 2 * x + 4
```

This doesn’t make any sense.

To make sure that our code makes sense, we attach a **type** to our variables so that we will get an error if we try to assign a value to the variable that is not appropriate.

We do this by declaring the variable and specifying what type of data it can contain. Once declared, we will not be able to assign an inappropriate value type to that variable.

Declare Variables

So how do we **declare** a variable?
It depends on the language we are using, but in general, we specify:

- its name,
- its type, and
- potentially its initial value.

{: .warning-title}

Note, if we do not specify its initial value, then we cannot read its value until we do.

In this short Typescript code snippet, we declare the variable `myValue` to hold a number and assign it an initial value of `4`.
We declare the variable `answer` as a `number`, but do not give it a value.
We then compute `myValue+3` and store it in answer.

```
 let myValue: number = 4;
let answer: number;
answer = myValue + 3;
console.log(answer);
```

Declaration Syntax

TODO: Convert this to an image with annotations, perhaps

Some key notes on the syntax of declaring a variable:

- Use `let` to **declare** a variable.
- Use `:` symbol after the name, followed by the type.
- Use `=` followed by an expression to assign an initial value.
- The statement should end with a `;`

Types in TypeScript

Typescript has only three basic types.

- `number`: Holds any numeric data (e.g. `42` or `3.14159`)
- `string`: Holds a string of characters (e.g. `"Hello World"`)
- `boolean`: Holds the value `true` or `false`

There are other more complex types we will examine later (like arrays) and we can even create our own types to use in our programs.

Combining Variables

```
 // Code to compute the area of a circle with radius 2.
let pi: number = 3.1415927;
let r: number = 2;
let answer: number = pi * r * r;
console.log(answer);
```

If we change the value of `r`, then we compute the area of a different circle.

Later we will look at turning this code into a **function** that can be called with different values of `r` and reused.

If we assign a non-numeric value to `r` (which makes no sense) we would get a compiler error telling us where the problem is so we can fix it.

```
 let pi: number = 3.1415927;
let r: number = "Hello";
let answer: number = pi * r * r;
console.log(answer);
```

Boolean Expressions

Since a variable can take on many values, we might want to compare the value to something to see if it is the same, or greater than or less than.

In typescript, we do this with:

- `===`: test if equal
- `!==`: test if not equal
- `<=`: test if less than or equal to,
- `>=`: test if greater than or equal to
- `<`: test if less than
- `>`: test if greater than

{: .info-title}
Note that the equality operator is `===` (three equal signs) and not `==` (two equal signs). There is a double equal operator (`==`) operator, but it is not recommended to use it since it is not type safe. Most modern TypeScript code will use the triple equal operator (`===`).

The result of the expression will have the type **boolean**. That is, it will be either `true` or `false`.

```
 let myValue: number = 5;
let isEqual: boolean = (myValue === 5);
// isEqual will be true
let isGreater: boolean = (myValue > 5);
// isGreater will be false
let isLessEqual: boolean = (myValue <= 5);
// isLessEqual will be true
console.log(isEqual);
console.log(isLessEqual);
let myString: string = "Hello"
let isStrEqual: boolean = (myString === "Hello");
// isStrEqual will be true
let isStrEqual2: boolean = (myString !== "Hello");
// isStrEqual2 will be false
console.log(isStrEqual);
```

Summary

Variables are a powerful way to create generic code that produces expected results on a variety of different inputs.

The values that we assign to variables can come from many sources like data files, user input, databases, or online resources. The code will work regardless of the values so long as they are of the correct type.

Throughout this text we will use variables to create reusable code. We will later learn other data types, and even how to create our own types containing complex data.

1.2) Functions

A **function** is a collection of code which performs a specific task. It can take parameters and return a value.

Functions Are Blocks of Code

For now, we will discuss functions as named blocks of code. Later we will learn how to create **anonymous functions** which do not have a name, but for this review, functions will have names.

We **declare** (or **define**) a function in typescript by specifying its:

- **Name**: The name of the function
- **Parameters**: Local variables that are set to the value of the **arguments** passed into the call
- **Return type**: The expected type that this function will return
- **Body**: The code that makes up the function and will be executed when the function is called.

Once declared, we can **call** (**use**) that function anywhere in our code to execute it without worrying about the code inside. As long as we know how to call it and the meaning of what it returns, we can use it.

{: .warning-title}

Remember, you should only use the verb “call” when you are talking about invoking a function. When you are talking about defining a function or variable, use the verb “declare” or "define". When you are talking about using a variable, use the verb "use", "access", or "get". You should never use the verb “call” when talking about accessing a variable (unless that variable is a function).

Examples

An Example Function

```
 function areaOfCircle(radius: number): number{
     let pi: number = 3.1415927;
     return pi * radius * radius;
}
```

{:.no-run}

In this example, we have a function named `areaOfCircle`. It takes one parameter, `radius`, which is a `number`. The function returns a `number`.

Notice that the parameter's type is specified after the parameter name, separated by a colon. The return type is specified after the parameter list, also separated by a colon.

The body of the function is enclosed in curly braces `{}`. The code that makes up the function goes inside the curly braces, on separate lines separated by semicolons.

The final line of the function is a **`return`** statement. This statement returns the value of the expression to the right of the `return` keyword. The function will exit at this point, and the value will be returned to the call site.

Another Example Function

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

{:.no-run}

In this example, we have two parameters, `a` and `b`, both of which are `number`s. The function returns a `number`. The parameters are separated by commas.

Example Function Calls

```
 function areaOfCircle(radius: number): number{
    let pi: number = 3.1415927;
    return pi * radius * radius;
}

let myArea: number = areaOfCircle(2);
console.log(myArea);
```

We can call this function from anywhere in our code by using its name.

This code will call our function `areaOfCircle` and substitute `2` for the parameter `radius`, then return the `calculation` and store the result `12.5663708` in the variable `myArea`.

Printing with `console.log`

You may have noticed the user of `console.log` in our previous examples. `console.log` is a very important built-in function in TypeScript. This function takes any number of arguments and prints them to the console.

```
console.log("Hello, world!");
```

This code will print `Hello, world!` to the console.

Calling and Printing

A common misconception is that functions print their return value. This is not true. Functions return a value, but they do not print it. If you want to see the value, you must print it.

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

let sum: number = addTwoNumbers(2, 3);
console.log(sum);
```

You do not have to store the return value in a variable before printing it. You can print it directly.

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

console.log(addTwoNumbers(2, 3));
```

Multiple Arguments to `console.log`

You can pass multiple arguments to `console.log`. It will print each argument separated by a space.

```
console.log("The sum of", 2, "and", 3, "is", 5);
```

The output of this code will be `The sum of 2 and 3 is 5`.

Testing Functions

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

test("Test addTwoNumbers", () => {
    expect(addTwoNumbers(2, 3)).toBe(5);
    expect(addTwoNumbers(0, 0)).toBe(0);
    expect(addTwoNumbers(-1, 1)).toBe(0);
});
```

{:.no-run}

We can test our functions by calling them with different arguments and checking the return value. Usually, testing in TypeScript is done with a **testing framework** like **Jest**. The tests will be placed in a separate file from the code being tested, and the testing framework will run the tests and report the results. These testing frameworks have built-in functions like `expect` and `toBe` that make it easy to write tests, and organize them into test suites using the `test` and `describe` functions. Much of these details are not important for now, but you should be aware that testing is an important part of software development.

Documenting Functions

```
/**
 * Compute the area of a circle
 * @param radius The radius of the circle
 * @returns The area of the circle
 */
function areaOfCircle(radius: number): number{
    let pi: number = 3.1415927;
    return pi * radius * radius;
}
```

{:.no-run}

We can document our functions by adding a **comment** above the function declaration. This comment should describe what the function does, what parameters it takes, and what it returns. This is called a **JSDoc** comment. It is a special type of comment that is used to document functions, variables, and classes in TypeScript. It is important to document your code so that others can understand it, and so that you can remember what you were thinking when you wrote it. We'll talk more about **documentation** later.

Summary

Functions are blocks of code that perform a specific task. They can take parameters and return a value. We declare a function by specifying its name, parameters, return type, and body. We can call a function anywhere in our code to execute it.

1.3) Conditionals

A **conditional** is a way to alter program flow based on the value (truthiness) of some boolean expression.

The `if` Statement

In typescript, the most common conditional is the `if` statement.

- The `if` statement evaluates a **conditional** (or **logical**) expression and executes the code inside the `if` statement only *if* that expression is `true`.
- The `if` statement can have an `else` branch. The `else` branch is only executed if the expression evaluates to `false`.

Using `if` statements we can execute different code based on the values of variables at run time, allowing us to create programs that are reactive to different states as the program runs.

Example of an `if` Statement

```
 let year = "freshman";

if (year !== "senior") {
    console.log("You must register for classes");
}
```

Consider the case of a program that asks the user their year.

- If they are not a senior, the program registers them for next semester.
- If they are a senior then the program does nothing.

```
 let year = "senior";

if (year !== "senior") {
    console.log("You must register for classes");
}
```

Example of an `if` Statement with an `else` Branch

Now suppose instead of doing nothing special when the user enters senior, we want to send them an invitation to graduation.
We can handle this with an `else` branch on our `if` statement.

```
 let year = "senior";

if (year !== "senior") {
    console.log("You must register for classes");
} else {
    console.log("Come to graduation");
}
```

Nesting `if` inside of Functions

We can also nest `if` statements inside of functions.

```
/**
 * Register for classes if not a senior
 * @param year The year of the student
 * @returns A message to the student
 */
function registerForClasses(year: string): string {
    if (year !== "senior") {
        return "You must register for classes";
    } else {
        return "Come to graduation";
    }
}

test("Test registerForClasses", () => {
    expect(registerForClasses("freshman")).toBe("You must
register for classes");
    expect(registerForClasses("senior")).toBe("Come to
graduation");
});
```

The else if construct

Consider the code

```
 if (x>4){
    //do something
} else {
    if (x>2){
        //do something else
    } else {
        //do a third thing
    }
}
```

{: .no-run}

We can see that this will behave as expected.
If x is > 4, the first block will execute, otherwise the second block will execute. Within the second block if x >2 the //do something else will execute, otherwise the do a third thing will happen. This is exactly like the else if behavior we want, just a little ugly.

We can rewrite this as

```
 if (x>4){
    //do something
} else if (x>2){
    //do something else
} else {
    //do a third thing
}
```

{: .no-run}

It turns out that if the block inside an if or else is only one statement long, we are allowed to drop the {}. The compiler will then assume only the next statement is inside the block. Even though the if is multiple lines, it is a single if statement with body, so this still works.
We end up with something that does the same thing, but looks a lot better.
We have simply dropped the {} around the first else block, since the (if x>2){...} statement is the only thing inside of it.

Comparison Operators for Equality and Ordering

As a reminder, there are six main comparison operators in TypeScript:

- Equality:
    - `X === Y`: `true` if `X` and `Y` are equal
    - `X !== Y`: `true` if `X` and `Y` are not equal
- Ordering:
    - `X < Y`: `true` if `X` is less than `Y`
    - `X > Y`: `true` if `X` is greater than `Y`
    - `X >= Y`: `true` if `X` is greater than or equal to `Y`

    - `X <= Y`: `true` if `X` is less than or equal to `Y`

All of these operators are comparison operators, but they are also either equality operators or ordering operators.

Boolean Operators

We can use Boolean operators to combine boolean expressions:

- **and** (`&&`): true when both conditions are true
- **or** (`||`): true when at least one of the conditions is true, and also when both are true

```
 let happiness: number = 8;
let luckiness: number = 9;

let happyLucky: boolean = (happiness >= 7 && luckiness >
7);
// Sets happyLucky to true when both conditions are true

let happyOrLucky: boolean = (happiness >= 7 || luckiness >
7);
// Sets happyOrLucky to true when at least one of the
conditions is true
console.log("Happy and Lucky: "+happyLucky);
console.log("Happy or Lucky: "+happyOrLucky);
```

Just think of this in words:

- A and B implies both.
- A or B implies either.

The Not Operator (`!`)

An additional Boolean operator that we have available is the not (`!`) operator (also called the **negation operator**). Unlike the other operator, this operator simply negates whatever comes next.

- `!A && B`: `true` when `A` is `false` and `B` is `true`
- `!(A && B)`: `true` when at least one of `A` and `B` are `false`
- `!A || !B`: `true` when at least one of `A` and `B` are `false` (DeMorgan’s Law)
- `!(A && B) || C`: `true` when at least one of `A` and `B` are `false` or any time `C` is `true`

By using a combination of comparison operators, logical connectors, and nots we can build complex logic to test state to use in conditionals and loops…

A Complex Example

```
/**
 * Bring an umbrella if it is not raining
 * @param raining True if it is raining
 * @param temperature The temperature in degrees Fahrenheit
 * @returns A message to the user
 */
function bringUmbrella(raining: boolean, temperature:
number): string {
    if (!raining && temperature < 70) {
        return "Bring an umbrella";
    } else if (raining && temperature < 70) {
        return "Bring an umbrella and a jacket";
    } else if (!raining && temperature >= 70) {
        return "No need for an umbrella";
    } else {
        return "No need for an umbrella, but bring a
jacket";
    }
}

test("Test bringUmbrella", () => {
    expect(bringUmbrella(false, 60)).toBe("Bring an
umbrella");
    expect(bringUmbrella(true, 60)).toBe("Bring an umbrella
and a jacket");
    expect(bringUmbrella(false, 80)).toBe("No need for an
umbrella");
    expect(bringUmbrella(true, 80)).toBe("No need for an
umbrella, but bring a jacket");
});
```

Summary

- An `if` statement is a way to alter program flow based on the value of some boolean expression.
- An `else` branch can be added to an `if` statement to handle the case when the expression is `false`.
- We can use comparison operators to compare values and logical operators to combine multiple conditions.
- An `if` statement can be nested inside of a function to create complex logic.

1.4) Strings

A string is sequence of character values used to store text data.

Overview

The `string` type is a primitive data type in Typescript.
We can declare a variable to be of type string directly:

```
 let username: string = "gauss";
let password: string = 'captain';
```

{: .no-run}

Notice how we can use either single or double quotes to define a string.

String Methods and operations

There are several functions which we can use to operate on strings in Typescript.
We will look at some of the most common ones briefly, but there are actually many more!

charAt, *indexOf*, and *lastIndexOf*

You can use the `charAt`, `indexOf`, and `lastIndexOf` methods to get information about the characters in a string.

- `charAt(index)`: This method will return character at the specified index, or an empty string if the index is out of range.
- `indexOf(value)`: This method will return the index of the first occurrence of the specified value, or -1 if not found.
- `lastIndexOf(value)`: This method will return the index of the last occurrence of the specified value, or -1 if not

found.

As a more concrete example:

```
let myStr: string = "Hello World";

console.log(myStr.charAt(2)); // "l"
console.log(myStr.indexOf("o")); // 4
console.log(myStr.indexOf("x")); // -1
console.log(myStr.indexOf("lo")); // 3
console.log(myStr.lastIndexOf("o")); // 7
console.log(myStr.lastIndexOf("z")); // -1
```

Square Bracket Access of Strings

Besides using the `charAt` method, you can also access individual characters in a string using square brackets.

```
let myStr: string = "Hello World";

console.log(myStr[2]); // "l"
console.log(myStr[4]); // "o"
console.log(myStr[10]); // "d"
```

No Negative Indices with Brackets

Unlike Python, you cannot access characters in a string using negative indexes in TypeScript. The result will be the special value `undefined`.

```
 let myStr: string = "Hello World";

console.log(myStr[-1]); // undefined
console.log(myStr[-2]); // undefined
```

With the `charAt` method, the result would be an empty string instead.

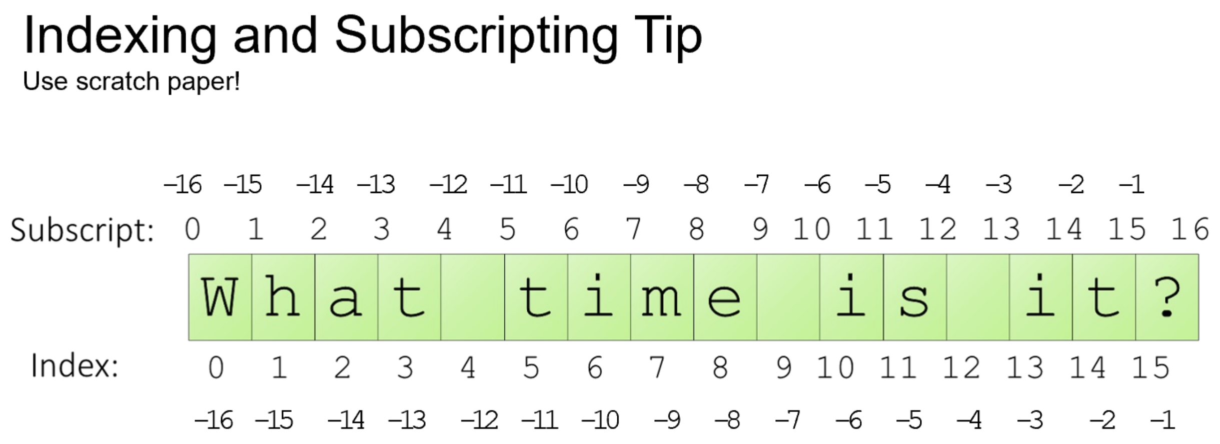
Taking Parts of Strings with `slice`

You can use the `slice` method to extract parts of a string.

- The first parameter is the starting slice position.
- The second parameter is the ending slice position (not included in the result).
- If the second parameter is omitted, the slice will go to the end of the string.
- If the first parameter is negative, it will be treated as an offset from the end of the string.
- If the second parameter is negative, it will be treated as an offset from the end of the string.

```
let myStr: string = "Hello World";

console.log(myStr.slice(2));     // "llo World"
console.log(myStr.slice(2, 5));  // "llo"
console.log(myStr.slice(-1));    // "d"
console.log(myStr.slice(-3));    // "rld"
console.log(myStr.slice(0, -1)); // "Hello Worl"
console.log(myStr.slice(5, -3)); // " World"
console.log(myStr.slice(5, 3));  // ""
console.log(myStr.slice(4, 5));  // "o"
```

Indexes and Slices in Strings


Indexing and Subscripting Tip
Use scratch paper!
–16 –15 –14 –13 –12 –11 –10 –9 –8 –7 –6 –5 –4 –3 –2 –1
Subscript: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
W h a t t i m e i s i t ?
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
–16 –15 –14 –13 –12 –11 –10 –9 –8 –7 –6 –5 –4 –3 –2 –1

It can be difficult to remember how string slicing works, compared to regular indexes. The image above should help you remember how to slice strings:

- When indexing, put numbers directly below the characters
- When slicing, put the numbers *between* the characters.

```
 let message: string = "What time is it?";

console.log(message.slice(0, 4)); // "What"
console.log(message.slice(5, 9)); // "time"
console.log(message.slice(-1)); // "?"
console.log(message.slice(-3)); // "it?"
console.log(message.slice(0, -1)); // "What time is it"
console.log(message.slice(5, -3)); // "time is"
console.log(message.slice(5, 3)); // ""
console.log(message.slice(4, 5)); // " "
```

Combining Strings with `concat`

The `concat` method will combine two separate strings and return that combined string.

```
 let myStr1: string = "Hello";
let myStr2: string = "World";

console.log(myStr1.concat(myStr2));      // "HelloWorld"
console.log(myStr1.concat(" ", myStr2)); // "Hello World"
console.log(myStr2.concat(myStr1));      // "WorldHello"
console.log(myStr2.concat(",", myStr1)); // "World,Hello"
```

Combining Strings with `+`

Note that you can also use the `+` operator to concatenate strings:

```
 let myStr1: string = "Hello";
let myStr2: string = "World";

let combined: string = myStr1 + " " + myStr2;
console.log(combined); // "Hello World"
```

The advantages of `concat` are that:

- You can combine more than two strings at once with a single operation
- You can make sure that you are only combining strings (no numbers or other types), since `concat` only works with strings. With the `+` operator, you can accidentally add numbers to strings, which can lead to unexpected results (since JavaScript will convert the number to a string and concatenate it).

The `substring` method

- Assume the string `let myStr="Hello World";`
- **split():** Splits the specified String object into an array of strings.
  - myStr.split(" "); //returns the array [“Hello","World”]
- **substring():** Returns character of string between two define indexes.

  - mySStr.substring(2); // returns “llo World”
  - myStr.substring(2,5); // returns “llo”

> *Note: the first parameter is the index of the first character to return, and the second is the index of the first character NOT returned.*

The `substring` and `slice` methods are very similar, with two differences:

- The main difference is that if the second parameter is less than the first, the `substring` method will swap them. The `slice` method will return an empty string in this case.
- The `substring` method does not support negative indexes.

The `toLowerCase` and `toUpperCase` methods

The `toLowerCase` and `toUpperCase` methods will create a new string with all characters in either lowercase or uppercase, respectively.

```
let myStr: string = "Hello World";

console.log(myStr.toLowerCase()); // "hello world"
console.log(myStr.toUpperCase()); // "HELLO WORLD"
```

Notice how the methods take no arguments; the parentheses are still required to call the method, even with nothing in between them. These are **nullary** methods because they take no arguments.

Number to String Conversion with `parseInt` and `+`

What if the string contains a number and we want to convert it to a number type? We can use two approaches:

- `parseInt`: This function will convert a string to a number, but only if the string contains a valid number. If the string does not contain a valid number, `parseInt` will return `NaN`.
- `+`: The unary addition operator can be placed before a value to convert the value to a number. This is different than the binary addition operator, which will add two numbers or strings together. The unary addition operator is less explicit than `parseInt`, but it is a common shorthand.

```
 let myNumStr: string = "42";

let myNum: number = parseInt(myNumStr); // this function
does the trick
let myNum2: number = +myNumStr; // this also works, but is
less explicit
console.log(myNum);
console.log(myNum2);
```

If `myNumStr` did not contain a valid number, the `parseInt` function would return the special value `NaN` to specify "Not a number".

```
 let myNumStr: string = "Hello";

let myNum: number = parseInt(myNumStr); // NaN
let myNum2: number = +myNumStr; // NaN
console.log(myNum);
console.log(myNum2);
```

Number to String Conversion with `toString`

If we want to go the other way, and convert a number to a string, we can use the `toString` method to explicitly convert a non-string value to a string.

```
 let myNum: number = 42;

let myNumStr: string = myNum.toString();
console.log(myNumStr);
```

The `toString` method is available on all non-string types in TypeScript, by default. That means we can use it on numbers, booleans, and other more complex types (although that is not always useful, as we will see).

Implicit String Conversion with `+`

If you use the binary `+` operator to combine a string and a number, the number will be converted to a string automatically.

```
 let myNum: number = 42;
let myStr: string = "The answer is " + myNum;
console.log(myStr);
```

This can be a useful shorthand, but it can also lead to unexpected results if you are not careful. For example, if you add a number to a string, the number will be converted to a string and concatenated to the other string.

```
 let myNum: number = 42;
let myStr: string = "The answer is " + myNum + 1;
console.log(myStr); // "The answer is 421"
```

Strings Are Immutable

The `slice` method does NOT modify the string. In fact, no methods or functions can modify a string in TypeScript. Instead, they return a new string.

```
 let myStr: string = "Hello World";

mySt.slice(1, 3); // "el"
console.log(myStr); // "Hello World"
```

Other String Methods

There are MANY other methods available to the string type, but these are some of the more useful and common. Some other useful ones we will not cover in detail here are:

- `startsWith(pattern)` / `endsWith(pattern)` : Check if a string starts or ends with a certain value
- `includes(pattern)` : Check if a string contains a certain value anywhere inside

- `padStart(length, padString)` / `padEnd(length, padString)` : Add characters to the start or end of a string.
- `replace(pattern, replacement)` : Replace a pattern with a new string
- `replaceAll(pattern, replacement)` : Replace all occurrences of a pattern with a new string
- `search(pattern)` : Find the index of a pattern in a string
- `trim` / `trimStart` / `trimEnd` : Remove whitespace from the start, end, or both ends of a string
- `split(separator)` : Split a string into an array of strings based on a separator

Summary

- Strings are a fundamental data type in TypeScript, used to store text data.
- There are many methods available to manipulate strings, and we have only covered a few of the most common ones here:
    - `charAt` , `indexOf` , and `lastIndexOf` to get information about characters in a string
    - `slice` to extract parts of a string
    - `concat` and `+` to combine strings

    - `substring` to get a substring of a string
    - `toLowerCase` and `toUpperCase` to change the case of a string
    - `parseInt` and `+` to convert a string to a number
    - `toString` to convert a number to a string
    - `slice` to extract parts of a string

- Strings are immutable in TypeScript, so any method that modifies a string will return a new string instead of modifying the original.

2) Loops and Arrays

2.1) Loops

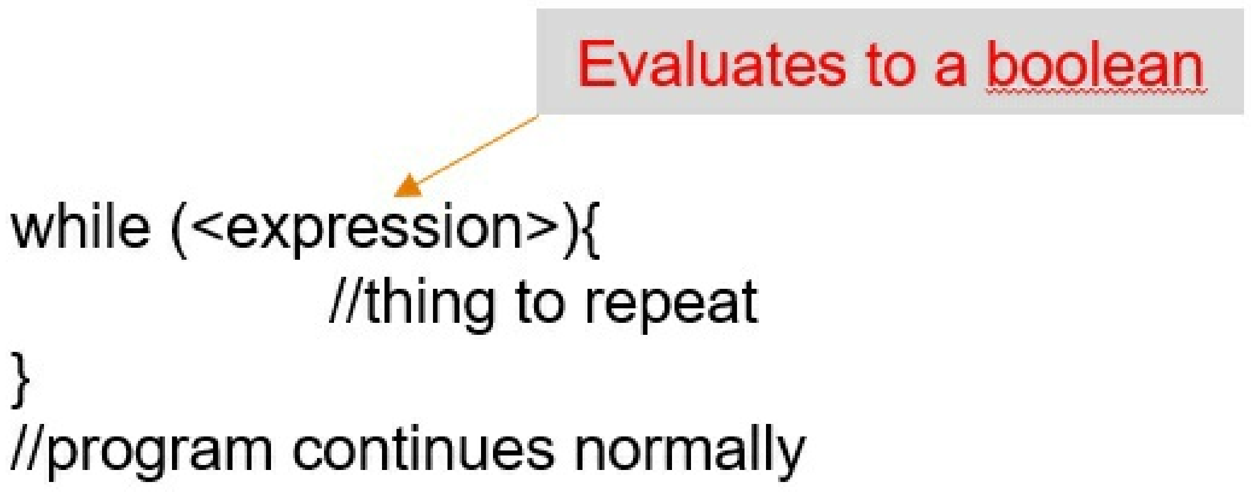
A loop is a control flow structure in programming that allows us to repeat a section of code until some boolean condition is met.

Overview

In programming we often have to do things more than once. Rather than copying and pasting our code over and over again, we can use a loop to run the same section of code repeatedly.
There are two basic types of loops that we will look at the while loop and the for loop.

While Loops

The while loop allows us to repeat the following block of code (code in braces {}) while the expression is true.



```
while (<expression>){
                //thing to repeat
}
//program continues normally
```

A simple while loop example

Consider the following function which implements a countdown. This can be done easily with a while loop.

```
  function countdown(count:number){
   while(count>0){
     console.log(count);
     count--;
   }
   console.log("beep beep beep!");
 }
 countdown(10);
```

Notice that we are calling the function countdown passing in the number we want to count down from. The number is then used in the condition of the while loop so that the function can count down from any valid non-negative integer.

> *Note that we use `console.log` to display information to the user. For now, this will be our primary way to display something from our programs.*

Exercise

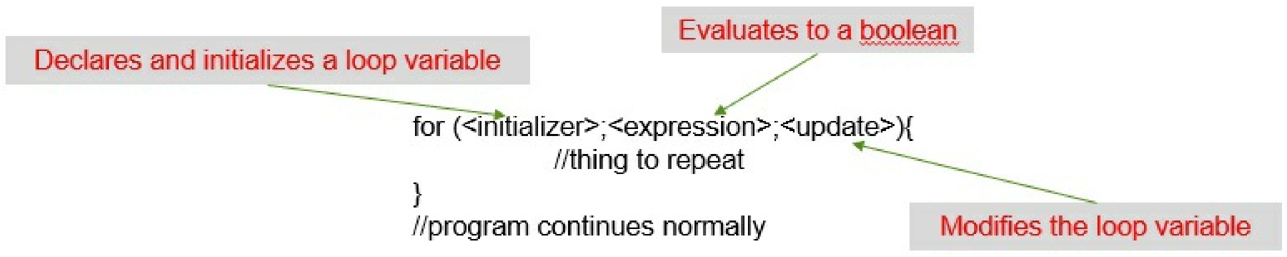
See if you can complete the function sillyMultiply and get the answer 20. You should do this using loops and you should not use multiplication in your function. You should repeatedly add the first number to itself the correct number of times.

```
 function sillyMultiply(x:number,y:number):number{
    //What goes here?
}
console.log(sillyMultiply(5,4));
```

▶ View solution

For Loops

The other primary type of loop we will be discussing is the for loop.
While you have seen for loops in other languages, they are somewhat different in typescript, and there are a couple of different versions.
Let’s start with the simplest form.


```
for (<initializer>;<expression>;<update>){
        //thing to repeat
}
//program continues normally
```
Declares and initializes a loop variable
Evaluates to a boolean
Modifies the loop variable

- The initializer is simply a variable declaration and initialization like you might use elsewhere in the program.
- The expression is the same as the expression we used for our while loop. The loop will continue to execute so long as the expression is true.
- The update statement will usually modify the loop variable so that it approaches a value that will cause the loop to exit.

A simple for loop example

Let’s take another look at the countdown example, but this time, using a for loop:

```
 function countdown(count:number){
  for (let i = count; i > 0; i--){
    console.log(i);
  }
  console.log("beep beep beep!");
}
countdown(10);
```

> *Note: i-- is just shorthand for i=i-1 (and i++ is similarly shorthand for i=i+1)*

- Our initializer sets our loop variable (i) to count
- Our expression continues the loop so long as count remains >0
- Our update statement decrements the value of i each time the loop runs

Exercise

See if you can complete the function sillyMultiply again and get the answer 20. You should do this using for loops and you should not use multiplication in your function. You should repeatedly add the first number to itself the correct number of times.

```
 function sillyMultiply(x:number,y:number):number{
    //What goes here?
}
console.log(sillyMultiply(5,4));
```

▶ View solution

Summary

We can create more complex program logic by repeating sections of our code to solve problems. This is important for many reasons including readability, reducing potential for errors, and variability of the number of times something must execute based on inputs. The two primary loops in Typescript are the while loop and the for loop. This section examined the while loop, and one of the formats of the for loop. We will examine the other for loop in the next section as it explicitly operates on collections which we will cover next.

2.2) Arrays

An array is an ordered list of values of the same type where each element in the array can be accessed using its index.

Overview

Arrays are an extremely important data structure because they allow us to store a collection of objects. We can build arrays out of any built-in or user-defined type we want, including out of other arrays.

In Typescript, the size of the array does not need to be defined. It will grow as necessary to hold the data placed into it (NOT TRUE IN C or C++).

Each element in the array has an index (starting at 0) which we can use to access the individual elements

> *i.e. if an array has 10 elements, the indexes would be 0-9.*

Defining Arrays

In typescript we define an array just like any other variable

```
 //define a single string containing the value Lisa
let name:string="Lisa";

//define an array of strings containing the values
//Lisa, Kaitlin and John
let names:string[]=["Lisa","Kaitlin","John"];
```

{: .no-run}

> *Note that we type the variable as an array of strings by using the type string[] where [] specifies that we are creating an array of that type.*

Using Arrays

Consider the following declaration of the variable ***names***. It’s type denotes an array of strings, and we initialize that array with three elements, the strings "Lisa", "Kaitlin", and "John". The array has 3 elements, so it will have indices `0, 1, and 2`. When the code accesses the element with index 1, it is requesting the second element in the array (Kaitlin) and thus the following code will print out the string ***Kaitlin***.

```
 //define an array of strings containing the values
//Lisa, Kaitlin and John
let names:string[]=["Lisa","Kaitlin","John"];
console.log(names[1]);
```

Since we can access an element of the array by its index, we can also modify that value using the index.

```
 //define an array of strings containing the values
//Lisa, Kaitlin and John
let names:string[]=["Lisa","Kaitlin","John"];
names[1]="Jan";
console.log(names[1]);
```

We would expect this code to print out ***Jan***. Initially, the second element is ***Kaitlin***, but the second line replaces the string in position 2 with ***Jan***. When we then access the second element of the array to display it, we get the updated value from the array at that position.

Array Methods and Properties

There are a number of methods that operate on arrays. We will cover some of the simple ones here. These should allow us to add elements, remove elements, and otherwise modify an array.

> *The idea of an object (like an array) having its own methods which operate on it will be central to our discussion of object oriented programming later in the text.*

The length property

We can get the current number of elements in an array by using the length property:

```
 let fruits: string[] = ["apple","banana","orange"];
let size: number=fruits.length;
console.log(size); //
```

> *Note that length is NOT a function, but rather it is a property of the array so we don’t use ().*

The push method

Using ***push*** we can add elements to the end of an array:

```
 let fruits: string[] = ["apple", "banana"];
fruits.push("orange");
console.log(fruits);  //Output: ["apple", "banana",
"orange"]
```

> *Note the* **.** *notation. We will learn more about this later.*

The pop method

Using ***pop*** we can remove elmeents from the end of an array. The ***pop*** method not only removes the last element, but it returns that value from the pop function.

```
 let fruits: string[] = ["apple", "banana","orange"];
let last=fruits.pop();
console.log(fruits);   // Output: ["apple", "banana"];
console.log(last);     // Output: orange
```

The shift/unshift methods

Analogous to push and pop, ***shift*** and ***unshift*** work on the front of the list.

```
 let fruits: string[] = ["apple", "banana"];
fruits.unshift("orange");
console.log(fruits);    // Output: ["orange","apple",
"banana"];
let first=fruits.shift();
console.log(fruits);    // Output: ["apple", "banana"];
console.log(first);     // Output: orange
```

> *Note: Adding or removing to/from the front of a list or array is generally inefficient compared to working on the end of the list. This largely depends on the implementation of arrays, but is generally true.*

The splice method

The ***splice*** method gives us a mechanism for editing the middle of an array. With the ***splice*** method, we can remove, replace, or insert elements in the middle of an array.

```
array.splice(index,[howMany],[element1],[..., elementN]);
```

{: .no-run}

- index: The array index at which to start changing the array

- howMany: The number of array elements to remove starting at index, defaults to all of them if no value is passed.
- element1…elementN: 0 or more elements to add to the array at the index.

If we only use the first parameter which is required, splice will remove that element and all elements after it from the array. It also returns what was removed.

```
 let fruits: string[] = ["apple",
"banana","orange","grape","mango"];
let removed=fruits.splice(2);
console.log(fruits);    //["apple", "banana"];
console.log(removed);   //["orange", "grape", "mango"];
```

If we set the second argument, then splice only removes that number of items:

```
 let fruits: string[] = ["apple",
"banana","orange","grape","mango"];
let removed=fruits.splice(2,2);
console.log(fruits);  //["apple", "banana", "mango"]
console.log(removed); //["orange", "grape"]
```

Any additional arguments will be added to the array at the index provided after the deletion has been completed.

```
 let fruits: string[] = ["apple",
"banana","orange","grape","mango"];
let removed=fruits.splice(2,1,"pear","kiwi");
console.log(fruits);  //["apple", "banana", "pear", "kiwi",
"grape", "mango"]
console.log(removed); //["orange"]
```

Finally, if we pass 0 as the second argument, then splice simply inserts element0,...,elementN into the array at the index position:

```
 let fruits: string[] = ["apple", "banana","orange"];
let removed=fruits.splice(2,0,"pear","kiwi");
console.log(fruits);  //["apple", "banana", "pear", "kiwi",
"orange"]
console.log(removed); //[]
```

Merging Arrays

There are a number of ways to merge arrays in typescript, but one of the simplest is to use the ***spread*** operator (three dots `...`). The spread operator extracts the elements of the array. This allows the elements to be combined into a new array.

```
 let fruits: string[] = ["apple", "banana","orange"];
let vegies: string[] = ["carrot", "potato"];
let allFood: string[] = [...fruits, ...vegies];
console.log(allFood);   //["apple", "banana", "orange",
"carrot", "potato"]
```

> *The spread operator can be used any time you need to extract the elements of an array.*

Arrays of arrays

Since arrays are just collections of objects, and arrays are themselves objects, we can build arrays out of other arrays, thus creating multi-dimensional arrays. Consider the example:

```
 let fruits: string[] = ["apple", "banana","orange"];
let vegies: string[] = ["carrot", "potato"];
let allFood: string[][] = [fruits, vegies];
console.log(allFood);   //[["apple", "banana", "orange"],
["carrot", "potato"]]
```

In this example *allFood* is an array of string arrays containing two elements.

- Each element is an array of strings
    - allFood[0] has 3 string elements
    - allFood[1] has 2 string elements

More to see

There are many other methods for manipulating arrays. We will cover many of these in later chapters. This set should be sufficient for the time being.

Specialized loops for working with arrays

One important use of loops is to iterate through the elements of an array. We can certainly do this using our existing knowledge of loops and arrays.

```
 let fruits: string[] = ["apple", "banana","orange"];
let size:number=fruits.length;
for (let i = 0; i < size; i++){
  console.log(fruits[i]);
}
```

This works and is perfectly acceptible, but there is a special version of the ***for loop*** which can be used to iterate through the elements of an array.

We can use this other version called a ***for…of loop*** to automatically iterate through the array.

```
 let fruits: string[] = ["apple","banana","orange"];
for (let fruit of fruits){
  console.log(fruit);
}
```

This is much cleaner, doesn't require getting the length of the array, and accesses every element in order just like the previous version.

> *It is common to use the **for…of loop** syntax when iterating through the elements of an array.*

Summary

Arrays provide a simple data structure to store collections of objects. These objects can be simple types (string, boolean, number), or complex types including other arrays. We can access elements in the array by their index which is 0 based (i.e. 0 is first element). There are also a number of functions to mutate the array by adding and removing elements to the back, front, or middle of an array. Array elements can be extracted from an array using the spread (…) operator. A special version of the for loop (for…of) can be used to automatically iterate through the elements of an array.

3) Data Classes

3.1) Introduction

Data Classes allow us to combine data into a grouping or class and use that grouping as a data type in our programs.

Complex Types

If we wish to combine data into a more complex type that represents the combination of various related data items, then there are two methods available to us in Typescript

- **The interface**
    - Interfaces describe the data that goes into an object and its types, but do not provide default values, or any additional logic.
- **The class**
    - Classes also describe the data that goes into an object, but provide a mechanism to set default values, construct the objects dynamically, and even define methods to operate on the internal data.

> *We will discuss interfaces and how and when to use them, later in the text. For the time being we will focus on classes, and specifically **Data Classes**.*

Classes in Typescript

To declare a class in typescript, we use the `class` keyword. The structure of a class internally is a set of data objects that make up the class.

```
 class MyType{
    //list of variables and types
    //constructor method to create an instance
    //0 or more methods which can operate on
       //the member variables in the class
}
```

{: .no-run}

> *Remember a class is a definition of a type. You must create an instance of that type in order to use it.*
> *We can define a variable of our new type and use it.*

```
let myObj:MyType=new MyType();
```

{: .no-run}

Motivation

Some things things belong together as they describe a more complex thing that we want to represent.

As an example, consider a simple drawing program we might want to build.

- Points have an x and y coordinate which are numbers
- Lines contain a start and end point
- Rectangles can be defined by 2 points (opposite corners)
- Polygons can be defined by an arbitrary list of points (The vertices)
- Each of these objects may have a color associated with it. (Color itself might contain components for Red, Green, and Blue as numbers.

Summary

Sometimes it makes sense to group data together. In these cases Typescript provides multiple mechanisms with which to do that. In the section we have introduced the idea of creating a ***class*** that represents a set of heterogeneous data. (i.e. strings, numbers, booleans, arrays, and other classes).

3.2) Basic Data Classes

Data Classes allow us to combine data into a grouping and use that grouping as a data type in our programs.

Drawing Program Classes

Color class

If we examine the objects we have proposed for our drawing program (points, lines, rectangles, polygons, color) we can see that just about everything has a color. The definition for a type that represents color would be useful as then we could group the things that make up a color. For our example we want to store a color as three numbers between 0 and 255 representing the red, green, and blue intensities.

Our class should contain 3 numbers (Red, Green, and Blue). We can define our class as described previously, since this contains only the primitive type number.

```
 class Color{
    public red:number=0;
    public green:number=0;
    public blue:number=0;
}
```

{: .no-run}

> *Note the **public** keyword before each member variable (sometimes called a property) of the class. This denotes that the property is accessible by methods and code outside the class. We could also mark it as **private** or **protected**.*

As you can see, our class definition is quite simple. We simply group the three components together and give it a name. We can then create objects of this type with the new keyword.

```
let myColor:Color=new Color();
```

{: .no-run}

And for a full example:

```
 class Color{
   public red:number=0;
   public green:number=0;
   public blue:number=0;
}
//We can use our new class to create a color object:
let myColor:Color=new Color();
//Then we can access its public members
myColor.red=255;
myColor.blue=128;
myColor.green=10;
console.log(myColor);
```

> *Note: If red, green, and blue had been labeled private, then we could not have accessed them. More on this later in the text.*

Point class

A point requires coordinates, x and y. These are both numbers. It also requires a color if we want points to be displayable (more on this later). We already have a definition for a color, so we can use that to define a point.

```
 class Color{
    public red:number=0;
    public green:number=0;
    public blue:number=0;
}
class Point{
    public x:number=0;
    public y:number=0;
    public color:Color=new Color();
}
let myPoint:Point=new Point();
myPoint.x=100;
myPoint.y=100;
myPoint.color.red=255;myPoint.color.blue=128;
myPoint.color.green=10;
console.log(myPoint);
```

> *Notice that we use the class Color inside of the class Point. This is referred to as* ***composition*** *and is a critical concept in understanding classes and Object Oriented Programming.*

We can build up complex objects by including other objects inside of them. Now every point will have a position (x,y) and a color contained inside the point itself.

Summary

Complex objects can be built from simpler ones by creating a ***class*** to represent a new type.

3.3) Data Class Constructors

Data Classes allow us to combine data into a grouping or class and use that grouping as a data type in our programs.

Class constructors

So far, to create a class we:

- Create an instance of a class with the **new** keyword and store it in a variable
- Use the variable to modify the properties of the class individually
    - For our Color example, this means setting red, green, and blue independently.

It would be much easier to have a function that takes the parameters we want to set and updates the object as it is being created.

```
 class Color{
   public red:number=0;
   public green:number=0;
   public blue:number=0;
   constructor(red:number,g:number,b:number){
      this.red=red;
      this.green=g;
      this.blue=b;
   }
}
```

{: .no-run}

By giving our class a constructor, we can create an instance of the class and initialize its values in one line:

```
 let veryRed:Color=new Color(255,0,0);
let veryBlue:Color=new Color(0,0,255);
let anotherColor:Color=new Color(27,115,98);
console.log(veryRed,veryBlue,anotherColor);
```

{: .no-run}

> *Note that now we are creating and initializing our objects in one line.*

While much better, the definition of Color still seems repetative. While 100% correct, Typescript gives us a shorthand.

```
 class Color{
   constructor(public red:number, public green:number,
public blue:number){
    //Note we don’t need anything inside.  This
automatically does everything.
   }
}
//this behaves equivalently in every way to our previous
example.
let veryRed:Color= new Color(255,0,0);
console.log(veryRed);
```

> *If we declare the parameters of the constructor with the private or public keywords, it both declares them as members, and initializes their values from the values passed into the constructor.*

> *Note that without the public or private keywords, the parameter is just local to the constructor function, but when included, the parameter becomes a member variable (property) and gets initialized to the value passed in.*

Back to the drawing program, we can now rebuild our classes using constructors and the Typescript shorthand.

```
 class Color{
    constructor(public red:number, public
green:number,public blue:number){ }
}
class Point{
    constructor(public x:number,public y:number,public
color:Color){}
}
// We can build a point in a few ways.
let myPoint:Point=new Point(100,100,new Color(0,0,255));
//create color on the fly
let red:Color=new Color(255,0,0);
let myOtherPoint:Point=new Point(200,200,red);  //use an
existing color object
console.log(myPoint);
console.log(myOtherPoint);
```

Other Drawing classes

What other classes do we need:

The Line class simply needs two points (start and end) and a color. We define the class to have those three components and initialize them with a constructor

```
 class Color{
    constructor(public red:number, public
green:number,public blue:number){ }
}
class Point{
    constructor(public x:number,public y:number,public
color:Color){}
}
class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
}
class Rectangle{
    constructor(public corner1:Point,public
corner2:Point,public color:Color){}
}
```

{: .no-run}

Polygons

Now we can represent basic shapes in a coordinate system and each shape has a color, but what about polygons. First, let’s list what we know about them:

- Generalized polygons have 3 or more points which are connected.
- Polygons have a color

Since we don’t know how many points there are to start with, we can represent the list of points using an array.

```
 class Polygon{
     constructor(public points:Point[],public color:Color){}
}
```

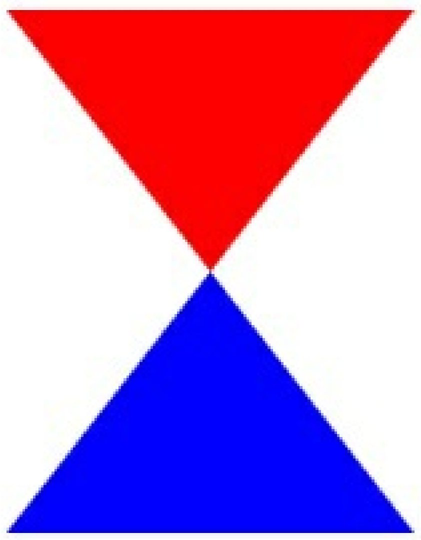
{: .no-run}

The polygon class is initialized by and contains a public member whose type is an array of Point classes. It also has an instance of a Color class.

Trying it out

```
 class Color{
    constructor(public red:number, public
green:number,public blue:number){ }
}
class Point{
    constructor(public x:number,public y:number,public
color:Color){}
}
class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
}
class Rectangle{
    constructor(public corner1:Point,public
corner2:Point,public color:Color){}
}
class Polygon{
    constructor(public points:Point[],public color:Color){}
}
let red:Color=new Color(255,0,0);
let blue:Color=new Color(0,0,255);
let points1:Point[]=[new Point(0,0,red),new
Point(100,0,red),new Point(50,100,red)];
let points2:Point[]=[new Point(50,100,blue),new
Point(100,100,blue),new Point(0,100,blue)];
let redTriangle:Polygon=new Polygon(points1,red);
let blueTriangle:Polygon=new Polygon(points2,blue);
let drawing:Polygon[]=[redTriangle,blueTriangle];
console.log(drawing);
```

With this code, drawing represents a drawing with two triangles (red and blue). If we wrote a program to render these objects, we would have all of the information that is needed.



Summary

To simplify the creation and initialization of a data class, we can provide a constructor method that takes parameters and can be used to set initial values for the member properties. If the parameters are preceeded by the words public or private, they automatically become member variables and get initialized to the values passed to the constructor. The constructor is called by using the ***new*** keyword to create an new instance of the class.

3.4) Instances and References

Data Classes allow us to combine data into a grouping or class and use that grouping as a data type in our programs.

Understanding Instances and References

When we define a class using the ***class*** keyword, we are creating a ***type***. This type does not exist in memory, but is a template for creating objects that have the methods and fields described in the class. When we use the ***new*** keyword, we create an **instance** of the class in memory and return a **reference** to the in memory object. If we call new again, we get a second instance of the class and a second **reference** to the new memory location.

```
import {Color,Point,Polygon} from 'ch2/drawing1';

let red:Color=new Color(255,0,0);
let points1:Point[]=[
    new Point(0,0,red),
    new Point(100,0,red),
    new Point(50,100,red)
];
let redTriangle:Polygon=
    new Polygon(points1,red);
console.log(redTriangle);
```

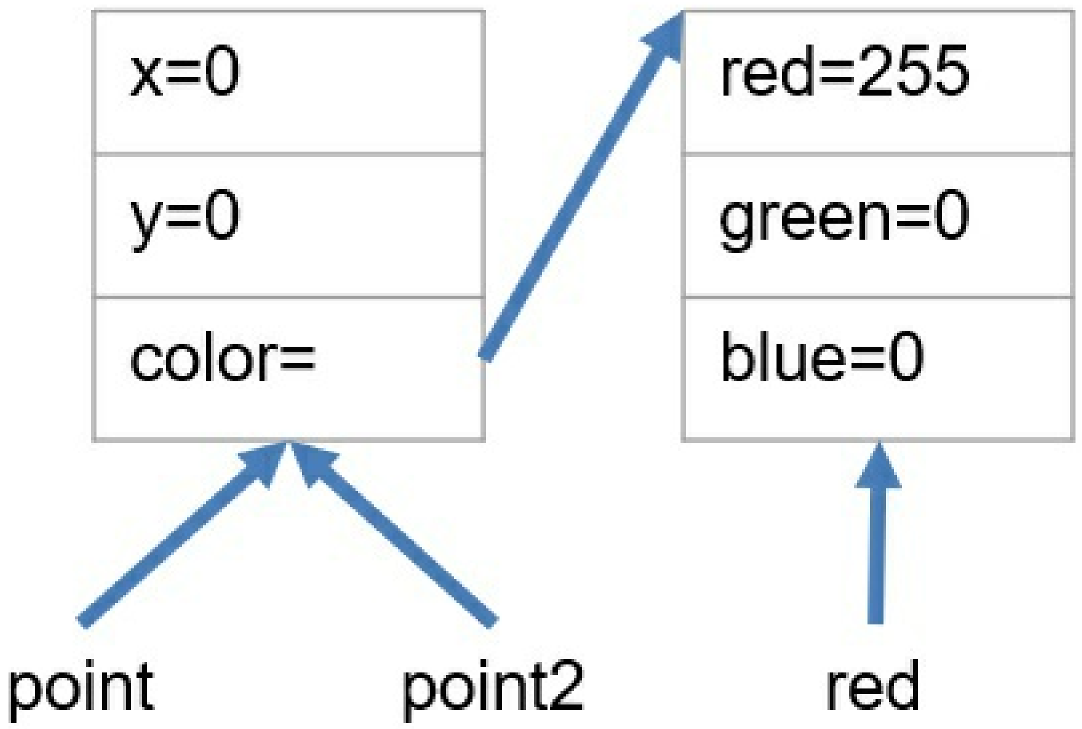
Examining this code in more detail, we see that each time new is called, we are creating an **instance** of the class. That means that each time we call ***new***, we are allocating a new chunk of memory to hold the values of that instance. What is returned, is not the value of the class, but a **reference** to the created object.

Consider the following code:

```
let red:Color=new Color(255,0,0);
let point:Point=new Point(0,0,red);
let point2:Point=point;
```

{: .no-run}

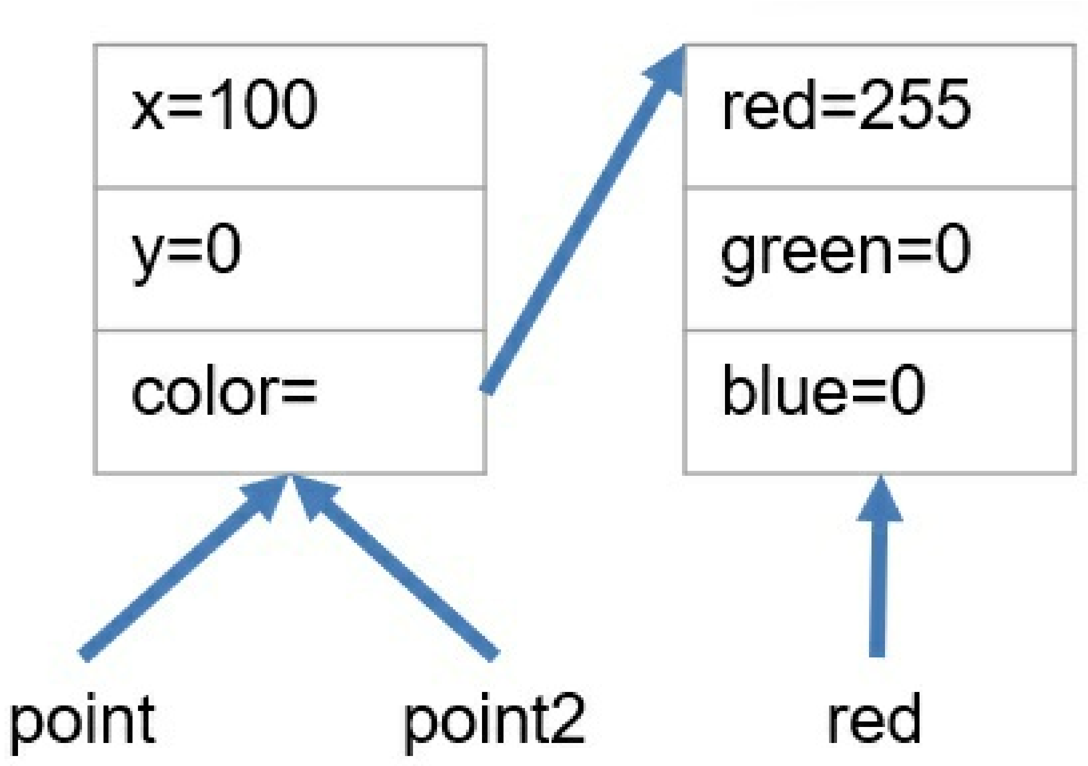
Graphically, this looks like:



What would happen if we update point.x. In this case we would also update the instance pointed to by point2, because they are the same instance. When we set `point2=point;` we are setting the variable point2 to contain the reference stored in point, and thus they reference the same chunk of memory allocated by the one and only call to ```new Point(…).

Let’s see that in action.

```
 import {Color,Point} from 'ch2/drawing1';

let red:Color=new Color(255,0,0);
let point:Point=new Point(0,0,red);
let point2:Point=point;
point.x=100;
console.log(point2);
```

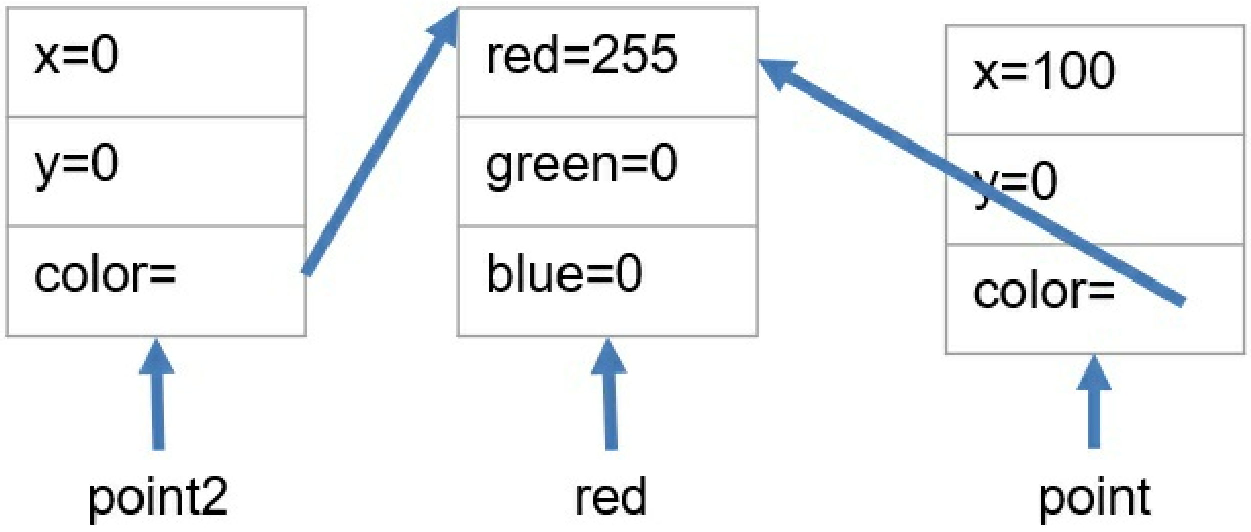


As you can see, updating point updates the memory location referenced by point which is the same memory location referenced by point2. In other words, we only have one point, but we have two **references** or **aliases** to that point. Changing either one, changes the one and only object that the variables point and point2 refer to.

> *Note: Sometimes this is what we want, but sometimes IT IS NOT!!!*

Later, we will look at other methods to create new objects based on existing objects, but for now, we would have to call ***new*** again and set point2 to that new object, then update its properties with the properties of point.

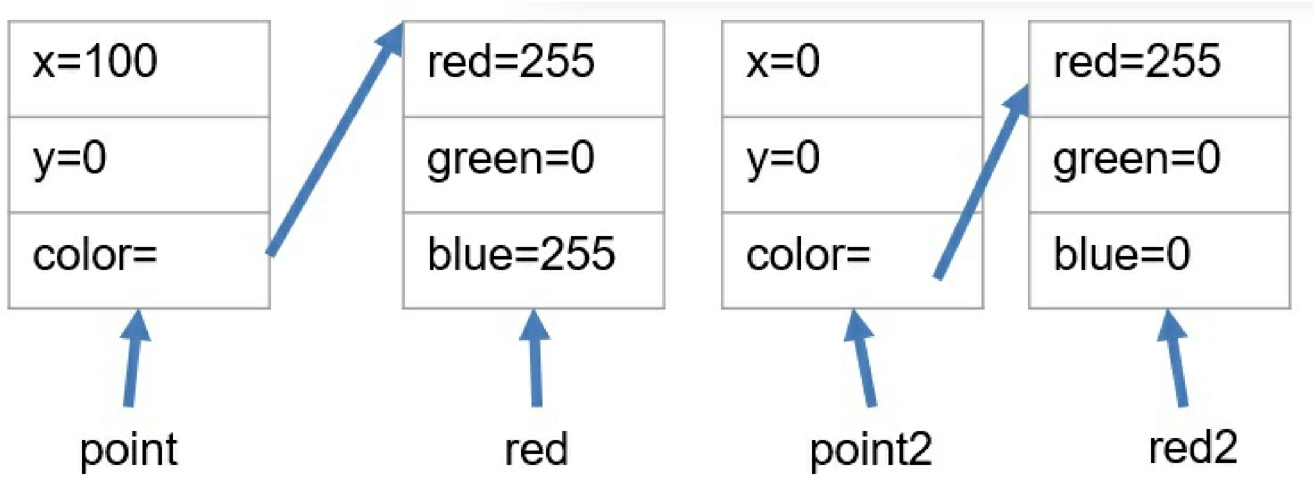
```
import {Color,Point} from 'ch2/drawing1';

let red:Color=new Color(255,0,0);
let point:Point=new Point(0,0,red);
let point2:Point=new Point(point.x,point.y,point.color);
point.x=100;
console.log(point);
console.log(point2);
```



This is a ***shallow copy*** of an object as we are only copying the top level.
This will make a new object, but only copy the top level or primitive types (number, boolean, string). Any deeper objects or arrays still remain as references.

What if we want a ***deep copy***. In other words, each point will, in addition to having a unique memory location for its primitive values, will also have a reference to a different Color object.

```
 import {Color,Point} from 'ch2/drawing1';

let red:Color=new Color(255,0,0);
let red2:Color=new Color(255,0,0);
let point:Point=new Point(0,0,red);
let point2:Point=new Point(point.x,point.y,red2);
point.x=100;
console.log(point);
console.log(point2);
```



This is probably what we wanted. This is called a ***deep copy***. While there are some ways to do this automatically in Typescript, they do not work in all cases, and can be problematic. We can do this manually as in this example, but we will look at better ways later.

Summary

Understanding references and instances is critical in nearly all programming languages. In typescript, every variable whose type is not a primitive type (string, boolean, number) stores a reference to the object. From our examples:

- point2=point; //makes a copy of the reference to the one and only object
- A ***shallow copy*** of the object only copies the top level primitive types, but does not duplicate any contained objects, rather it copies the reference to the same object.
- A ***deep copy*** of the object makes copies of all of the objects, nested objects and primitive types. Gives you a true clone of the object that is independent of the original. Later, we will learn how to clone the object, but for now, we have to create an independent object with the same values.

3.5) This keyword

Data Classes allow us to combine data into a grouping or class and use that grouping as a data type in our programs.

Overview

There is a special keyword ***this*** that can be used from inside the constructor (or any method inside the class) that will allow us access to the member variables of the object.

Abstracting the constructor

Consider our color class

```
 class Color{
   constructor(public red:number, public
green:number,public blue:number){ }
}
```

{: .no-run}

What if instead of passing in values for red, green, and blue, we wanted to pass in a string (either “red”,”green”, or “blue”) to initialize our color to one of these three colors. We can go back to our original syntax and define the members explicitly, and change our constructor to take a string that is not marked with the public or private keywords since we only need it to initialize the members.

```
 class Color{
     public red:number=0;
     public green:number=0;
     public blue:number=0;
     constructor(colorStr:string){
         //what goes here
     }
}
```

{: .no-run}

The idea is that we can use the string to determine how to set the members. We can use the ***this*** keyword to access the member variables of the current instance.

```
 class Color{
    public red:number=0;
    public green:number=0;
    public blue:number=0;
    constructor(colorStr:string){
        if (colorStr==="red"){
            this.red=255;
        } else if (colorStr==="green"){
            this.green=255;
        } else if (colorStr==="blue"){
            this.blue=255;
        }
    }
}
console.log(new Color("red"));
console.log(new Color("green"));
console.log(new Color("blue"));
```

Here we can initialize our members indirectly by using the value of the parameter colorStr. The ***this*** keyword allows us access to our own members from within the instance. If the string is not recognized (i.e. not red, green, or blue) then the default values of (0,0,0) remain which is our intention. We would want to make a comment on our constructor that this is the behavior to help users of our class to know how to use it.

Summary

Typescript allows the use of the ***this*** keyword in order to access the members of the current instance of the class. From within the class, using the ***this*** keyword allows us access to all of the member properties (public or private) within the class instance.

Chapter Summary

Now we have the ability to create complex data types of our own using the class keyword. These data types can contain any other type of object including another class, a primitive type, or an array. There is no limitation on what the array or embedded class contain (other class objects, arrays of primitives, arrays of other class objects, etc.) We have a special method in our objects called a constructor. The constructor can be used to initialize our object, or by using the public and private keywords, it can define members of our object. Parameters without these keywords behave just like parameters to any other function, but with these keywords, that parameter also becomes a member of the object. We can access the members of our class instance using the ***this*** keyword.

4) Classes

4.1) Class Methods

Classes allow us to combine data and methods into a grouping or class and use that grouping as a data type in our programs.

In addition to properties and constructors which we saw in our discussion of ***Data Classes***, generalized classes in typescript can also contain functions (called methods) that can access both *public* and *private* members of the class.

A class with methods can be viewed as a self-contained entity which ***encapsulates*** some concept, allowing us to use the class without knowing anything about its internal structure or implementation.

> ***Encapsulation*** *is a key concept of this course. The idea of creating reusable, self contained types which contain both data, and functions ot operate on that data is central to Object-Oriented Programming*

Adding functionality to a class

Let’s consider our drawing example from the previous chapter

```
 class Color{
    constructor(public red:number, public
green:number,public blue:number){ }
}
class Point{
    constructor(public x:number,public y:number,public
color:Color){}
}
class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
}
class Rectangle{
    constructor(public corner1:Point,public
corner2:Point,public color:Color){}
}
```

{: .no-run}

Specifically, if we look at our Line class which contains two points with x and y coordinates, we might want an easy way to get a line's length. We can expand our definition of a line to contain a method to accomplish this. The `getLength()` method can be added inside the class definition.

```
 class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}

    getLength():number{
        let x=this.start.x-this.end.x;
        let y=this.start.y-this.end.y;
        let len:number=Math.sqrt(x*x+y*y);
        return len;
    }
}
```

{: .no-run}

> *Note that we don't need to know how the line is represented to use this method. If we have a line and want it's length, we simply call the getLength method. This is important because in the future we might change the internal representation of a line, but this method would still work if we rewrote it. The calling program would not need to change.*

Let's try it:

```
 import {Color,Point} from 'ch4/drawing1';

class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
    public getLength():number{
        let x=this.start.x-this.end.x;
        let y=this.start.y-this.end.y;
        let len:number=Math.sqrt(x*x+y*y);
        return len;
    }
}
let myLine:Line=new Line(new Point(0,0,new
Color(0,0,0)),new Point(100,100,new Color(0,0,0)),new
Color(255,0,0));
let lineLen:number=myLine.getLength();
console.log(lineLen);
```

We can add as many methods as we want to a class. The methods allow us to manipulate the data within the class or do calculations using the data within the class without knowing how the data within the class is actually represented.

The method itself must obviously know, but external code that uses the class does not need to know anything about the internal structure.

Later we will use the ***private*** keyword to hide that information from users of the class.

Our class will have a ***public interface*** which may be separate from its private internal representation.

Default Parameters

It is possible to provide default values for the parameters in the function signature. We can use this to provide default values for our color class. Now we can create a color object with these default values. In the example below, we create a color with a specific color, and one using the defaults (0,0,0).

```
 class Color{
    constructor(public red:number=0, public
green:number=0,public blue:number=0){ }
}
let specificColor:Color=new Color(255,128,44);
let defaultColor:Color=new Color();
console.log(specificColor);
console.log(defaultColor);
```

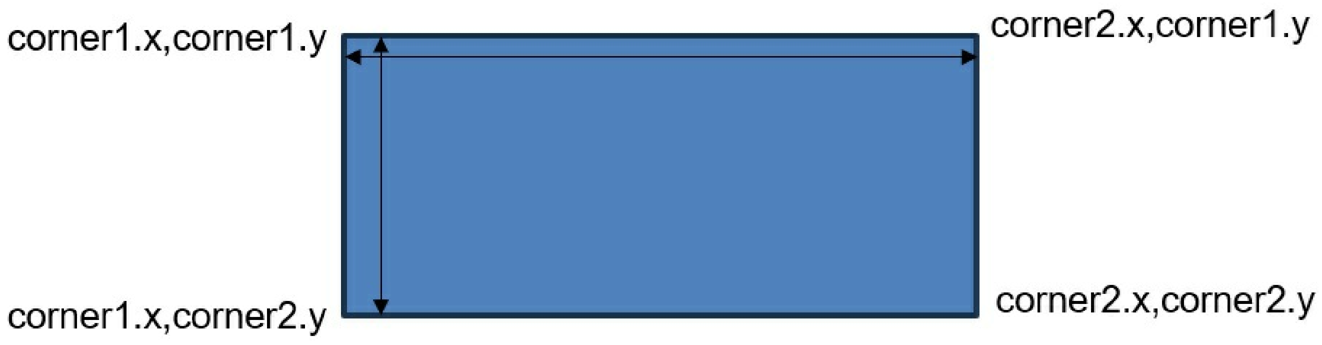
Another example

Let's try to add a `getArea()` method to our rectangle class. This should be straight forward since we have the corners.

```
 class Rectangle{
    constructor(public
corner1:Point,corner2:Point,color:Color){}

    getArea():number{
        //we want to multiply width * height, but we
already have a way to get the width and the height
        //using our line class from before.
        //Our width is (this.corner1.x,this.corner1,y) to
(this.corner2.x,this.corner1.y)
        //Our height is (this.corner1.x,this.corner1.y) to
(this corner1.x,this.corner2.y)
        //make lines for the top and left of the rectangle,
and get there lengths, and multiply them together.
    }
}
```

{: .no-run}

The area is the length of the line from (corner1.x,corner1.y) to (corner2.x,corner1.y) times the length of th eline from (corner1.x,corner1.y) to (corner1.x,corner2.y)



```
 import {Color,Point} from 'ch4/drawing1';

class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
    getLength():number{
        let x=this.start.x-this.end.x;
        let y=this.start.y-this.end.y;
        let len:number=Math.sqrt(x*x+y*y);
        return len;
    }
}
class Rectangle{
    constructor(public corner1:Point,public
corner2:Point,public color:Color){}

    getArea():number{
        let corner3:Point=new
Point(this.corner2.x,this.corner1.y,new Color());
        let corner4:Point=new
Point(this.corner1.x,this.corner2.y,new Color());
        let horizLine:Line=new
Line(this.corner1,corner3,new Color());
        let vertLine:Line=new Line(this.corner1,corner4,new
Color());
        let
area:number=horizLine.getLength()*vertLine.getLength();
        return area;
    }
}
let rect:Rectangle=new Rectangle(new Point(0,0,new
Color()),new Point(100,100,new Color()),new Color());
console.log(rect.getArea());
```

Exercises

Fill in the method `getDiagonals()`, `getPerimeter()`, and `getDiagonalLength()` methods as specified in the comments.

```
 import {Color,Point} from 'ch4/drawing1';

class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
    getLength():number{
        let x=this.start.x-this.end.x;
        let y=this.start.y-this.end.y;
        let len:number=Math.sqrt(x*x+y*y);
        return len;
    }
}
class Rectangle{
    constructor(public corner1:Point,public
corner2:Point,public color:Color){}

    getArea():number{
        let corner3:Point=new
Point(this.corner2.x,this.corner1.y,new Color());
        let corner4:Point=new
Point(this.corner1.x,this.corner2.y,new Color());
        let horizLine:Line=new
Line(this.corner1,corner3,new Color());
        let vertLine:Line=new Line(this.corner1,corner4,new
Color());
        let
area:number=horizLine.getLength()*vertLine.getLength();
        return area;
    }
    /**
         * Return an array of line objects which represent
the two diagonals of the rectangle.
         * @param none
         * @returns An array of 2 points representing the
diagonals.  The first point in the array should be top
         * left to bottom right.  The second point should
be top right to bottom left.
         * @sideEffects None
    */
    getDiagonals():Line[]{
    }
    /**
         * Return the length of the diagonal of the
rectangle.
         * @param none
         * @returns The length of the diagonal of the
rectangle.
         * @sideEffects None
    */
    getPerimeter(): number {
    }
    /**
       * Return the length of the diagonal of the
rectangle.
       * @param none
       * @returns The length of the diagonal of the
rectangle.
       * @sideEffects None
    */
    getDiagonalLength():number{
    }
}
let rect:Rectangle=new Rectangle(new Point(0,0,new
Color()),new Point(100,100,new Color()),new Color());
console.log(rect.getDiagonals());
console.log(rect.getPerimeter());
console.log(rect.getDiagonalLength())
```

▶ Show Solution

One thing to notice is that we had to compute the missing corners in every function. It would make more sense to compute them when the object is created and store them as member variables. We can do this without changing the ***public interface*** of the class and simplify all of our member methods. We will do this in the next section.

So now we can add methods to our classes to create robust objects that encapsulate not just some heterogeneous data, but also methods that can work on that data.
We can use the classes to create instances with the new operator which store their own data, and have methods that work on the data inside the instance.

```
 let color1=new Color(0,0,0);
let color2=new Color(255,255,255);
color1.red=255;
```

{: .no-run}

> *NOTE: color2 is unchanged. It is a distinct instance of our class Color.*

Summary

Classes in typescript can contain only data (Data Classes) or they can contain a combination of data and methods that operate on that data. The methods can access the properties of the class instance by using the ***this*** keyword. In this way, we can create classes that not only combine data that goes together, but also encapsulate it with the methods that act upon that data.

4.2) Data Hiding

Classes allow us to combine data and methods into a grouping or class and use that grouping as a data type in our programs.

Data Hiding

Consider our rectangle class again:

```
 class Rectangle{
    constructor(public corner1:Point, public
corner2:Point,public color:Color){ }
}
```

{: .no-run}

We made all of the member variables (properties) public for simplicity, but now we cannot change the internal representation.
Making members private hides them from everything outside the class making them inaccessible.
We can rewrite this class making our point members private.

```
 class Rectangle{
    constructor(private corner1:Point, private
corner2:Point,public color:Color){}
}
```

{: .no-run}

Nothing changes except we cannot access corner1 and corner2 outside our class, but our methods (diagonal, area, perimeter) that we wrote in the exercise in the previous chapter are fine because they are inside the class.
We can still create a rectangle and call our methods on it, we just can't get the corners any more. If we really need them, we can write methods to get them or change them.

> *But Why? Imagine we wrote this for a client, and suddenly after we have written a 100,000 line drawing program they want us to add the ability to rotate a rectangle.*
> *Our implementation DOES NOT ALLOW THIS!!!.*
> *Also, many of the methods we wrote required us to compute the missing corners. If we stored all 4 corners, then we could do all of these things without breaking the 100,000 lines of external code.*

We can make the change easily without breaking anything outside our code. We will renumber the corners from the upper left clockwise for simplicity. Note that we do not change the ***signature*** of the constructor, only the hidden data.

```
 class Rectangle{
    private corner2:Point;
    private corner4:Point;
    constructor(private corner1:Point, private
corner3:Point,public color:Color){
        this.corner2=new Point(corner3.x,corner1.y,color);
        this.corner4=new Point(corner1.x,corner3.y,color);
    }
}
```

{: .no-run}

> *Nothing is changed in how you create instances of this class, but now we have all 4 points stored. Now we could add a rotate method if we choose.*

Because we relabled our corners, and added the new corners, we should rewrite all of the internal methods (but we won't change the signature of the method).

Here is a complete working example:

```
 import {Color,Point} from 'ch5/drawing1';

class Line{
    constructor(public start:Point,public end:Point,public 
color:Color){}
    getLength():number{
        let x=this.start.x-this.end.x;
        let y=this.start.y-this.end.y;
        let len:number=Math.sqrt(x*x+y*y);
        return len;
    }
}
class Rectangle{
    private corner2:Point;
    private corner4:Point;
    constructor(private corner1:Point, private 
corner3:Point,public color:Color){
        this.corner2=new Point(corner3.x,corner1.y,color);
        this.corner4=new Point(corner1.x,corner3.y,color);
    }
    getArea():number{
        let horizLine:Line=new 
Line(this.corner1,this.corner2,new Color());
        let vertLine:Line=new 
Line(this.corner1,this.corner4,new Color());
        let 
area:number=horizLine.getLength()*vertLine.getLength();
        return area;
    }
    /**
         * Return an array of line objects which represent 
the two diagonals of the rectangle.
         * @param none
         * @returns An array of 2 points representing the
```

```
diagonals.  The first point in the array should be top
         * left to bottom right.  The second point should
be top right to bottom left.
         * @sideEffects None
    */
    getDiagonals():Line[]{
        let result=[
            new Line(this.corner1,this.corner3,new
Color()),
            new Line(this.corner4,this.corner2,new
Color()),
        ];
        return result;
    }
    /**
         * Return the length of the diagonal of the
rectangle.
         * @param none
         * @returns The length of the diagonal of the
rectangle.
         * @sideEffects None
    */
    getPerimeter(): number {
        let horizLine:Line=new
Line(this.corner1,this.corner2,new Color());
        let vertLine:Line=new
Line(this.corner3,this.corner4,new Color());
        return
horizLine.getLength()*2+vertLine.getLength()*2;
    }
    /**
       * Return the length of the diagonal of the
rectangle.
       * @param none
       * @returns The length of the diagonal of the
```

```
rectangle.
         * @sideEffects None
     */
    getDiagonalLength():number{
        let diags:Line[]=this.getDiagonals();
        return diags[0].getLength();
    }
}
let rect:Rectangle=new Rectangle(new Point(0,0,new
Color()),new Point(100,100,new Color()),new Color());
console.log(rect.getDiagonals());
console.log(rect.getPerimeter());
console.log(rect.getDiagonalLength())
```

Summary

Data hiding is an important tool for object oriented programming. It allows us, as the programmer, to decide what functionality, methods, and data we expose to the users of our class without worrying about things we have hidden inside.

If we provide a ***public interface*** to our class that is consistent, then we should try not to change it, but anything that is private can be changed so long as we make sure that the ***public interface*** still works as expected without breaking anything that uses our class.

4.3) Object Cloning

Classes allow us to combine data and methods into a grouping or class and use that grouping as a data type in our programs.

Types of copies

Recall from the previous chapter the discussion of copying.

- `point2=point;` //makes a copy of the ***reference*** to the one and only object
- A ***shallow copy*** of the object only copies the top level primitive types, but does not duplicate any contained objects, rather it copies the reference to the same object. For arrays, we can use the spread operator (...) to do this.
- A ***deep copy*** of the object makes copies of all of the objects, nested objects and primitive types. Gives you a true clone of the object that is independent of the original. Later, we will learn how to clone the object, but for now, we have to create an independent object with the same values.

A ***deep copy*** of the object makes copies of all of the objects, nested objects and primitive types. Gives you a true clone of the object that is independent of the original. Later, we will

learn how to clone the object, but for now, we have to create an independent object with the same values.
How do we do this in a structured way?

- We teach each class how to clone itself, and then use those methods if we have a class that contains another class.
- We will work from the bottom up of our hierarchy of classes. The simplest of which is our color class.

Consider the Color class we have been working with. Cloning that is eash as a ***shallow copy*** is sufficient. The classes data items are all primitive types (numbers).

```
 class Color{
    constructor(public red:number=0, public
green:number=0,public blue:number=0){ }
    clone():Color{
        return new Color(this.red,this.green,this.blue);
    }
}
let red=new Color(255,0,0);
let blue=red.clone();
blue.red=0;
blue.blue=255;
console.log(red,blue);
```

> *We can create a new color object from an existing one by calling the existing one's clone method.*

Our point method is more difficult in that it contains a Color object. Here a ***deep copy*** is required to not only copy the point object into a new instance, but also create a new instance of the color object. Luckily the color object already has a clone method.

```
 import {Color} from 'ch5/drawing2';

class Point{
    constructor(public x:number,public y:number,public
color:Color){}
    clone(): Point{
        return new Point(this.x,this.y,this.color.clone());
    }
}
let p=new Point(5,5,new Color());
let q=p.clone();
q.color=new Color(255,255,255);
q.x=0;
console.log(p,q);
```

> *Note, if we passed the color, we would get a reference to the same color object, but by calling its clone method, we get a new one (since we wrote it that way).*

Likewise, we can add a clone method to our Line class as well. Again, since this class contains references to objects, we must ***deep copy*** the line class. Luckily each of the object types (color and line) already has a clone method we can use.

```
 import {Color} from 'ch5/drawing3';

class Line{
    constructor(public start:Point,public end:Point,public
color:Color){}
    clone():Line{
        return new
Line(this.start.clone(),this.end.clone(),this.color.clone()
);
    }
}
let line=new Line(
    new Point(0,0,new Color()),
    new Point(100,100,new Color()),
    new Color()
);
let line2=line.clone();
line2.color.red=255;
line2.start.x=5;
console.log(line,line2);
```

We can easily do the same for our Rectangle and Polygon classes. For the rectangle class

```
 import {Color,Point} from 'ch5/drawing3';

class Rectangle{
   private corner2:Point;
   private corner4:Point;
   constructor(private corner1:Point, private
corner3:Point,public color:Color){
       this.corner2=new Point(corner3.x,corner1.y,color);
       this.corner4=new Point(corner1.x,corner3.y,color);
   }
   clone():Rectangle{
      return new
Rectangle(this.corner1.clone(),this.corner3.clone(),this.co
lor.clone());
   }
}
let rect=new Rectangle(
    new Point(0,0,new Color()),
    new Point(100,100,new Color()),
    new Color()
);
let rect2=rect.clone();
rect2.color.red=255;
console.log(rect,rect2);
```

For the polygon class, things are a little trickier. The class contains an array of references to Point. If we use the spread operator to create a new array, we will only get a ***shallow copy*** and the individual points will reference the same Point objects as the original Polygon. We will need to iterate through the array and clone the objects indivisually to create a new ***deep copy*** of the array to use in our cloned object.

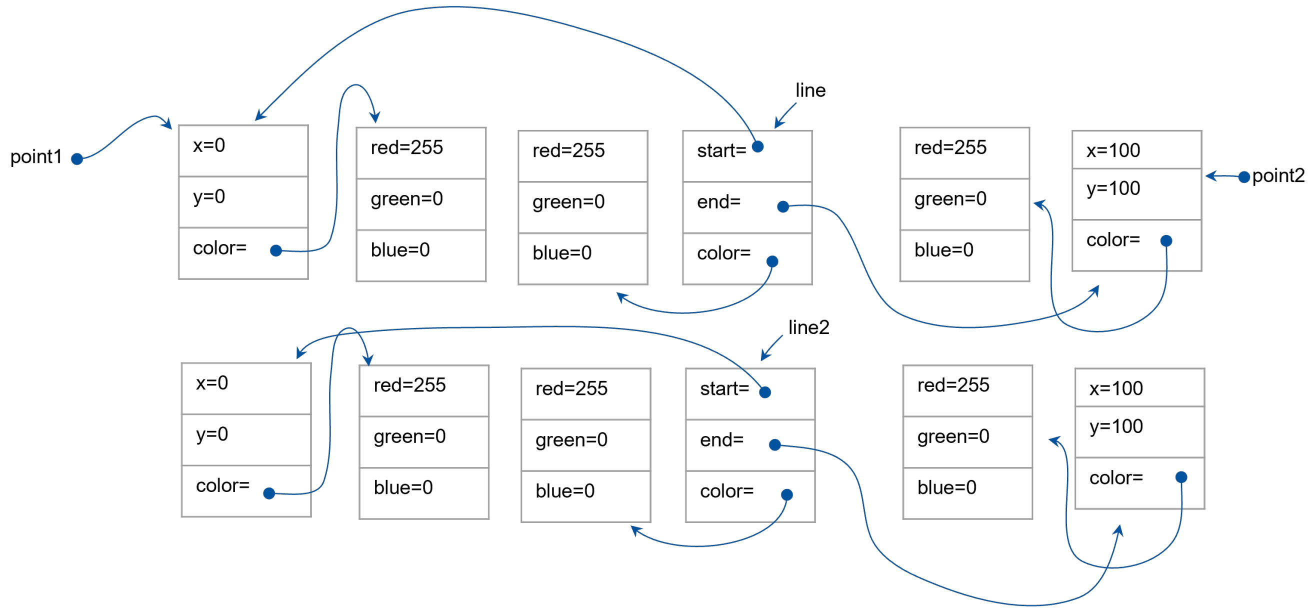
```
 import {Color,Point} from 'ch5/drawing3';

class Polygon{
    constructor(public points:Point[],public color:Color){}
    clone():Polygon{
        let newPoints:Point[]=[];
//initialize a new empty array.
        for (let point of this.points){
            newPoints.push(point.clone());      //don’t
push the point, push a clone of it.
        }
        // so newPoints is a new array containing clones of
all the points in this polygon.  We can pass it directly
since it is completely new.
        return new Polygon(newPoints,this.color.clone());
    }
}
let pts=[
    new Point(0,0,new Color()),
    new Point(100,0,new Color()),
    new Point(100,100,new Color()),
    new Point(100,0,new Color())
];
let poly=new Polygon(pts,new Color());
let poly2=poly.clone();
poly2.color.red=255;
console.log(poly,poly2);
```

Understanding memory layouts

Let's consider how using clone affects the layout of our objects in memory. This can be a good way to understand what is going on in your program.

```
 let point1: Point = new Point(0, 0, new Color(255,0,0));
let point2: Point = new Point(100, 100, new
Color(255,0,0));
let line: Line = new Line(point1, point2, new
Color(255,0,0));
let line2: Line = line.clone();
```

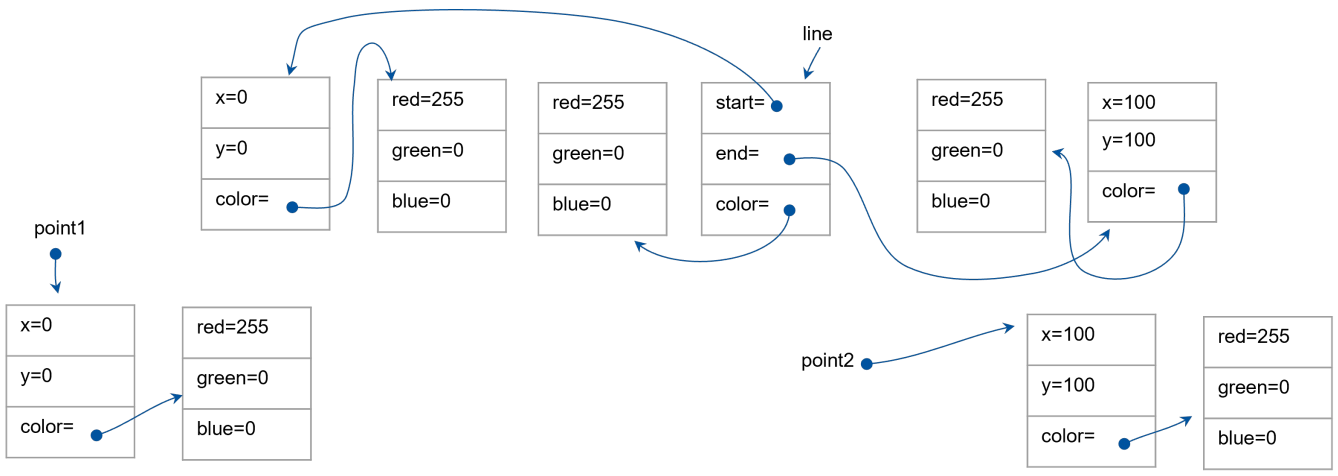
{: .no-run}



Notice point1 and point2 are still the same references as we have in line. We can clone the points making them distinct.

```
 let point1: Point = new Point(0, 0, new Color(255,0,0));
let point2: Point = new Point(100, 100, new
Color(255,0,0));
let line: Line = new Line(point1.clone(), point2.clone(),
new Color(255,0,0));
```

{: .no-run}



> *By using our clone methods in all of our classes, this code now has each element of each class as a distinct instance.*

Summary

The simplest way to ensure deep cloning is to *teach* each class how to deep copy itself. If we do this then classes that contain the class in question can just call its clone method to deep copy it.

Chapter Summary

In addition to storing data (Data Classes), classes can also contain methods. These methods can operate on the data within the class without regard to its visibility. We can change the visibility of a member property or method with the ***public/private*** keywords. Anything marked as public is accessible outside of the class instance. Anything marked as private can only be accessed within a method inside that class.

5) Composition and Inheritance

5.1) Composition

Using ***composition***, we can build complex objects in order to define new types that has a ***contains a*** relationship with some existing type.

Composition in Typescript

So far we have examined classes which contain both data and methods. We can combine classes by including another class as a member of our class

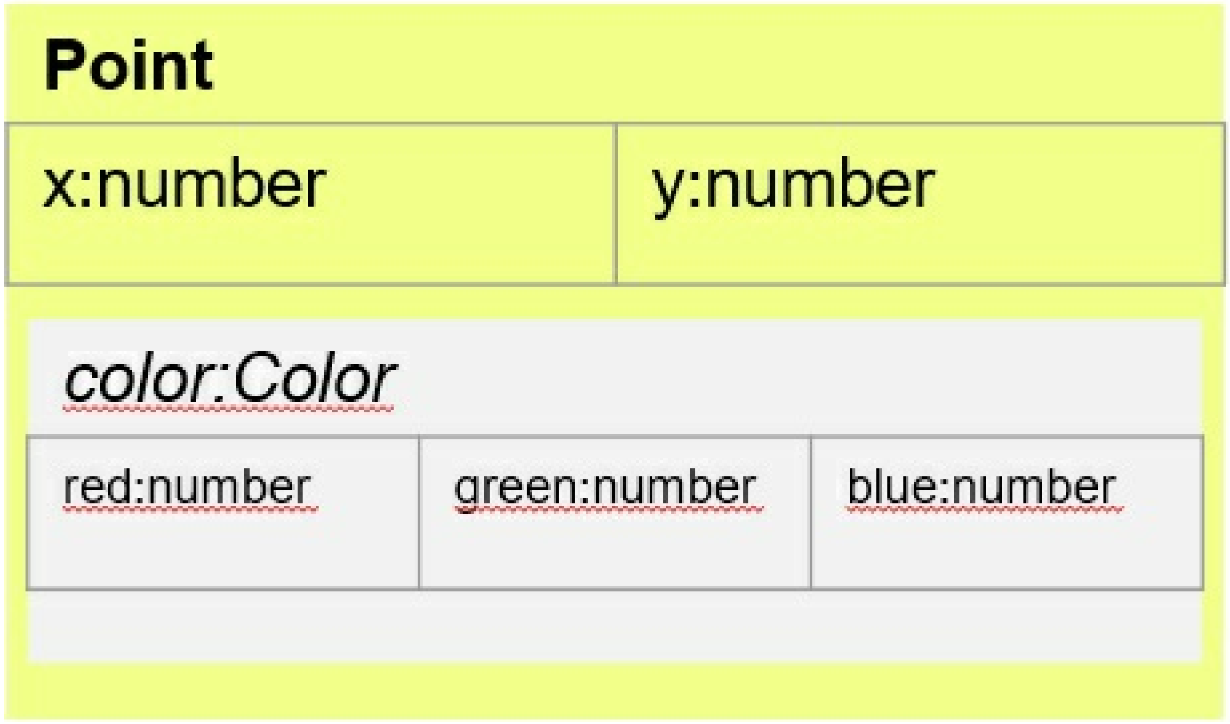
- Consider the Point class which contains an instance of the color class.
- Consider the Rect class which contains instances of our color class and 2 point classes

This method of combining classes to produce other classes is known as ***composition*** because we are adding classes as members of our new class.
This is a powerful tool for building classes, as it allows us to compartmentalize concepts (like color, or point) then use them to build more complex concepts.

Understanding the Relationship

The important thing here is the relationship with composition:

- In general, if a concept that a class (Class1) represents is a part of another class (Class2), then we add Class1 to Class2 as a member variable (property).
- We could also say that if Class2 contains Class1,



> *Note that the instance of Color is inside Point. This makes sense since the point has a Color.*

Recognizing the relationship between concepts that are to be represented as Classes is critical to Object Oriented Programming. Here are some simple examples:

- A Car *has a* tire. If we have a tire class, we can represent a car by ***composition***. We would add 4 (or 5) tire instances to our car class.
- A course *has a* final exam. If we had an exam class, we can represent a course by ***composition***. We would add an instance of our exam class to course.
- A classroom *has* desks. If we had a desk class, we can represnet a classroom by ***composition***. We would add 1 or more instances of our desk class to our classroom.
- A fruit basket *has* fruit. The following example shows how we use ***composition*** to represent a basket of fruit by adding an array of fruit to our basket class.

```
 class Fruit{
  constructor(public type:string,public color:string,public
price:number){};
}
class FruitBasket{
  constructor(private fruits:Fruit[],private
basketCost:number){}
  public getPrice(){
    let sum:number=0;
    for (let fruit of this.fruits){
      sum+=fruit.price;
    }
    return sum+this.basketCost;
  }
}
let basket:FruitBasket=new FruitBasket(
  [new Fruit("apple","red",.50),new
Fruit("orange","orange",.92),new
Fruit("lemmon","yellow",1.50)],4.00
);
//expect 6.92
console.log(basket.getPrice());
```

Composition allows us to reuse our fruit class for various types of fruit and combine them into a basket. Our basket can then expose public methods (like getPrice() which have access to the member fruits) to sum up the price of all the fruits, add it to the price of the basket, and return a total price which is dependent on the fruits inside.

Summary

Composition gives the programmer the ability to represent a ***has a*** or a ***contains*** relationship. The relationship is the key to understanding when to use ***composition*** over other methods.

5.2) Inheritance

Using ***Inheritance***, we can build complex hierarchies of objects in order to define new types that are a ***type of*** some existing type.

Understanding the relationship

In the previous section we discussed ***composition*** which allowed us to represent a *contains* or *has a* relationship between two classes. Recall that a course has a final and a fruit basked contains fruit. While useful in many situations, we often wnat to represent a ***type of*** relationship. In typescript, the `extend` keyword allows us to represent a class in terms of another class that it is a *type of*.

Consider the following:

- An apple is a type of fruit.
- A car is a type of vehicle.
- A triangle and a rectangle are types of polygons (more on this later)
- In a University computer system, a student and a faculty member are both types of Users.

Why inheritance

We can inherit the properties and methods of an existing class and extend that class by either adding new members, or replacing the functionality of existing members to suit the new object's needs.

Suppose I have a class *Users* that represents a system user on a University's central IT system.

```
 class Users {
  constructor(private name: string,private age: number) {}
  public getName(): string {
    return this.name;
  }
  public getAge(): number {
    return this.age;
  }
}
```

{: .no-run}

This class has private properties name and age, and two functions to retrieve the values in these properties. In other words, users of the class CANNOT change the name or age, but they can retrieve them.

Now suppose I want to create two new classes called Students and Faculty. I want them to have all of the abilities of a User, but they also need some additional capabilities based on the type.

> *It is extremely important to note that a Student, does not contain a User, the Student is a User. We cannot say this about points and colors. A point is a color? That makes no sense.*
> *A student is a user, that makes sense.*

So how do we deal with this type of relationship between classes?

We can extend an existing class when the relationship between the objects is an ***is a*** relationship. Our new classes act like the old class unless we add some functionality to it.

```
 class Student extends Users {
}
class Faculty extends Users {
}
```

{: .no-run}

We can now define objects of type Student and Teacher, and instantiate them with new and they work just like our Users class.

```
 class Users {
  constructor(private name: string,private age: number) {}
  public getName(): string {
    return this.name;
  }
  public getAge(): number {
    return this.age;
  }
}
class Student extends Users {
}
class Faculty extends Users {
}
let collegeStudent = new Student("John", 20);
let teacher = new Faculty("Jane", 30);
console.log(collegeStudent.getName(),
collegeStudent.getAge());
console.log(teacher.getName(), teacher.getAge());
```

While all Users share some things in common,there are a lot of things that are unique to being a student or Faculty.

- Students have a gradTerm and a gpa. They are still users, but they are a ***type of*** user.
- Faculty has a department, an office, and a list of classes they teach. Again, they are still a ***type of*** user.

```
 class Student extends Users {
  private gradTerm: string='';
  private gpa:number=0;
}
class Teacher extends Users {
  private department:string='';
  private classes: string[]=[];
  private office: string='';
}
```

{: .no-run}

In more formal terms, the Student class ***inherits*** from the Users class.
We say that Student is a ***subclass*** of Users and that Users is a ***superclass*** of Student (and Faculty).
Implementing this sort of relationship (type of, is a, etc.) in this manner is referred to as ***inheritance***.
We inherit everything about the superclass, but still are a distinct type with our own properties and methods in addition to those in the ***subclass***.

> *The **superclass** is often referred to as the **base class** of the relationship.*

If we want to create a constructor to initialize our object, we must remember that it is a User so its constructor must also be responsible for the name and age fields from the parent or superclass, otherwise, how would they ever get set?

It is easy to initialize gradTerm and gpa, but how do we initialize the members from the superclass?

```
 class Student extends Users {
    private gradTerm: string = "";
    private gpa: number = 0;
    constructor(name: string, age: number, gradTerm:
string, gpa: number) {
      //SOMEHOW WE HAVE TO INITIALIZE THE SUPERCLASS (or
PARENT)
      this.gradTerm = gradTerm;
      this.gpa = gpa;
  }
}
```

{: .no-run}

We can call the superclass' constructor within our constructor by calling the super() method. This will take the same arguments as the constructor of the superclass.
Here those arguments are name and age. This calls the constructor in Users which takes care of its part of the initialization.

```
 class Student extends Users {
    private gradTerm: string = "";
    private gpa: number = 0;
    constructor(name: string, age: number, gradTerm:
string, gpa: number) {
      //calling super as the first line of our constructor
initializes the superclass by calling its constructor.
      super(name,age);
      //After it is initialized, we can then initialize our
member variables as usual
      this.gradTerm = gradTerm;
      this.gpa = gpa;
  }
}
```

{: .no-run}

Here is a completed example:

```
 class Users {
  constructor(private name: string,private age: number) {}
  public getName(): string {
    return this.name;
  }
  public getAge(): number {
    return this.age;
  }
}
class Student extends Users {
  private gradTerm: string = "";
  private gpa: number = 0;
  constructor(name: string,age: number, gradTerm: string,
gpa: number) {
    super(name, age);
    this.gradTerm = gradTerm;
    this.gpa = gpa;
  }
  public getGradTerm(): string {
    return this.gradTerm;
  }
  public getGPA(): number{
    return this.gpa;
  }
}
class Faculty extends Users {
  private department:string='';
  private classes: string[]=[];
  private office: string='';
  constructor(name: string,age: number,
department:string,classes:string[],office:string) {
    super(name, age);
    this.department=department;
    this.classes=classes;
    this.office=office;
  }
  getDepartment():string{
    return this.department;
  }
  getClasses():string[]{
    return this.classes;
  }
  getOffice():string{
    return this.office;
  }
}
let jan:Student=new Student("Jan",19,"25S",3.95);
let lisa:Faculty=new Faculty("Lisa",42,"Computer Science",
["CISC181","CISC210"],"317 Morris Hall");
console.log(jan.getName()+" has a GPA of "+jan.getGPA())
console.log(lisa.getName()+" is in the
"+lisa.getDepartment()+" department.")
```

> *Both Lisa and Jan can call getNme because it is inherited from Users in both Student and Faculty classes, but only Jan can call getGPA, because it is only defined in the child or subclass Student. Likewise, Lisa can call getDepartment, but Jan can't because it is only defined in the subclass Faculty.*

Another way to think about this is that Teachers and Students share some things in common:

- They both have names
- They both have ages (although Teacher.age > Student.age)

They also have some differences:

- Students have a GPA and a gradTerm
- Faculty have a department, an office, a list of classes, and don't show up on photographic film.

We encapsulate their commonality in the Users class, then extend Users to make new classes that express the differences.

Summary

Inheritance allows the programmer to represent an ***is a*** or ***type of*** relationship. Using inheritence through the `extends` keyword, we can express both the similarities and differents between objects in these types of relationships. We can call the constructor (we must actually) of our superclass in the constructor of our subclass by calling the `super` method and passing it the same list of parameters we would pass to the ***superclasses*** constructor.

5.3) Putting it all Together

Using ***Inheritance***, we can build complex hierarchies of objects in order to define new types that are a ***type of*** some existing type.

Termiology review

Composition:

- Add a class or array of class as a property to your class.
- Represents a has a relationship

Inheritance:

- Extend an existing class by adding functionality, but keeping the functionality of the original class.
- Represents a is a relationship

The class that we are extending is called the ***superclass*** or sometimes the ***base class*** or ***parent class***
The class that we are creating by extending is called a ***subclass*** or ***child class***.

Back to drawing

Is there something most of our objects have in common?

All of the drawing objects (Point, Line, Rectangle, Polygon) have a Color component. If we create a class with just a color component, we could share that definition in all our drawing classes by extending it.

What should we call our new class?

We want something descriptive that supports the ***is a*** relationship with all the other classes. For this example, I will choose to create a class *Drawable*.

```
 class Drawable {
    public color: Color;
    constructor(color: Color) {
        this.color = color.clone();
    }
    public clone(): Drawable {
        return new Drawable(this.color);
    }
}
```

{: .no-run}
Here is a simple drawable class. It contains a color (***composition***), a clone method, and automatically makes a deep copy of the color object in the constructor.
It just holds our color object, so we will extend this to make all of our other drawables.

```
export class Point extends Drawable {
    constructor(
        public x: number = 0,
        public y: number = 0,
        color: Color = new Color(),
    ) {
        super(color);
    }
    clone(): Point {
        return new Point(this.x, this.y,this.color);
    }
}
```

{: .no-run}

Our point class inherits color from the Drawable class. Our Point constructor calls the constructor for our Drawable class and passes it the color so it can do its initialization (all drawables have a color). It does this by passing color to `super`

> *Notice, that the public interface is unchanged, but we don't have to worry about the color, the drawable does.*

```
 class Line extends Drawable{
    public start: Point;
    public end: Point;
    constructor(start: Point,end: Point,color: Color=new
Color()) {
        super(color); //Must be first thing in constructor
always
        this.start = start.clone();
        this.end = end.clone();
    }
    clone(): Line {
        return new Line(this.start, this.end, this.color);
    }
}
```

{: .no-run}

Our Line class can also inherit from our Drawable class. Again it calls super to initialize the Drawable portion of the object.
Note also that the constructor clones the corner points.
Reminder:

- Drawable is the ***superclass, base class, parent class***
- Line is the ***subclass, child class***

```
 class Polygon extends Drawable{
    public points: Point[],
    constructor(points: Point[], color: Color ) {
        super(color);
        let newPoints=[];
        for (let point of points) {
            newPoints.push(point.clone());
        }
        this.points=newPoints;
    }
    clone(): Polygon {
        return new Polygon(this.points, this.color);
    }
}
```

{: .no-run}

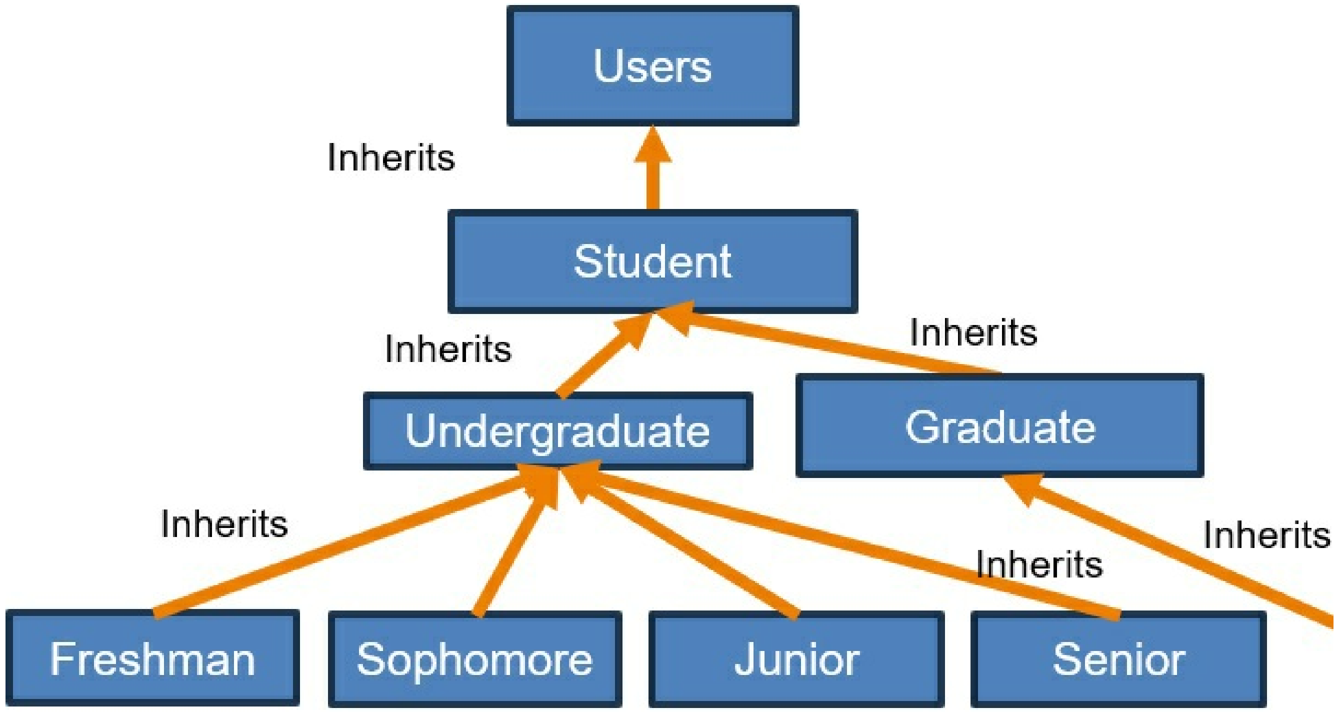
Our polygon class can also inherit from our Drawable class. Again it calls super to initialize the Drawable portion of the object.
Note also that the constructor clones the array of points by cloning each point and pushing them onto a new array before setting the member variable points.

Deeper hierarchies

We can create deeper hierarchies to express these types of relations.

- Everyone is a User

- A Student is a type of User
  - An undergrad is a type of Student
  - A Freshman is a type of Undergrad
    Etc.



The point of inheritance is to capture these types of relationships. Be careful that the relationship you are capturing is a ***type of*** relationship as many inexperienced programmers overuse ***inheritance***, where the relationship really calls for ***composition***.

- A point is not a type of color, so we don't derive point from color.
- An undergraduate is a type of student, so we derive Undergraudate from Student

Summary

Inheritance allows us to capture an ***is a*** relationship between two classes. When a class inherits from a ***superclass***, it gets access to everything in the superclass as well as anything defined within the ***subclass***. We can use this to build complex deep hierarchies where we can represent complex objects by extending existing classes.

Chapter Summary

In this chapter we have introduced two ways to build up a class from other classes.

- If the two classes have an ***has a*** or ***contains*** relationship, then we use ***composition*** by adding member variables to our class of the other classes type. A drawable contains a color by this method.
- If the two classes have a ***type of*** or ***is a*** relationship, then we use ***inheritance*** by extending one class and inheriting all of its members and functionality. A line is a drawable by this method.

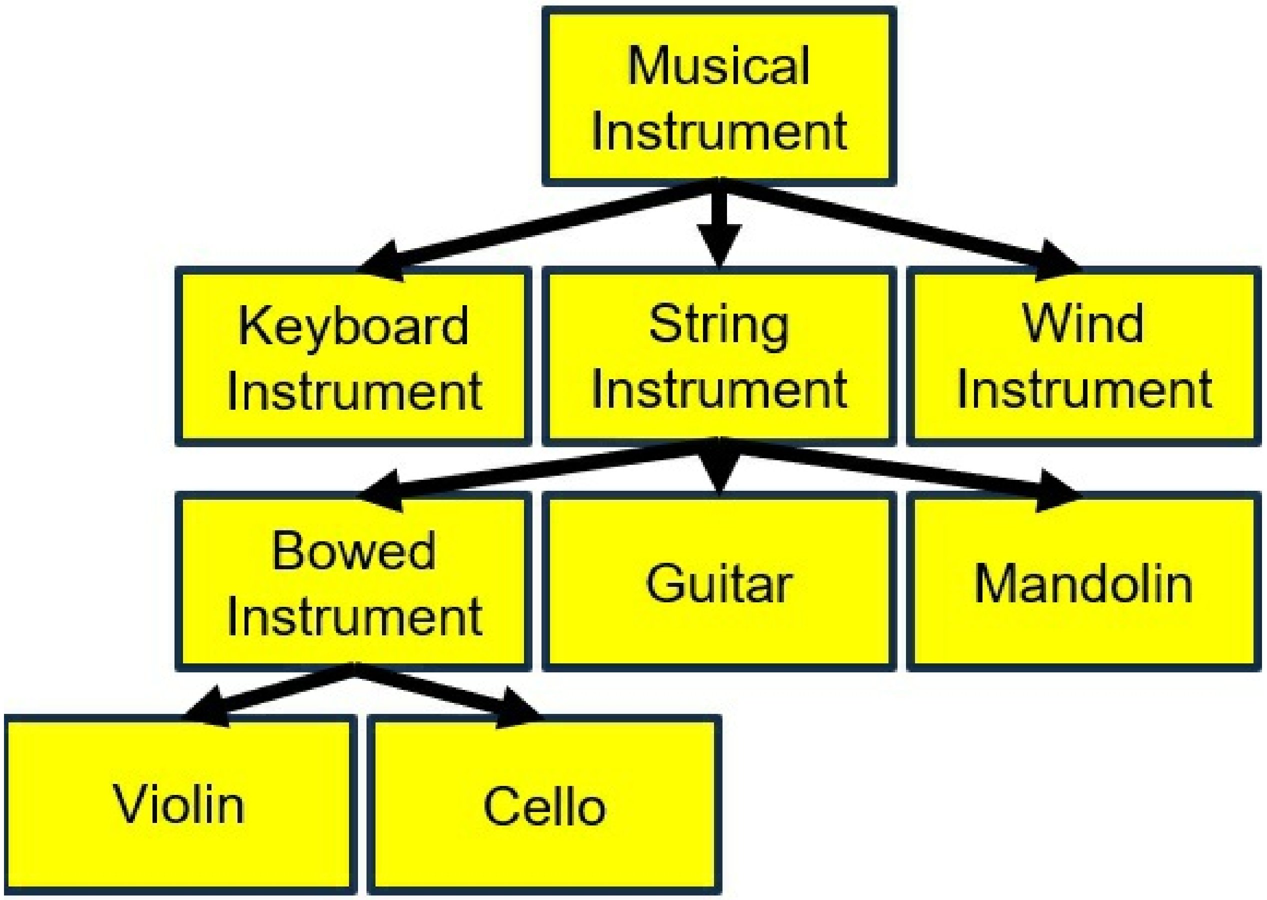
6) Overrides and Polymorphism

6.1) Member Access

We can control ***access*** to the members of a superclass with the ***private, public, and protected*** keywords.

Understanding Inheritance

In the previous chapter we introduced the notion of inheritance to support relationships between concepts that represent an ***is a*** or ***type of*** relationship.
This is different from composition which supports relationships between concepts that represent a ***has a*** or ***contains a*** relationship.



> *Note: Each subclass has a* ***type of*** *relationshiplower with its superclass.*

Assume that the Musical Instrument has a `name`, a `musical key` (ie. C#, Bb), and a `year invented` field as well as a method `getName()` which returns the name of the instrument.

Then all the other classes ALSO have those fields. We don't need to recreate them in our child since we ***inherit*** them from the parent class. This is one of the primary benefits of inheritance.

> *Note: Cellos have a name, key and year field and a getName() method automatically due to inheritance.*

If you can map out the relationships between concepts, then by using a combination of inheritance and composition, we can build complex hierarchies out of simple objects.

Controlling Access

When we create a class, we have options about how that class can be used and inherited.
Fields and methods can be:

- private: Only accessible within the class
- protected: Only accessible within the class or any defined subclasses
- public: Accessible from anywhere (inside or outside the class hierarchy).

By controlling access to properties and methods, we expose to the outside world a minimal set of public properties and methods are exposed. Public items are more difficult to change because others might be using them. Protected are slightly easier and only break classes inherited from us. Changes to private methods effect nothing outside of the class itself.

Let’s briefly go back to our drawing example.
Note that our points are private. This is good in case we want to change how we store polygons without breaking the rest of the code base, but it doesn’t allow us to build other objects from polygon, like triangles, rectangles, etc.

```
 export class Polygon extends Drawable{
  private points: Point[],
    constructor(points: Point[], color: Color ) {
        super(color);
        let newPoints=[];
        for (let point of points) {
            newPoints.push(point.clone());
        }
        this.points=newPoints;
    }
    clone(): Polygon {
        return new Polygon(this.points, this.color);
    }
}
```

{: .no-run}

We can still prevent outsiders from accessing our array of points, while giving access to the array to any subclass of our class by using the ***protected*** keyword.

```
 export class Polygon extends Drawable{
  protected points: Point[],
  constructor(points: Point[], color: Color ) {
    super(color);
    let newPoints=[];
    for (let point of points) {
      newPoints.push(point.clone());
    }
    this.points=newPoints;
  }
  clone(): Polygon {
    return new Polygon(this.points, this.color);
  }
}
```

{: .no-run}

The points array is still not available to the outside world, and changing it would only affect the subclasses we create from Polygon (like rectangle and triangle), but users of our classes will not see a change. They still will not be able to access the points array just like before.

Now we can simplify the rectangle class by recognizing that a rectangle is a type of polygon. Because all of the members are private (i.e. not being used by anyone outside our class), we can change those members without fear of breaking other code.

```
 class Rectangle extends Polygon{
  constructor(corner1: Point, corner3: Point, color: Color)
{
    super([
      corner1,
      new Point(corner3.getX(), corner1.getY()),
      corner3,
      new Point(corner1.getX(), corner3.getY()
    ], color);
  }
  clone(): Rectangle {
    return new Rectangle(this.corner1, this.corner3,
this.color);
  }
}
```

{: .no-run}

Notice that now we are deriving from Polygon instead of Drawable. Because a polygon can already represent a rectangle, we don’t need any other properties (we can delete the corners).

We call the superclasses constrctor with the array of points for the particular 4 sided polygon that this rectangle represents.
We would need to rewrite the area, perimeter and diagonals methods to use our new implementation, but users of our class will see no change in how they use it.

Because we are passing the points to the Polygon constructor, and that constructor clones the points when it builds the member variable points, we do not need to do it here. It would work if we did, but we would have short lived, unnecessary copies of the points in memory.
Knowing how the parent works informs how we write the subclass.

> *If no **access specifier** (public,private,protected) is given, the compiler will default to public.*

Important points on experience

Some important points on inheritance.

- You do not need to reimplement the properties of the parent class as you are inheriting them.
- super(…) calls the constructor of the parent class and takes whatever arguments the parent constructor takes.
- If a member is public or protected, you can access it in the subclass, if it is private, you cannot, but it is still there.

When we subclass, we get all of the properties of our parent class and can access them if they are public or protected.
For methods (i.e member functions), the same holds true based on if they are public, protected, or private. We get the functions in the superclass.

Summary

We can control access to the members of a class (both properties and methods) by using the ***public, private, and protected*** keywords. ***Public*** members are accessible to all, ***private*** members are only accessible within the class, and ***protected*** members are accessible in the class and in any subclass of the class.

6.2) Overrides

We can ***override*** a method in our subclass by creating a method with the same signature as a method in our superclass.

Altering functionality

In the previous sections, we learned that when we ***inherit*** or ***subclass*** a class, we get all of its methods (i.e. functions). Sometimes this is not what we want.

Let’s consider that we want to add a getArea method to all of our drawable classes. This doesn’t really make sense for Drawable and Line, but does for the rest. The calculation is, however, very different.
If we add a default getArea method to our Drawable with the same signature as it has elsewhere in the class hierarchy, then objects that do not implement getArea, will inherit the default behavior, and objects that define the method will get the new behavior

```
 class Drawable {
  public color: Color;
  ...
  ...
  ...
  getArea(): number {
    console.log("This object does not have an area");
    return 0;
  }
}
```

{: .no-run}

If a subclass implements getArea (like rectangle, circle, and triangle), then the version in the subclass is used, otherwise, the version in the base class is used. This is called ***overriding*** a class method.

Consider a new class for the drawing example. A circle:

```
 class Circle extends Drawable {
  private center: Point;
  private radius: number;
  ...
  ...
  ...
  getArea(): number {
    return Math.PI * this.radius * this.radius;
  }
}
```

{: .no-run}

Now if the object is a circle, we get its area. If the object is a line, we get the message, and a value of 0. If we add getArea to the drawables that make sense, then only those classes that do not override getArea will use the implementation in the superclass.
If it is implemented in the subclass, then the subclass version will be used.
Overriding of methods is a powerful tool to express different behaviors in subclasses, while allowing us to have a default implementation.
We can even call the superclass implementation from our overridden method.

We can build in some default behaviors to our superclasses, and override those behaviors in our subclasses if it makes sense, or just use the superclass implementation if it is sufficient.

An Example

Here is an example of an overridden method that calls the parent’s version of the method, but then adds some functionality of its own.

> *Notice the code super.getDescription()*
> *While we user super() to call the constructor of the superclass, we can use super.methodname() to call any method on the superclass even if it is overridden.*

```
 class Fruit{
  constructor(private name: string){}
  public getDescription(): string{
    return `This is a fruit called ${this.name}`;
  }
}

class Orange extends Fruit{
  constructor(protected subType:string){
    super("orange");
  }
  public getDescription(): string{
    return super.getDescription() + " of type " +
this.subType;
  }
}
class Apple extends Fruit{
  constructor(protected subType:string){
    super("apple");
  }
  public getDescription(): string{
    return super.getDescription() + " of type " +
this.subType;
  }
}
```

```
let apples:Apple[]= [new Apple("red"), new Apple("green")];
let oranges:Orange[] = [new Orange("blood"), new
Orange("navel")];
for (let apple of apples){
  console.log(apple.getDescription());
}
for (let orange of oranges){
  console.log(orange.getDescription());
}
```

With our current knowledge we need to make an array of Orange objects, and an array of Apple objects, then iterate through them independently. In the next section we will learn a better way to accomplish this.

Summary

When we subclass a class, we get all of its members, both properties and methods. If the members are public or protected, we can access them within the subclass. If we wish to change or augment the behavior of a given method of the child class, we can ***override*** that method and replace it with our own functionality. Within the overridden method, we can call the superclass' method if we choose.

6.3) Polymorphism

Polymorphism in Object Oriented Programming is the provision of a single interface to entities of different types.

Motivation for Polymorphism

From the fruit example in the previous section, it would be preferable if we could just store an array of fruits and call getDescription on each fruit. It would be great if the correct getDescription got called based on the type of fruit that was created, not the type of the array.

It turns out that this WORKS! For apples it will call the apple version of getDescription, and for oranges it will call the orange version.

```
 class Fruit{
  constructor(private name: string){}
  public getDescription(): string{
    return `This is a fruit called ${this.name}`;
  }
}

class Orange extends Fruit{
  constructor(protected subType:string){
    super("orange");
  }
  public getDescription(): string{
    return super.getDescription() + " of type " +
this.subType;
  }
}
class Apple extends Fruit{
  constructor(protected subType:string){
    super("apple");
  }
  public getDescription(): string{
    return super.getDescription() + " of type " +
this.subType;
  }
}
let fruits:Fruit[] = [new Apple("red"), new Apple("green"),
  new Orange("blood"), new Orange("navel")];
for (let fruit of fruits){
  console.log(fruit.getDescription());
}
```

If either class did not implement getDescription(), then the superclass version would be called. This powerful behavior is a type of ***polymorphism*** and allows us to create very powerful class hierarchies, that are simple to access and use.

In other words, in our fruit example, we provided a public interface for all fruits that included the method getDescription(). Regardless of the type of fruit, the public interface does not change, and the language is able to ***dispatch*** the method call to the appropriate subclass for us automatically.
This type of ***polymorphism*** is ***subclass*** or ***subtype*** polymorphism. There are other types of polymorphism including ad-hoc polymorphism and parametric polymorphism. We will examine parametric polymorphism later.

So with creative use of subclass polymorphism, we can provide a generic interface to all objects that share a base class, with a default behavior.

Back to the *drawing* board

Returning to the drawing example, if we added a draw method to the drawable class that does nothing, then implemented the draw method in each of our subclasses, then we could store a drawing as an array of drawables, iterate through the array, and call the draw method. This is acceptable because Drawable has a draw method, but the correct draw method (depending on the type of object) will be called for us automatically. This is ***polymorphism***

```
 class Drawable {
  public color: Color;
  constructor(color: Color) {
    this.color = color.clone();
  }
  clone(): Drawable {
    return new Drawable(this.color);
  }
  draw(page:Page): void {
    //Do nothing, I don't know how
  }
}

class Line extends Drawable {
   ...
  draw(page: Page): void {
    page.drawLine(
      this.start.getX(),
      this.start.getY(),
      this.end.getX(),
      this.end.getY(),
      this.color.toString(),
    );
  }
}
let obj:Drawable=new Line(new Point(0,0),new Point(1,1),
  new Color());
obj.draw(this.drawingSurface);
```

{: .no-run}

> *Note: You can install the drawing library using in this example with the page object using npm.*
> `npm i --save @boots-edu/web-draw`

It is safe to call draw on a Drawable object, it just doesn't do anything.
If we call it on a Line object, it draws the line.
If we call it on a Line object stored in a Drawable variable (which is allowed since it is a Drawable), it calls the method in the Line class.

Summary

Polymprhism in general denotes the idea of several different types of objects having the same public interface. Specifically, in this section we examined ***subtype*** or ***subclass*** polymorphism which is when we ***override*** methods in a superclass allowing us to call the methods on a variable of the superclass type which contains an object of the subclass type. This causes the system to ***dispatch*** the call to the correct subclass.

6.4) Abstract Classes

An ***abstract*** class is a class that cannot be instantiated, but which can be used as a superclass for other classes.

Abstract Classes

With the version of our drawing program from the last section, what happens when a developer using our class creates an actual Drawable object. We built it to act as a superclass for all of the drawable objects, but it makes no sense to create one on its own. It isn't really drawable since the draw function doesn't do anything. It provides no functionality, and serves no purpose other than to act as a superclass to our other elements, hold their color, and dispatch their draw requests.

```
 let weird:Drawable=new Drawable(new Color());
weird.draw(this.drawingSurface);
```

{: .no-run}

It would be nice not to be able to prevent a user of our class from accidentally creating and using one of these.

Let's begin with our definition of a Drawable from the last section:

```
 class Drawable {
  public color: Color;
  constructor(color: Color) {
    this.color = color.clone();
  }
  clone(): Drawable {
    return new Drawable(this.color);
  }
  draw(page:Page): void {
  }
}
```

{: .no-run}

We can modify our drawable class to prevent it from being instantiated directly by tagging it as abstract in the method signature.
This breaks our clone method, how do we fix it.

```
 abstract class Drawable {
  public color: Color;
  constructor(color: Color) {
    this.color = color.clone();
  }
  clone(): Drawable {
    return new Drawable(this.color);  // Since it is
abstract, we are not allowed to create one anymore.
  }
  draw(page:Page): void {
  }
}
```

{: .no-run}

We simply remove the body of clone and mark it as abstract

```
 abstract class Drawable {
  public color: Color;
  constructor(color: Color) {
    this.color = color.clone();
  }
  abstract clone(): Drawable;  //Just the signature
followed by a semicolon is sufficent to create the
interface without an implementation.
  draw(page:Page): void {
}
```

{: .no-run}

Since we can't make one of these directly, we cannot clone it. We rely on the implementation in the super class.
If you derive from an abstract class, then all abstract members MUST be implemented in the subclass since now there is no default implementation.

We can take a this a step further and remove the do nothing method draw by making it an abstract method as well.

```
  abstract class Drawable {
   public color: Color;
   constructor(color: Color) {
     this.color = color.clone();
   }
   abstract clone(): Drawable;
   abstract draw(page:Page): void;
 }
```

{: .no-run}

Now any class that derives from Drawable will not compile if it does not implement clone and draw itself.
However, since they are still defined in the superclass, we can still call it on any object derived from Drawable and it will still dispatch to the correct subclass method. If we removed it altogether, it would not dispatch correctly when called.

Summary

A base class that wants to express a public interface for its subclasses, but does not provide an implementation for that interface is called an ***abstract class***. Any methods within the class that do not have implementations are called ***abstract methods***. We denote both a class or a method being abstract by using the `abstract` keyword.

6.5) Polymorphism Notes

Polymorphism in *Object Oriented Programming* is the provision of a single interface to entities of different types.

Things to know

It is ok to store an object of a subclassed type in a variable typed to the superclass.

```
 let dObj:Drawable=new Line(new Point(1,2),new
Point(3,4),new Color(1,2,3));
```

{: .no-run}

Calling methods on that variable will call the method in Line if it is implemented, and fall back to calling the method in Drawable if it is not.

```
 dObj.draw(this.drawingSurface);
```

{: .no-run}

If a class has no intended use on its own, but only is used as a parent class, then we can make it abstract, meaning that it cannot be created with new.

```
abstract class Drawable {
```

{: .no-run}

If we have methods that make no sense in the superclass, and must be implemented in the subclass, then we can declare them as abstract as well to support dispatch.

```
abstract draw(page: any): void;
```

{: .no-run}

An Example

Remember our Users/Student/Faculty classes.
Here is a simplified and updated version for us to look at.
The base class Users implements name, age, and two methods to access them.
It is abstract and cannot be created.
In addition, suppose we want to build a database of users, the Database class implements that.

```
 abstract class Users{
   constructor(protected name: string, protected age:
number){}
   getName():string{return this.name};
   getAge():number{return this.age};
```

```
  abstract getDetails():string;
}
class Students extends Users{
  constructor(name: string, age: number, private grade:
number){
    super(name, age);
  }
  getDetails():string{
    return `N: ${this.name}, A: ${this.age}, G:
${this.grade}`;
  }
}
class Faculty extends Users{
  constructor(name: string, age: number, private
department: string){
    super(name, age);
  }
  getDetails():string{
    return `N ${this.name}, A: ${this.age}, D:
${this.department}`;
  }
}
class Database{
  private users: Users[] = [];
  addUser(user: Users):void{
    this.users.push(user);
  }
  getUsers():Users[]{
    return this.users;
  }
  getUser(name:string):Users[]{
    let result:Users[] = [];
    for(let user of this.users){
      if(user.getName() === name){
        result.push(user);
      }
    }
    return result;
  }
}
let db:Database=new Database();
db.addUser(new Students("Lisa",19,4.0));
db.addUser(new Faculty("Linda",45,"Computer Science"));
let users=db.getUsers();
for(let user of users){
  console.log(user.getDetails());
}
```

Even though the database contains a mix of Students and Teachers, we return an array of Users to make the method more generic.
We can loop through the returned values getting details on each object regardless of type.

In general, you should return the most generic (i.e. superclass) type possible to make your method generic. There are ways to look and see what class we actually are, but if we are calling overridden methods that exist in the superclass, we don't need to worry about that. We just use it.

Summary

- You now know most of the generic things about OOP. In other words, while the syntax may differ slightly, all of the concepts hold true in most OO languages like Java, C++, C#, etc.

- We can construct complex classes by building them out of parts that they contain using composition.
- We can construct complex classes by extending other classes and adding functionality to create more and more specific classes that take advantage of the features that already exist in the superclass.
- We can use the idea of polymorphism to reference objects through their superclass, and have the correct implementation in the subclass execute for us through polymorphism.
- We can use the idea of ***polymorphism*** to reference objects through their superclass, and have the correct implementation in the subclass execute for us.
  We can prevent the creation of a class being used exclusively as a superclass by marking it as ***abstract***.
  We can force subclasses to create overridden methods for our superclass by declaring methods as ***abstract***. This does not prevent dispatch, but does remove the default behavior, making all subclasses implement the method themselves.

And with all of this, we have an elegant way to design programs that leverages the ability to share code, and view a problem in terms of objects.

7) Exceptions and Code Quality

7.1) Exceptions

An ***Exception*** is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions at run time.

Exceptions in Typescript

What is an exception?

- An exception is a way to break the “normal” flow of a program in the event that an abnormal condition exists.
- This can be due to invalid inputs or data provided at runtime or any other condition that is not the “common case” behavior of a method or function.
- It is a way to respond to validation within your code in a structured way.
- Some exception may be generated by libraries that you may use.
- You can raise and throw exceptions within your own code
- When an exception is thrown, the program will terminate unless the exception is caught.

```
 let x:number=50;
throw new Error("This is an error");
console.log(x);
```

Note that the line console.log(x) will not execute. The current function will exit immediately and if the exception is not “handled” by a calling method somewhere in the call stack, the program will terminate immediately.
We will talk about handling exceptions in a bit, but for now, we want to be able to generate them when ***exceptional conditions*** occur.

So let’s examine in detail what the above code does:

- Sets the variable x to the value 50.
- Immediately terminates execution of the current method and begins to “bubble up” the exception through all of the calling methods until it is handled.
- If the exception bubbles past the first function called, the program terminates and prints an error message to the console.

If the exceptions is not handled, the program exits. The system prints out the call stack in the console.

```
Error: This is an error
    at test (/home/xxx/test.js:4:11)
    at Object.<anonymous> (/home/xxx/test.js:7:1)
    at Module._compile
(node:internal/modules/cjs/loader:1376:14)
    at Module._extensions..js
(node:internal/modules/cjs/loader:1435:10)
    at Module.load
(node:internal/modules/cjs/loader:1207:32)
    at Module._load
(node:internal/modules/cjs/loader:1023:12)
    at Function.executeUserEntryPoint [as runMain]
(node:internal/modules/run_main:135:12)
    at node:internal/main/run_main_module:28:49
```

> *Note: The call stack shows us all the places where we could have caught the error as well as all the internal code that is part of the Typescript system. In this example, the first 2 lines show where we could have caught the exception.*

Using exceptions

We can use exceptions to improve our software design and make it react in a structured way to ***exceptional conditions***.

Let’s consider the code for our drawing program again.

```
class Color {
  constructor(
    private red: number = 0,
    private green: number = 0,
    private blue: number = 0,
  ) {}
  clone(): Color {
    return new Color(this.red, this.green, this.blue);
  }
  getRed(): number {
    return this.red;
  }
  getGreen(): number {
    return this.green;
  }
  getBlue(): number {
    return this.blue;
  }
}
```

Valid color values in our program are numbers between 0 and 255. What happens if we try to create a color with different values?

- The code will allow these non-sensical values to be stored in red, green and blue.

We can use exceptions to prevent this.

```
 export class Color {
  constructor(
    private red: number = 0,
    private green: number = 0,
    private blue: number = 0,
  ) {
    if (red<0 || red>255) throw new Error(“Invalid red
value”);
    if (green<0 || green>255) throw new Error(“Invalid
green value”);
    if (blue<0 || blue>255) throw new Error(“Invalid blue
value”);
  }
  clone(): Color {
    return new Color(this.red, this.green, this.blue);
  }
  //Rest of code removed for brevity
  . . .
  . . .
}
```

We can check the values in the constructor, and throw an exception if they are invalid. It will be up to the code that is creating the color object to “handle” the exception, otherwise the program will exit with an error like the one we saw previously.

> *Note: Now we can't create a color objects with invalid values. If we try, the Color class will raise an exception to notify the calling code that something bad happened.*

If the calling code does not "handle" the exception, then the program will terminate with an error message (the one you threw) and the call stack to help you figure out where the exception occurred in the execution of your program.

```
const color:Color=new Color(400, 400, 400);
```

Throws (raises) an exception with the message "Invalid red value". Again, if this is not handled somewhere in the code that calls this, the program will exit.

Custom Errors

If we want to pass more information with our Error, we can create our own class that extends error, and throw that.

```
  class ColorError extends Error {
    constructor(
      message: string,
      public red: number,
      public green: number,
      public blue: number,
    ) {
      super(message);
      this.name = "ColorError";
    }
 }
```

Here `this.name` is part of the Error class which we are extending (inheritance). The message is as well which we are updating by calling `super(message);` then we are adding properties red, green, and blue so that they are reported to the calling method with the exception. This can be very useful when we get to exception handling.
If a block of code throws different kinds of exceptions, this can be a good way to notify the calling method as to the type of exception and can help in writing the handler.

Exceptions are useful during programming even if we don't handle them.
If you throw an exception every time the inputs to your method are wrong, or some other kind of error occurs, and you have good tests, you will see those errors and be able to fix them.

If we accidentally try to create an invalid color object, the program will terminate and tell us why. The call stack will tell us where the method was called.

There are other places in our drawing code where we are allowing an invalid or incorrect state to occur because we are not checking. Again, we can prevent this by throwing an exception when this happens.

In our polygon class, I can create polygonswith no points, 1 point, or 2 points which are NOT POLYGONS.We can also create millions of polygons, perhaps we can prevent that as well.
Good documentation can help, but using exceptions will prevent it.
Can I use exception handling to make sure it is not possible to create an invalid polygon?

```
 const MAX_POINTS:number = 10;
class Polygon extends Drawable {
  protected points: Point[] = [];
  constructor(points: Point[], color: Color) {
    super(color);
    if (points.length<3 || points.length>MAX_POINTS)
      throw new Error("Invalid polygon");
    for (let point of points) {
      this.points.push(point.clone());
    }
  }
  //Rest of class removed for brevity
  . . .
  . . .
}
```

Now, if I try to create a polygon with less than 3 or more than 10 points, an exception is thrown. If not, then program execution continues normally.
If we don't handle this exception, the program will terminate (letting us know to either handle the exception, or fix the calling code to prevent it.

Where else might exception handling help us find issues with our drawing program?

How about a circle with 0 or negative radius?

```
 class Circle extends Drawable {
    private center: Point;
    private radius: number;
    constructor(center: Point, radius: number, color:
Color) {
      super(color);
      if (radius<=0) throw new Error("Radius must be
greater than 0");
      this.center = center.clone();
      this.radius = radius;
    }
    //Rest of class removed for brevity
    . . .
    . . .
}
```

A line where the two points are the same

First it might be useful to add a method to compare to points. We can then use that method to determine if two points have the same value (not the same object reference).

```
 class Point {
  constructor(
    private x: number = 0,
    private y: number = 0,
  ) {
    super(color);
    if (start.equals(end))
      throw new Error("Start and end points must be
different");
    this.start = start.clone();
    this.end = end.clone();
  }
  equals(other: Point): boolean {
    return this.x === other.x && this.y === other.y;
  }
  //Rest of class removed for brevity
  . . .
  . . .
}
```

Remember if a and b are Point objects, then a===b asks if they are the same object reference in memory, but a.equals(b) checks if they have the same coordinates, whether or not they are the same physical object reference.

We can use the `equals` to validate our line object. If the two points have the same coordinates, regardless of if they are references to the same object, the constructor will throw an exception.
Now our Line is guaranteed to have start and end points with different coordinates.

> *Because `Color` throws an exception if the values are invalid, we don't need to check that here. The call to the color constructor will throw an exception if the color is invalid, so we don't need to worry about it here.*

We could do something similar with our polygon class to verify that none of the points are the same. This would also handle things for our `Rectangle` and `Triangle` classes since they are now ***derived*** from `Polygon`

```
 class Polygon extends Drawable {
    protected points: Point[] = [];
    constructor(points: Point[], color: Color) {
        super(color);
        if (points.length < 3 || points.length >
MAX_POINTS)
            throw new Error(
                `A polygon must have at least 3 points and
at most ${MAX_POINTS} points`,
            );
        // Check for duplicate points
        for (let i = 0; i < points.length; i++) {
            for (let j = i + 1; j < points.length; j++) {
                if (points[i].equals(points[j])) {
                    throw new Error("Duplicate points are
not allowed in a polygon.");
                }
            }
        }

        for (let point of points) {
            this.points.push(point.clone());
        }
    }
  //Rest of class removed for brevity
  . . .
  . . .
}
```

> *Note the **Brute force** approach to searching for duplicates. For each element, check all the remaining elements for duplicates. Also note that we still need to make sure there are at least 3 and not more than MAX_POINTS points in the polygon. Now we are also making them unique.*

> *Thought Question: Why does j start at i+1 and not 0?*

Defensive Programming

So now we can prevent our code from being exposed to “exceptional” or invalid operation, by simply throwing an exception when those cases arise.
If we write good test cases, we will find errors in our code, but right now, our program will just exit with an error message.
Making sure that our code will not accept invalid values and thus have undocumented, or undefined behaviors is good ***defensive programming***.
It would be better if we were able to catch the exception somewhere in the call stack and handle it elgently instead of just having our program crash with an error message just

because of some invalid input. At a minimum it would be nice to exit cleanly and report the problem to the user in a more "user friendly" way.

Exception Handling

We can use the try/catch/finally approach to handle errors thrown by methods that we call.

```
 try {
   //do something which might throw an exception
} catch (e) {
   //handle the exception in some way
}finally{
   //do something after regardless of the try/catch result
}
```

If we do one or more operations which might throw an error within a try block, if an exception occurs within that code or any code that is called within the block, that code exits immediately, and the catch block is called, where e is the Error derived object that was passed to throw within the code.
This will prevent the program from exiting and consume the exception and the program will continue normally after the try/catch/finally block. You can rethrow the error in the

catch block, which will continue to “bubble up” the exception so our caller can handle the error after we recognize it (maybe we log, then rethrow).

```
 let color: Color;
let line: Line;
const start = new Point(100, 100);
const end = new Point(200, 200);
try {
    color = new Color(0, green, 0);
} catch (e) {
    console.log(e);
    color = new Color();
} finally {
    line = new Line(start, end, color);
}
```

Here we try to create a color. If the color is valid, it is created, if not, the error is logged to the console, and a default color object is created.
The finally block runs after either way. It creates a line with the newly defined color. We have handled the exception and our code will work, even if the value of green is invalid. It will either create a green line if green>=0 && green<=256 or the default colored line if not.

```
 let color: Color;
let line: Line;
const start = new Point(100, 100);
const end = new Point(200, 200);
try {
    color = new Color(0, green, 0);
} catch (e) {
    console.log(e);
  color = new Color();
}
line = new Line(start, end, color);
```

A note about finally. In this code it is not necessary since the code continues after the try/catch either way, so we can remove it and just let the program continue with creating the line. There are many use cases where we don't need a finally block, but there are some where we do.

Here is a case where ***finally*** is useful:

```
 import * as fs from "fs";
let fileDescriptor: number;
let fileContents: string;
try {
    fileDescriptor = fs.openSync("test.txt", "r");
} catch (e) {
    throw new Error("Could not open file test.txt");
}
try {
    fileContents = fs.readFileSync(fileDescriptor, "utf8");
} catch (e) {
    throw new Error("Could not read file test.txt");
} finally {
    console.log("Closing file");
    fs.closeSync(fileDescriptor);
}
console.log(fileContents);
```

If I have an open file, and encounter an error while reading it, we want to rethrow the exception, but first we want to close the file.
This code opens the file, tries to read it, and regardless of success or not, closes the file.
On success it prints the contents, and on error it throws an exception

Common Pitfalls and Mistakes

- Throwing a string instead of an Error: Allowed but bad form

- Using exceptions to communicate non-exceptional situations. These are designed for expressing error conditions, and should not be used as a way to return data in normal execution.
- If we want the exception to continue to bubble, we must rethrow it, or throw a new exception of our own. `throw e` or `throw new Error(“This is my error”)`

Summary

In summary, when writing our code we should program defensively.
When a method or code block accepts input, throw an exception if the input is not valid.
We can override (extend) the Error class to create our own more detailed Error classes for our exceptions.
The thrown exception will “bubble up” through the code that called the code that threw the exception, all the way to the top of the call stack. If nothing handles it, then the program terminates and displays the exception and the full call stack.
We can catch a thrown exception with the try/catch or try/catch/finally constructs. These consume the exception (stop bubbling).

7.2) Comments

Producing well documented, high quality, efficient and readable code is always the goal in software development.

Code Quality in General

Why comments?

Helps others (and yourself) use your code without having to read it. Informs user of everything they need to know to use your method or class.

If in the correct format, they can automatically produce documentation.

If in the correct format, they can be read by IDE's like vscode.

Why naming matters?

If we do need to revisit our code (and we will), having well named variables and methods makes figuring out what the code is doing internally much easier.

Our classes will be easier to use if our public interface uses names that make sense given the purpose of the thing we are referencing.

Code Quality

This is a general measure of how good the code is. It includes:

- Efficiency (more on this next semester)
- Readability
    - Comments, naming, indenting, consistency of the code, adherence to standards, etc.
- Usability
    - How easy to use is the code. If it is a class, how easy is it to create objects or extend. How easy is it to make changes. If a program, how what is the user experience like?

Comments

At this point, you should be convinced you that comments are worth your time. Now lets look at how to format a comment in typescript to make it more usable.
We are using the jsdoc format for our comments. This is a good solution because we can automatically generate our documentation of our classes and methods, as well as provide tool tip help in vscode (and other IDEs).

The most common tags available to us for jsdoc are:

@param	@private	@example	@override
@returns	@protected	@memberof	@implements

@description	@throws	@property	@interface
@class	@export	@function	

Some of these are for constructs we have not learned yet, but all but 2 can be understood now.

```
 /**
  * A class that represents a polygon.
  * @class Polygon
  * @extends Drawable
  * @description A class that represents a polygon.
  * @method clone A method that returns a new polygon object
that is a clone of the current polygon object.
  * @method draw A method that draws the polygon on the
drawing surface.
  * @throws An error if the number of points is less than 3
or greater than 10.
  * @throws An error if there are duplicate points.
  */
class Polygon extends Drawable {
   . . .
```

Here is a well formatted comment for the polygon class.
Note it tells us everything we need to know about the class to use it.
It also describes the exceptions that it may throw.

We should also comment the methods inside our class. This is what a comment for the constructor might look like:

```
 /**
     * Create a new polygon object.
     * @param {Point[]} points Array of vertices of the
polygon.
     * @param {Color} color The color of the polygon.
     * @throws An error if the number of points is invalid
     * @throws An error if there are duplicate points.
     * @sideEffects Allocates a new polygon object.
     * @memberof Polygon
     * @constructor
     * @example
     * let p1 = new Point(0, 0);
     * let p2 = new Point(0, 1);
     * let p3 = new Point(1, 1);
     * let polygon = new Polygon([p1, p2, p3], new
Color());
*/
    constructor(points: Point[], color: Color) {
```

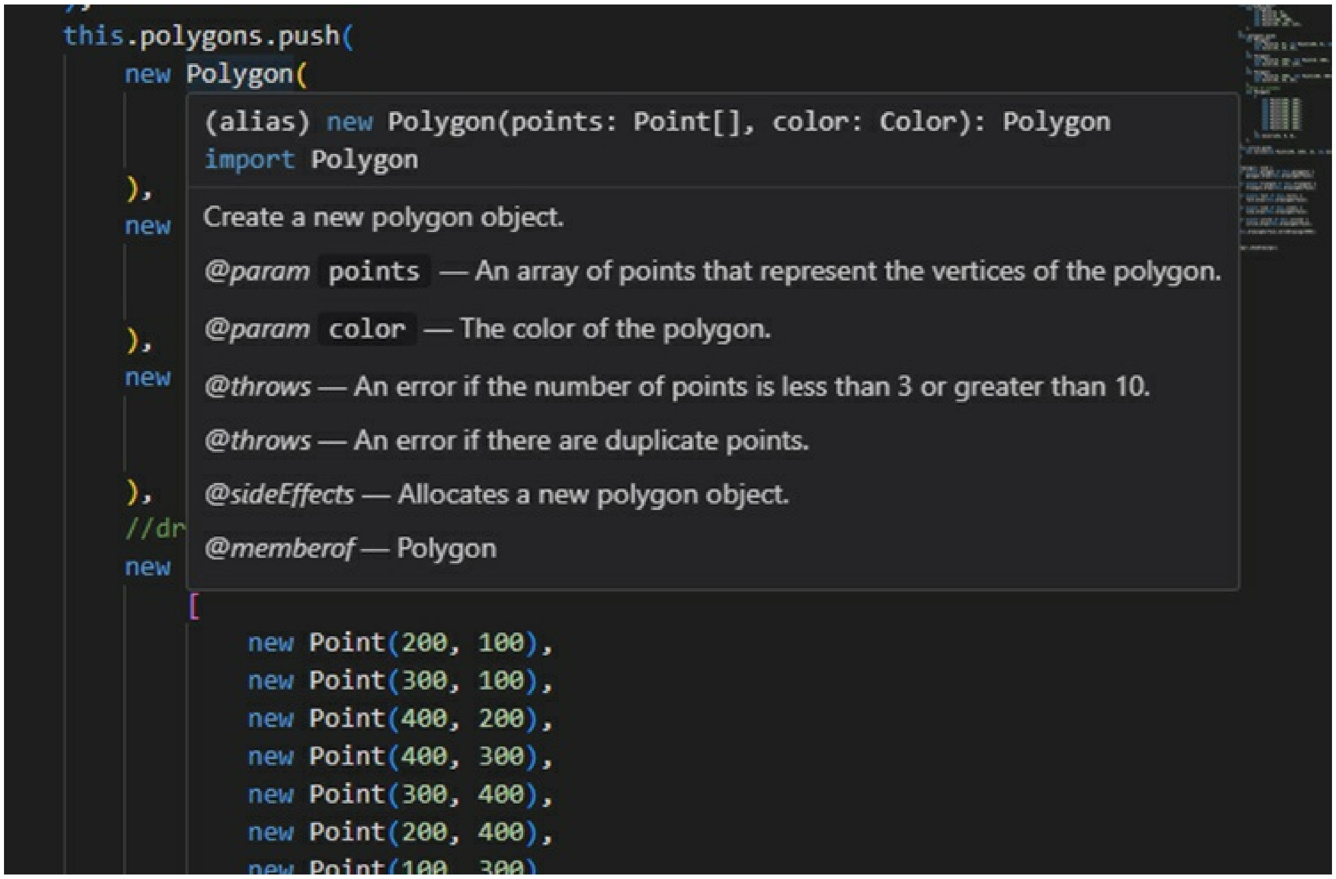
- We see the parameters and their types and description.
- What exceptions to expect
- It’s side effects
- It’s parent class
- It is a constructor
- An example of how to use it.

The clone method as well:

```
 /**
     * Return a deep copy of our polygon object in a new
one.
     * @description Clones a polygon object
     * @param none
     * @returns A new polygon object that is a clone of the
current polygon object.
     * @override The clone method of the Drawable class.
     * @memberof Polygon
     * @function clone
     * @sideEffects Allocates a new polygon object.
     * @example
     * let p1 = new Point(0, 0);
     * let p2 = new Point(0, 1);
     * let p3 = new Point(1, 1);
     * let polygon:Polygon = new Polygon([p1, p2, p3], new
Color());
     * let polygon2:Polygon = polygon.clone();
     */
    clone(): Polygon {
```

- We see the parameters and their types and description.
- The return values
- It’s side effects
- It’s parent class
- It is a function
- An example of how to use it.

Why bother with all this formatting?



```
this.polygons.push(
    new Polygon(
```

```
(alias) new Polygon(points: Point[], color: Color): Polygon
import Polygon
```

Create a new polygon object.

@param `points` — An array of points that represent the vertices of the polygon.

@param `color` — The color of the polygon.

@throws — An error if the number of points is less than 3 or greater than 10.

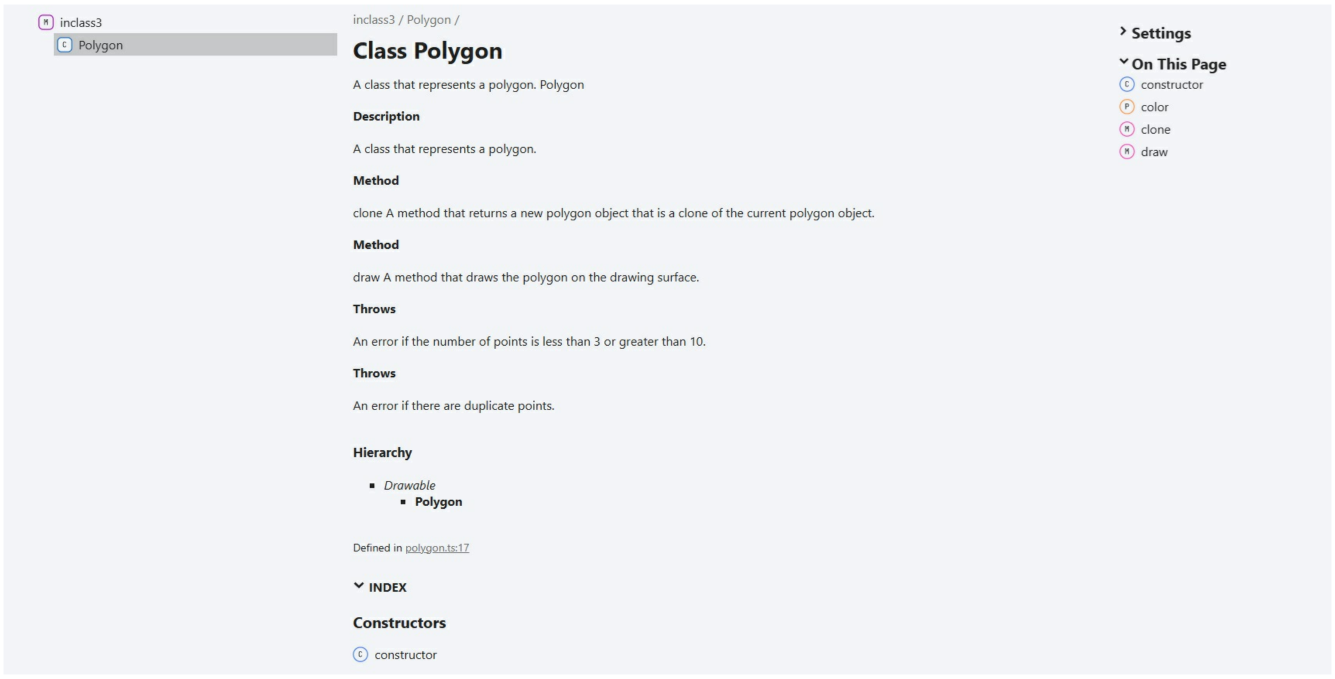
@throws — An error if there are duplicate points.

@sideEffects — Allocates a new polygon object.

@memberof — Polygon

```
    ),
    new
    ),
    new
    ),
    //dr
    new
        [
            new Point(200, 100),
            new Point(300, 100),
            new Point(400, 200),
            new Point(400, 300),
            new Point(300, 400),
            new Point(200, 400),
            new Point(100, 300)
```

Look what happens when I hover over the polygon class in vscode now. I now get help on using this class constructor.
We can also generated detailed technical documentation automatically by using the typedoc command.



inclass3
Polygon

inclass3 / Polygon /

Class Polygon

A class that represents a polygon. Polygon

Description

A class that represents a polygon.

Method

clone A method that returns a new polygon object that is a clone of the current polygon object.

Method

draw A method that draws the polygon on the drawing surface.

Throws

An error if the number of points is less than 3 or greater than 10.

Throws

An error if there are duplicate points.

Hierarchy

- *Drawable*
    - **Polygon**

Defined in polygon.ts:17

INDEX

Constructors

constructor

Settings

On This Page

constructor
color
clone
draw

Quality, well formatted comments make your code more usable, manageable, and maintainable.

A journalist asked a programmer:- What makes code bad?
No comment.

6:16 PM · 20 Jul 18

16 Retweets 71 Likes

There are other things we can do to improve code quality as well.

Summary

Clear, straight forward comments on our code make our code more useful. We can specify details about how to use the code, what its limitations are, if it throws exceptions, and what it expects and returns. If formatted using jsdoc, then we can also get help in IDEs like Visual Studio Code and generate a detailed documentation website using typedoc.

7.3) Naming

Naming elements in a way that we can tell what type of thing/data the element is/contains makes code more readable.

What’s in a name?

Consider the following class:

```
 class A {
  constructor(public X: string,public Y:number) {}
}
```

- What do objects of this class represent?
- Can we tell what it is and when to use it?
- Do we know what the parameters represent?
- What is its purpose, why does it exist?

Rewritten with meaningful names:

```
 class Person {
  constructor(public name: string,public age:number) {}
}
```

- Now it is clear what this class represents.

- It is clear what the meaning of the parameters are
- It is clear why this class exists and when we would use it.
- It’s not that hard to do it right.

A more complex example

```
 class Jane {
    constructor(
        public lisa: number,
        public bill: number,
    ) {}
    f(): string {
        return `${this.lisa}e${this.bill}`;
    }
    g(other: Jane): Jane {
        if (this.bill === other.bill) {
            return new Jane(this.lisa + other.lisa,
this.bill);
        } else {
            const expDiff = Math.abs(this.bill -
other.bill);
            if (this.bill > other.bill) {
              return new Jane(
                this.lisa + other.lisa * Math.pow(10,
expDiff),
                this.bill,
              );
            } else {
              return new Jane(
                this.lisa * Math.pow(10, expDiff) +
other.lisa,
                    this.bill,
              );
            }
        }
    }
}
```

- What does Jane represent
- What does f do?
- What does g do?
- Why did someone write this?

While a somewhat extreme example, bad naming is quite common, and makes no sense to do.

A much better code block with proper naming makes things clear:

```
 class RealNumber {
    constructor(
        public integer: number,
        public exponent: number,
    ) {}
    toString(): string {
        return `${this.integer}e${this.exponent}`;
    }
    add(other: RealNumber): RealNumber {
        if (this.exponent === other.exponent) {
            return new RealNumber(this.integer +
other.integer, this.exponent);
        } else {
            const expDiff = Math.abs(this.exponent -
other.exponent);
            if (this.exponent > other.exponent) {
                return new RealNumber(
                    this.integer + other.integer *
Math.pow(10, expDiff),
                    this.exponent,
                );
            } else {
                return new RealNumber(
                    this.integer * Math.pow(10, expDiff) +
other.integer,
                    this.exponent,
                );
            }
        }
    }
}
```

- It’s clear what the class represents

- It’s clear what toString does
- It’s clear what add does
- It’s clear what this is for.

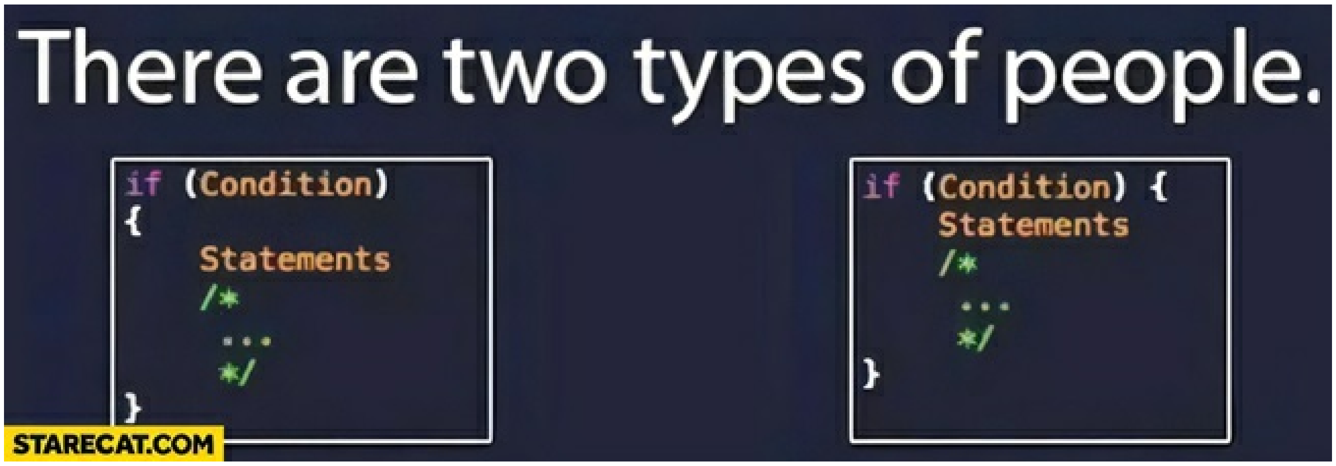
Summary

When writing code, choosing good names that represent the objects or purpose of classes, variables, or functions makes it possible to figure out what the code does and makes it easier to maintain and use.

7.4) General Code Quality

Programmers should always try to create efficient, readable, and maintainable code. It’s not that hard to do it right.

Best Practices



```
if (Condition)
{
    Statements
    /*
     ...
     */
}
```

```
if (Condition) {
    Statements
    /*
     ...
     */
}
```

Good formatting, indenting, and consistency of style are important to maintaining a large code base. Many organizations will dictate these types of things.

- Indents are 2 or 4 spaces
- Braces at the end of lines or on a new line
- Parameters on one line or multiple lines
- The list goes on. There are best practices, but while many are agreed upon, some are preferences.

> *Rule number 1: be consistent.*

Summary

When coding, remember you are not the only one who will look at your code. Others will be responsible for maintaining, updating, or using the code you produce. Writing well comments, well named, clear and consistent code is critical to success as a software developer.

Chapter Summary

In order to write better programs, we must handle exceptional cases. The exception handling process (try/catch/finally) gives us a mechanism to easily handle these cases.

Code should be readable, maintainable, and understandable. Good comments are critical. Naming of functions, classes, and variables so that their names represent the information they will hold or action they will take is also important.

8) Software Testing

8.1) Testing

What we are concerned with in software testing:

- Validate the software is bug free
- Validate the software meets requirements
- Validate the software behaves as expected on boundary cases
- Validate the software behaves as expected on exceptional cases

Verification and Validation

- ***Verification*** refers to the set of tasks that ensure that the software correctly implements a specific function. It means “Are we building the product correctly?”.
- ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. It means “Are we building the correct product?”.

Motivation

A little history to motivate the discussion.

Software bugs can be expensive, but they can also be very dangerous. Here are a few examples of software bugs causing terrible outcomes:

- 1985: Canada’s Therac-25 radiation therapy malfunctioned due to a software bug and resulted in lethal radiation doses to patients.
- 1994: China Airlines Airbus A300 crashed due to a software bug killing 264 people.
- 1999: A software bug caused the failure of a $1.2 billion military satellite launch.
- 2015: A software bug in an F-35 resulted in it being unable to detect targets correctly.
- Starbucks was forced to close more than 60% of its outlet in the U.S. and Canada due to a software failure in its POS system.
- Nissan cars were forced to recall 1 million cars from the market due to a software failure in the car’s airbag sensory detectors.

We have to get it right!!!

Types of testing

- ***Functional***: Does it do what it is supposed to do? Does it meet requirements? Does it work correctly on all possible inputs?

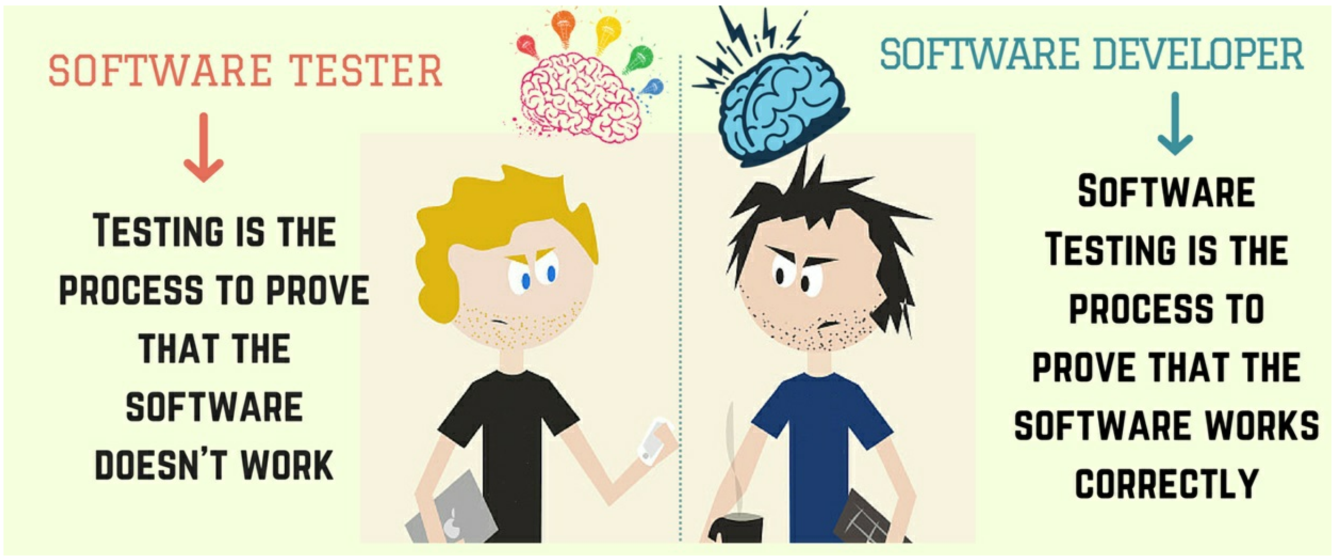
- ***Non-Functional***: How does it perform on various inputs? Does it scale? How usable is it? How does it behave under heavy use/load?
- ***Regression Testing***: After the software is modified, verify that the modifications did not damage previously working components of the system.

Testing levels

- ***Unit testing***: Test small independent components for correct behavior. The purpose is to validate that each unit of the software performs as designed.
- ***Integration testing***: Combining units and testing as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
- ***System testing***: Tests of the completed system. The purpose of this test is to evaluate the system's compliance with the specified requirements.
- ***Acceptance testing***: Test to ensure compliance with the requirements specification. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Best Practices

- Test continuously throughout the development process.
- Make tests small and include many to make finding issues easier
- Use tools to evaluate things like code coverage to ensure thorough testing
- Don’t skip regression testing.


SOFTWARE TESTER
TESTING IS THE PROCESS TO PROVE THAT THE SOFTWARE DOESN’T WORK
SOFTWARE DEVELOPER
SOFTWARE TESTING IS THE PROCESS TO PROVE THAT THE SOFTWARE WORKS CORRECTLY

Some comments from experience:

- While time consuming, testing is critical to writing good software systems.
- Poorly written tests are like having no tests at all. This requires some thought.
- Failure to write tests will eventually cause problems in any system of a reasonable size.

- Seemingly unrelated code segments can and do break each other.
- Test after each change to aid in solving issues that arise as you code.

> *In other words, you should write tests early in the process. Possibly even before writing a line of code.*

Good Tests

So what makes a good test?
Start at the unit level (function) and validate that the function behaves as expected in all cases.

- Make sure you test its behavior on edge cases
- Make sure you test its behavior on exceptional/invalid inputs
- Make sure your comments document the behavior in exceptional/invalid instances. (ie. Does it replace the value, throw an exception)
  Once you have unit tests, start testing higher level operations (i.e. instantiate classes that use your unit tested code. Simulate the overall behavior of the system. Again, use the same methodology.

Understanding what to test.

We will be using jest to write tests in Typescript. You have already seen this in lab, but now we are going to write our own tests.
Jest provides a format for writing tests in a simple and organized way.
Jest can run tests on the entire system or on individual components.
Jest can produce a coverage report to let you know which lines are not “covered” by the test (i.e. functions not called, branches not taken, etc.)
Testing can easily be built into the build cycle, so that tests are run as part of each build.

Here is some simple code that adds the root of the passed value to an array.

```
 const valueArray:number[]=[];
/**
 * @description Takes a number and adds its square root to
the array valueArray
 * @param value {number} - The number to be squared
 * @returns {number} - The square root of the number
 * @sideEffects - Adds the square root of the number to the
valueArray
 */
function addToRootArray(value:number):number{
    const root=Math.sqrt(value);
    valueArray.push(root);
    return root;
}
```

Let’s start by asking what we might want to know about how this code behaves, and how we could test that:

- How does it behave on a positive integer?
    - Pass it one and verify contents of the array
- How does it behave on a positive real number?
    - Pass it a positive real number and verify contents of the array
- How does it behave when passed a 0.
    - Pass it a 0 and verify the contents of the array
- How does it behave on a negative integer.

    - Pass it a negative integer and verify the contents of the array
- How does it behave on a negative real number?
    - Pass it a negative real and verify the contents of the array
- How does it behave when the array is empty/populated already?
    - Create various arrays with 0, 1, 2, and many elements, call the function and check the contents of the array.

> *Are those behaviors what we expect and what is documented?*

And here is another example.

```
 export class Elements {
    private stringArray: string[] = [];
    /**
     * @description This function returns and     * removes
the last element
     * @returns {string} - The last element of the
     * array
     * @sideEffects - Removes the last element of
     * the array
     */
    getLastElement(): string {
        return this.stringArray.pop();
    }
}
```

What questions might we ask here?

- Can I construct one of these?
    - Call the constructor and verify
- Does it work normally?
    - Populate with some items and try
- What happens if the array is empty?
    - Ensure array is empty and try
- What happens if the array has only one element in it? *Populate with 1 item and try

To create tests in a project that is already configured for jest, we create files with the word ‘test’ in their filename (i.e. myprogram.test.ts)

> *This can be changed, but our projects will be pre-configured to work this way.*

- Running jest on the command line by itself within the project folder will run tests in all properly named files.
- Running jest on the command line with the name of the file (without the test.ts) will run tests in only that file.
- Running jest on the command line with –coverage will produce a coverage report.

Summary

Designing good tests and testing methodologies will help create software that can be validated and verified. Different levels of testing allow for testing individual functions, classes, or sets of code as well as the full system. Before writing tests, ask what types of thing should be tested. Make sure you test edge cases and exceptional situations to make sure you have covered all possible inputs.

8.2) Testing in Jest

Jest is a test runner and testing framework that works with javascript and Typescript

Jest Syntax

A few simple commands we need:

- describe: Create a new test section
- test: Write a specific test
- expect: expect an expression to behave a certain way Example: expect(value).toBeInstanceOf(MyClass)

> *There are many others, but we can get by with these three for now.*

Back to the example from the previous section, let’s look at what we want to do for each of these:

```
class Elements {
    private stringArray: string[] = [];
    /**
     * @description This function returns and
     * removes the last element
     * @returns {string} - The last element of the
     * array
     * @sideEffects - Removes the last element of
     * the array
     */
    getLastElement(): string {
        return this.stringArray.pop();
    }
}
```

We will start with a describe block for the Elements class:

```
describe("Elements", () => {
    //Our tests go here
});
```

Can I construct one of these?

```
describe("Elements", () => {
    test("Create Instance", () => {
        const elements = new Elements();
        expect(elements).toBeInstanceOf(Elements);
    });
});
```

Does it work normally? Just create a test block that populates some items and then verify them.

```
 describe("Elements", () => {
     test("Array populated 2 or more", () => {
         const elements = new Elements();
         elements.stringArray=["a","b","c"];
         expect(elements.stringArray.length).toBe(3);
         const value = elements.getLastElement();
         expect(value).toBe("c");
         expect(elements.stringArray.length).toBe(2);
         expect(elements.stringArray).toContain("a");
         expect(elements.stringArray).toContain("b");
         expect(elements.stringArray).not.toContain("c");
     });
});
```

Test what happens when the array is empty.

```
 describe("Elements", () => {
   test("Array is empty", () => {
       const elements = new Elements();
       elements.stringArray = [];
       expect(elements.stringArray.length).toBe(0);
       expect(elements.getLastElement()).toThrowError("Array
is empty");
     });
});
```

> *This test fails because we don’t throw an exception and neither does pop which we are using to implement this. We need to fix the code.*

```
 class Elements {
    private stringArray: string[] = [];
    /**
     * @description This function returns and
     * removes the last element
     * @returns {string} - The last element of the
     * array
     * @sideEffects - Removes the last element of
     * the array
     * @throws {Error} - If the array is empty
     */
    getLastElement(): string {
        if (this.stringArray.length === 0) throw new
Error("Array is empty");
        return this.stringArray.pop();
    }
}
```

> *Note that now getLastElement throws an exception if the array is empty, so our test will now pass.*

What happens if the array has only one element in it?

```
     test("Array populated 1 item", () => {
        const elements = new Elements();
        elements.stringArray=["a"];
        expect(elements.stringArray.length).toBe(1);
        const value = elements.getLastElement();
        expect(value).toBe(“a");
        expect(elements.stringArray.length).toBe(0);
        expect(elements.stringArray).not.toContain(“a");
    });
```

So what does our final test suite for this code look like?

```
 // Currently does not work due to incomplete Jest
implementation.
class Elements {
    public stringArray: string[] = [];
    /**
     * @description This function returns and
     * removes the last element
     * @returns {string} - The last element of the
     * array
     * @sideEffects - Removes the last element of
     * the array
     * @throws {Error} - If the array is empty
     */
    getLastElement(): string {
        if (this.stringArray.length === 0) throw("Array is
empty");
        return this.stringArray.pop();
    }
}
```

```
describe("Elements", () => {
    test("Create Instance", () => {
        const elements = new Elements();
        expect(elements).toBeInstanceOf(Elements);
    });
    test("Array populated 2 or more", () => {
        const elements = new Elements();
        elements.stringArray = ["a", "b", "c"];
        expect(elements.stringArray.length).toBe(3);
        const value = elements.getLastElement();
        expect(value).toBe("c");
        expect(elements.stringArray.length).toBe(2);
        expect(elements.stringArray).toContain("a");
        expect(elements.stringArray).toContain("b");
        expect(elements.stringArray).not.toContain("c");
    });
    test("Array is empty", () => {
        const elements = new Elements();
        elements.stringArray = [];
        expect(elements.stringArray.length).toBe(0);

expect(()=>elements.getLastElement()).toThrowError("Array
is empty");
    });
    test("Array populated 1 item", () => {
        const elements = new Elements();
        elements.stringArray = ["a"];
        expect(elements.stringArray.length).toBe(1);
        const value = elements.getLastElement();
        expect(value).toBe("a");
        expect(elements.stringArray.length).toBe(0);
        expect(elements.stringArray).not.toContain("a");
    });
});
```

If we run our test using the ***jest*** command line, we get

```
 PASS  src/app/mathpain.test.ts

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.151 s
```

Here I see that all 4 tests passed.
Now I know that the class method works as expected, in all of the cases that I could think of.
I have also verified that it behaves as documented on exceptions.
Once written this test will run every time I run tests, handling regression testing of this particular method of this class when future updates are made elsewhere in the program.

Code Coverage

Coverage is important when writing tests
While you should not specifically write tests to coverage, since those tests will not cover all possible inputs, you should make sure that your tests actually cover your code.
Let’s look at our example again, from a coverage standpoint
Running: jest –coverage will produce a shortened coverage report like this:

```
 --------------------|---------|----------|---------|-----
----|-------------------
File                 | % Stmts | % Branch | % Funcs | %
Lines | Uncovered Line #s
--------------------|---------|----------|---------|------
---|-------------------
All files            |   97.61 |    92.66 |   94.84 |
97.78 |
 EzComponent.ts      |     100 |      100 |     100 |
100 |
 EzDialog.ts         |    93.1 |       75 |   83.33 |
94.44 | 135-140
 bind.decorators.ts  |   97.38 |    93.65 |   91.66 |
97.27 | 601-602,621-622
 bootstrap.ts        |     100 |      100 |     100 |
100 |
 event.decorators.ts |     100 |      100 |     100 |
100 |
 eventsubject.ts     |     100 |      100 |     100 |
100 |
--------------------|---------|----------|---------|------
---|-------------------
```

> *If we add `--coverageDirectory=‘./coverage’` to our jest command with –coverage, we still get the same information, but we also get a website with detailed info including source links.*

Summary

Jest is a powerful platform for designing suites of tests that cover all types and levels of code testing. Tests will run automatically when jest is run providing regression testing accross the application.

8.3) Anonymous Functions

A function that is declared with no name is an ***anonymous function***.

Normal functions

Normally, when we create a function or method, we define it with a name that we can use to reference it (call it) later.

```
 function MyName(a:number,b:number):number{
     return a+b;
}
class MyClass{
    MyName(a:number,b:number):number{
        return a+b;
    }
}
```

We can then call or reference that method by its defined name

```
 let a=MyName(1,2);
let b=new MyClass().MyName(1,2);
```

This is normal and a reasonable way to access methods and function in any programming language.
Sometimes, however, we just need a function right where we want to use it, and it is easier to be able to provide the function, rather than declare it elsewhere.

Anonymous functions

We have already seen this in our jest tests in both the describe method and the test method.
Let’s take a closer look at the second parameter to the describe and test methods.

```
 describe('Test Name',()=>{
    test('Test MyName',()=>{
        let a=MyName(1,2);
        expect(a).toBe(3);
    });
    test('Test MyClass',()=>{
        let b=new MyClass().MyName(1,2);
        expect(b).toBe(3);
    });
});
```

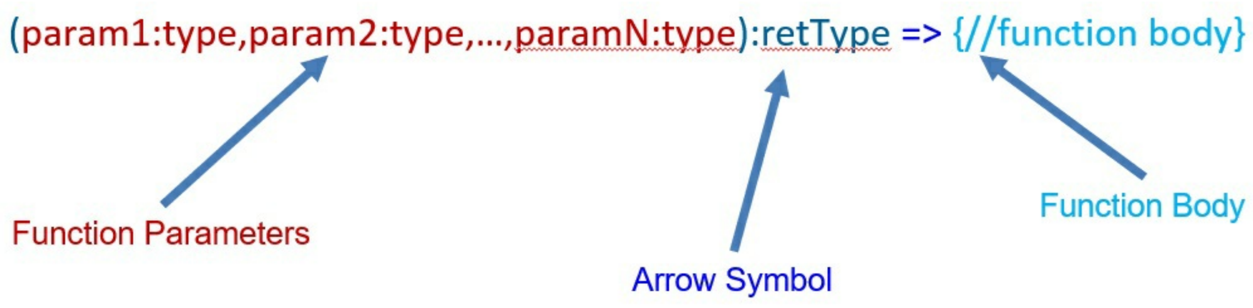
This parameter is an anonymous function. It is a function that takes no arguments, and contains the statements inside the {} block.

> *NOTE: We are not calling this method, we are just passing it in as an argument to describe or test.*

We could do this the hard way, and create a named function and pass that as the second parameter, but we are only using it once, and it is much easier to see what is going on this way.
Anonymous functions behave like any other function. We can declare them, call them, and pass them around as parameters to functions. Functions in typescript are what is referred to as ***first class objects***.

Syntax

Let’s look at the overall structure of an anonymous function:


(param1:type,param2:type,...,paramN:type):retType => {//function body}
Function Parameters
Arrow Symbol
Function Body

So what can we do with this:

- We have already seen that we can pass it as a parameter to another method as in “describe” and “test”

- Many methods in typescript can take a function as a parameter including filter, map, find, reduce, etc. We can use anonymous functions there as well.
- Since functions are first class objects, we can also store them in variables (i.e. function as value)

Example

Let’s look at an example of removing negative values from a list.
We already know how to do this with a for loop.
We can iterate through the list, adding non-negative numbers to a new list, which we then return.

```
 let arr:number[]=[1,-2,3,-4,5];

function removeNegativesFor():number[]{
    const newArr:number[]=[];
    for (let num of arr){
        if(num>=0){
            newArr.push(num);
        }
    }
    return newArr
}
```

There is another way to accomplish this using the typescript Array.filter method

```
 let arr:number[]=[1,-2,3,-4,5];
function removeNegatives():number[]{
    return arr.filter((x:number)=>{
        return x>=0
    });
}
```

The filter method takes a function that returns true if we want the value included in the returned list, and false if we want it removed from the list.
Here the anonymous function is: `(x:number)=>{return x>=0}`.
Now we can use filter to filter any list by providing such a method to specify what we want in the list.

If an anonymous function only contains a single statement that returns a value, then we can shorten this syntax by removing the braces and the return.
Now the anonymous function is: `(x:number)=>x>=0`
This gives a clean concise way to pass around simple methods without naming them.

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

But wait, there's more.

First Class Objects

Being ***first class objects*** functions can be used in many places.

- As a paramter to methods

```
 function removeNegatives():number[]{
     return arr.filter((x:number)=>x>=0);
}
```

- As the value of a variable or class property

```
 let f:(z:number)=>boolean=(x:number)=>x>=0;
```

- As the return value of a function

```
 function getGTFunction(num:number):(x:number)=>boolean{
     return (x:number)=>x>num;
}
```

Functions have types

A function type is defined by its parameters and return type We can define variables to be of that type, then store functions in that variable.

```
 let f:(x:number)=>Boolean
f=(x:number)=>x>=0;
```

We can them call those functions just like we would if they were defined with a name.

```
 let f:(x:number)=>boolean=(x:number)=>x>=0;
let b=f(4);
let c=f(-4)
```

We can even declare a type to use for our functions.

```
 declare type ChkFunction=(x:number)=>boolean;
let f:ChkFunction=(x:number)=>x>=0;
```

Summary

Annonymous functions are a useful shortcut for passing functionality around a program, either as a variable, a parameter, or a return value. They are typed by the types of their parameters and return value.

9) Webz Introduction

9.1) Web Basics

Learning to develop web applications is a critical skill for software developers.

Review of Web Basics

The internet is the network over which many protocols can be transmitted (like email, IM, www, etc).
DNS (Domain Name Service) is a distributed database that maps names to network addresses (e.g. udel.edu => 128.175.13.247)
One of the protocols the internet supports is Hyper Text Transport Protocol (http) or it’s secure cousin (https).
Over this protocol, we send regular text files, that contain a specialized language called HyperText Markup Language (html) that tells a web browser reading the file how to render the page.
That’s right, the web is basically just a bunch of text files (and a lot of cat videos).

HTML Basics

HTML is a simple tag based language where elements are defined with an opening and closing tag

```
 <p>Something</p>
<button>Click Me</button>
<span>Something else</span>
<br/> (has no body, so no closing tag
These tags can be nested inside of each other
<span>Hello <button>Click</button></br><span>World</span>
```

If a tag is inside another tag, it can be affected by the parent's size, position, and style.

Common HTML Tags

Common tags:

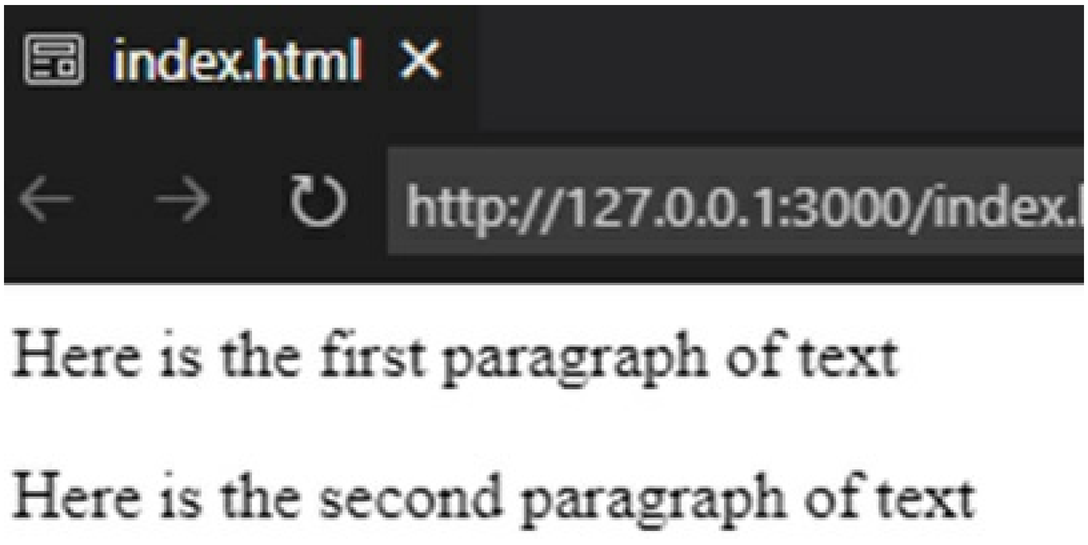
- `<div></div>` : Create a block which can be styled.
- `<p></p>` : A paragraph
- `<input type="text" />` : An input box
- `<input type="password" />` : An input box with the letters obscured
- `<input type="radio" />` : A radio button
- `<input type="checkbox" />` : A checkbox
- `<button>Button Text</button>`
- `<span></span>` : An enclosing element that doesn't do much but can be styled.
- `<select><option>1</option><option>2</option></select>` : A

drop down

HTML Tags in action

```
 <div>
   <p>Here is the first paragraph of text</p>
   <p>Here is the second paragraph of text</p>
</div>
```

Here are two of the common tag types. The outer div is not really doing anything other than grouping the other tags, but later we will learn to style that div which will make it important.



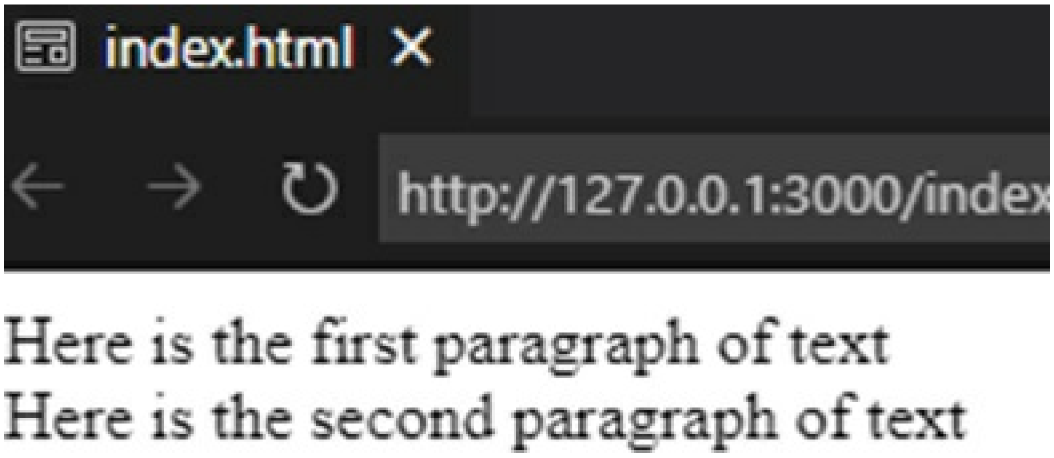
Here is the first paragraph of text

Here is the second paragraph of text

The 2 paragraph tags simply output the text to the browser with paragraph spacing between them.

If we want less spacing between the two lines of text, we can use a line break instead of putting the text in paragraph tags.

```
 <div>
  Here is the first paragraph of text<br/>
  Here is the second paragraph of text
</div>
```

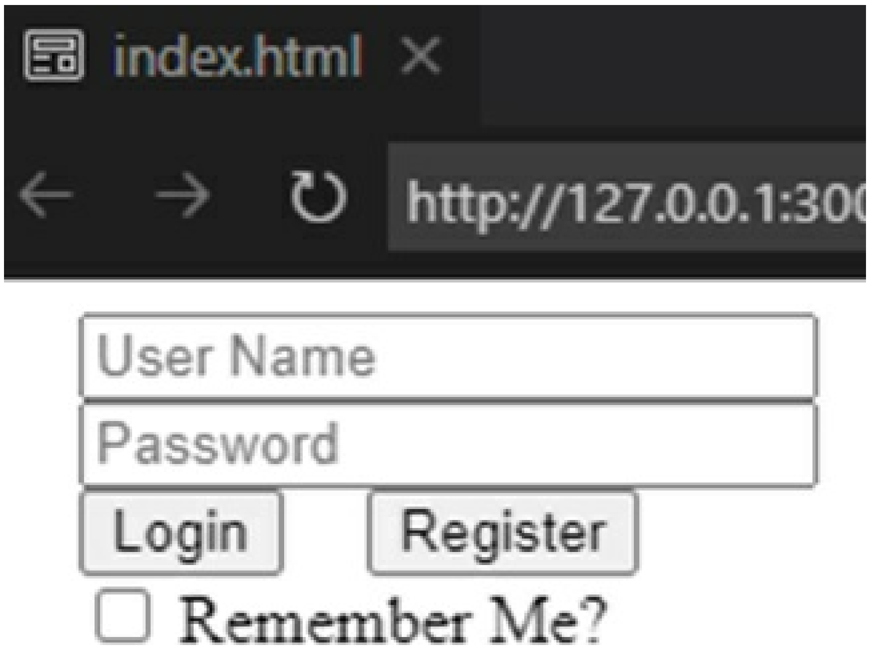


Here is the first paragraph of text
Here is the second paragraph of text

By removing the `<p></p>` tags, and adding a `<br/>` I get line spacing instead of paragraph spacing.

You can set various attributes on each tag. Here is a simple login screen:

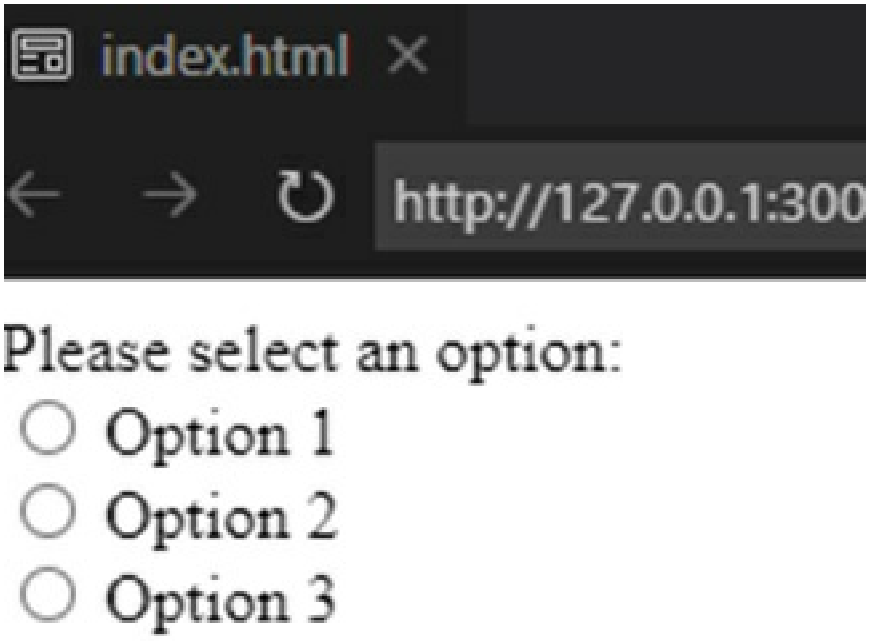
```
 <div>
    <input type="text" placeholder="User Name"/>
    <input type="password" placeholder="Password"/><br/>
    <button>Login</button>
    <button>Register</button><br/>
    <input type="checkbox"/>
    Remember Me?
</div>
```



Notice the placeholder attribute being set (and the type attribute)

Here is a set of radio buttons:

```
 <div>
    Please select an option:<br/>
    <input type="radio" name="option" value="1"/>
    Option 1<br/>
    <input type="radio" name="option" value="2"/>
    Option 2<br/>
    <input type="radio" name="option" value="3"/>
    Option 3<br/>
</div>
```

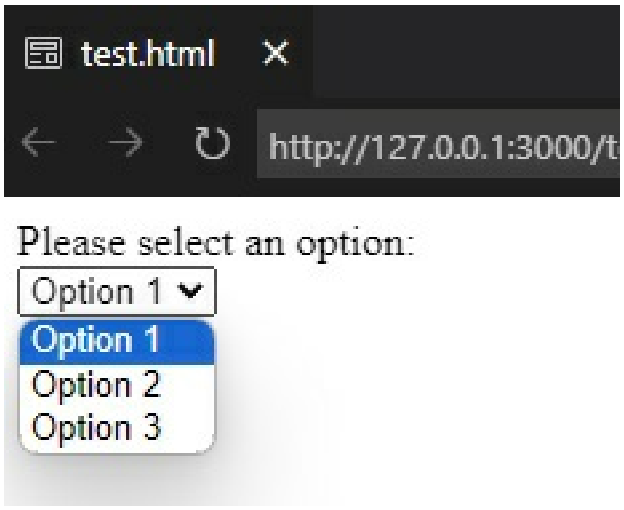


Please select an option:
Option 1
Option 2
Option 3

> *If radio buttons have the same name property, they will act as a group, where selecting one deselects the others.*

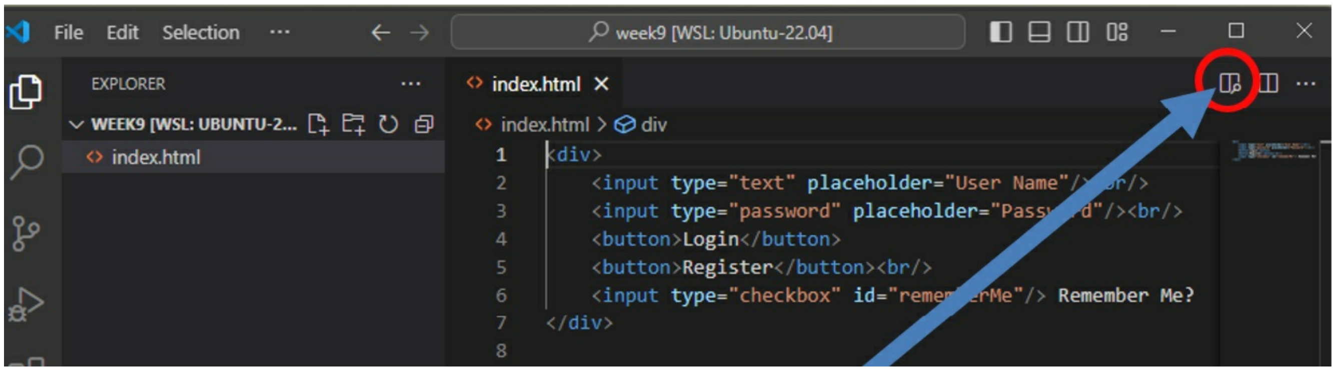
Here is a dropdown box:

```
 <div>
    Please select an option:<br/>
    <select id="options">
        <option value="1">Option 1</option>
        <option value="2">Option 2</option>
        <option value="3">Option 3</option>
    </select>
</div>
```



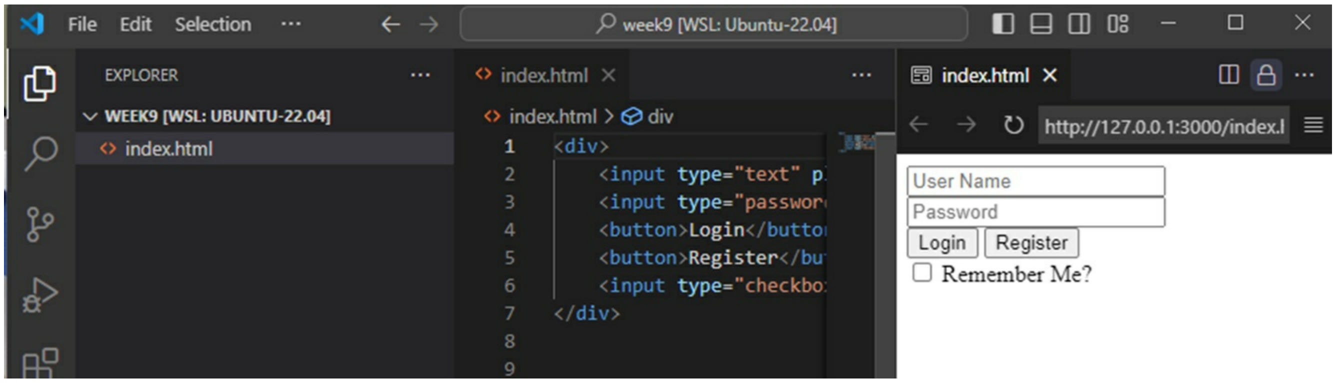
Please select an option:

Web Development in VS-Code



```
<div>
    <input type="text" placeholder="User Name"/><br/>
    <input type="password" placeholder="Password"/><br/>
    <button>Login</button>
    <button>Register</button><br/>
    <input type="checkbox" id="rememberMe"/> Remember Me?
</div>
```

Clicking this button, will bring up a preview panel on the right which will change automatically as you edit the file.



```
<div>
    <input type="text" p
    <input type="passwor
    <button>Login</butto
    <button>Register</bu
    <input type="checkbo
</div>
```

Styling and CSS

Styling our elements allows us to alter colors, shapes, behaviors, appearance, and placement.
There are basically a few ways to style:

- Style a tag: Note: This styles all tags of that type, so should not be used
- Inline style: Add the style attribute in the html and set styles there.
- Style a class: We can add one or more CSS classes to any

element, and they will take on that style. The style affects all elements with that class.

- Style a specific element: We can apply a style to an id (remember from a few slides ago). The style will only affect that element.
  Often these styles are placed in a separate file with a .css extension (stands for ***cascading style sheets***).

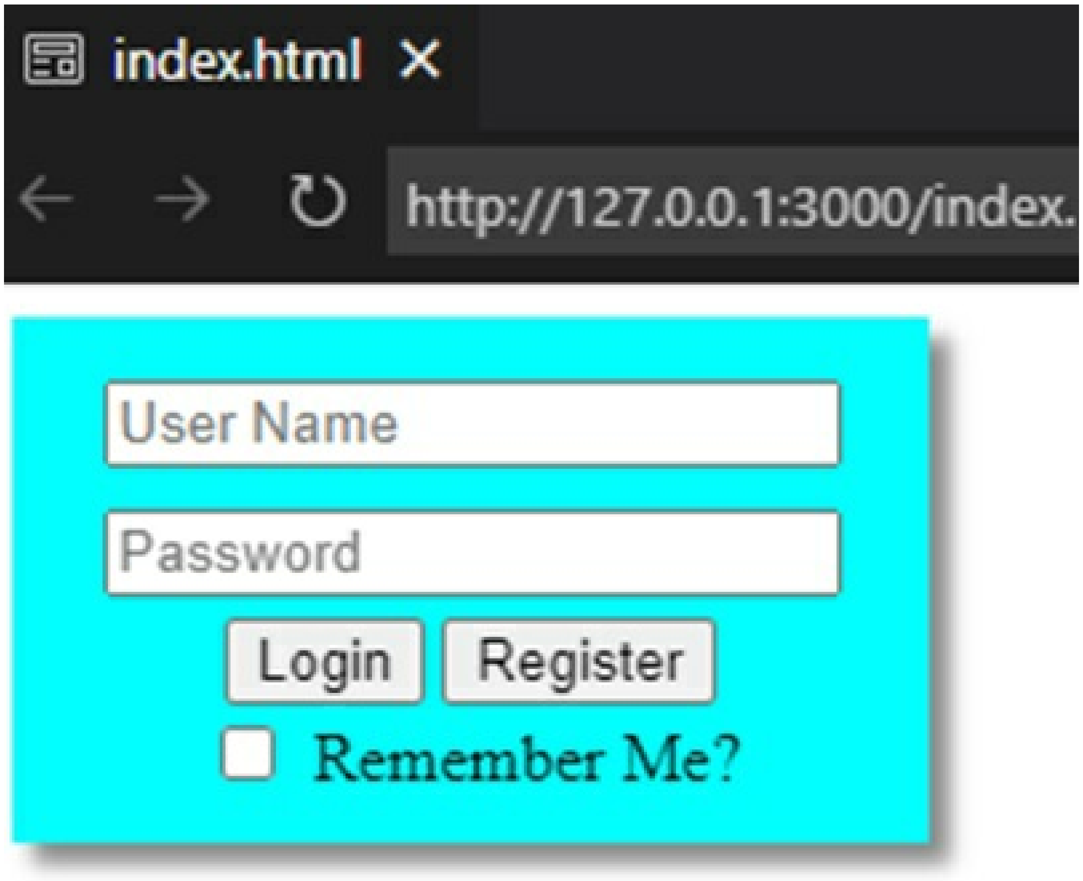
Returning to our login screen example, we have added *id* and *class* attributes to our elements to allow us to style them.

```
 <div id="loginForm">
    <input type="text" id="username" placeholder="User
Name"/><br/>
    <input type="password" id="password"
placeholder="Password"/><br/>
    <button id="loginBtn" class="btn">Login</button>
    <button id="registerBtn“ class="btn">Register</button>
<br/>
    <input id="rememberMe" type="checkbox"
id="rememberMe"/> Remember Me?
</div>
```

> *Note: This does not change the appearance, but we can use the id and class properties as references in our style sheet.*

First let's style the div. It has an id, so we will style it by id. To reference an id in a style, we put a # in front.
The style for the outer div tag is `#loginForm`

```
 #loginForm{
   text-align: center;
   background-color: cyan;
}
#loginForm input{
   margin: 5px;
}
```

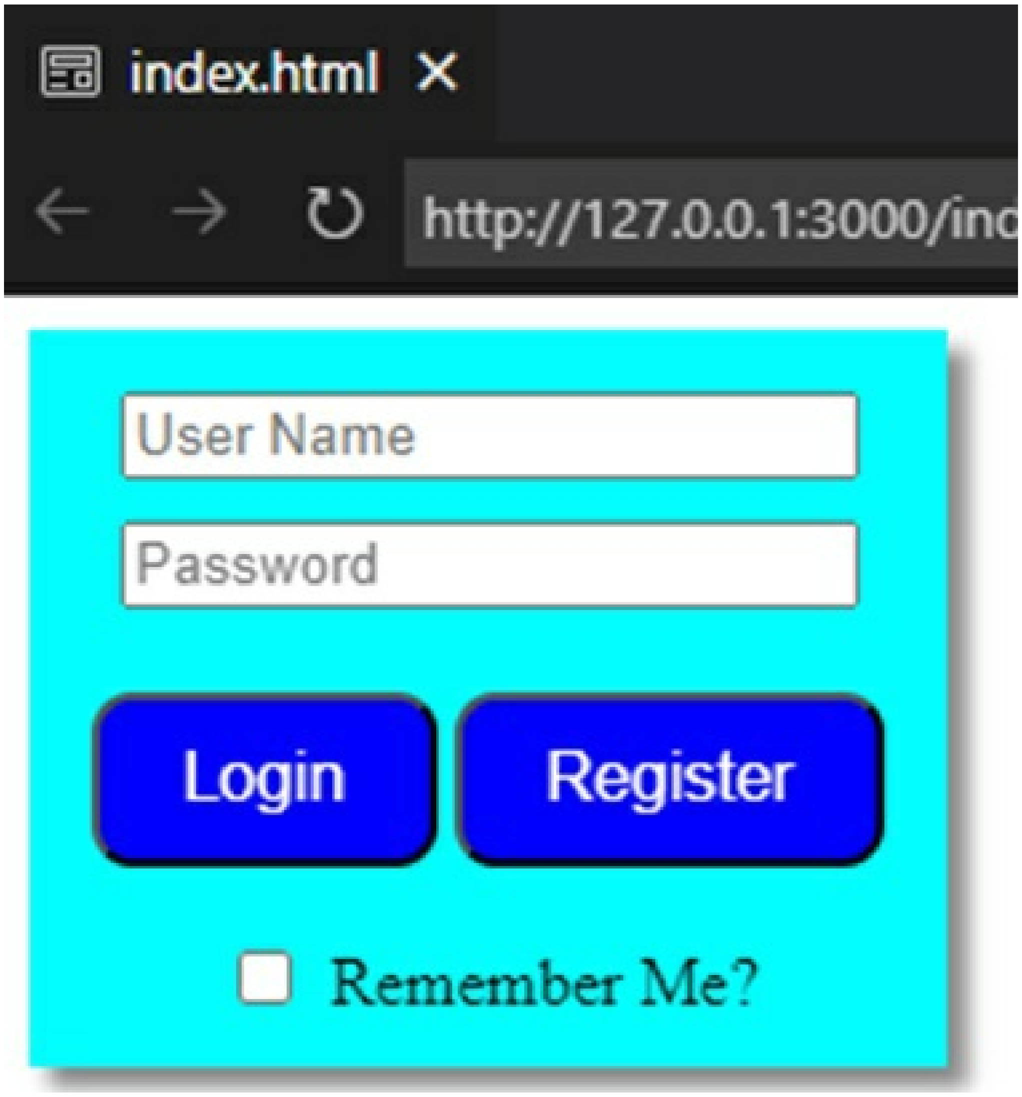


The first rule tells the loginForm div to center its content horizontally and set its background color.
The second is a combination of rule selectors. This rule says to style all input tags inside the element with id `#loginForm`.

Now our content is centered in the div, and there is a 5 pixel margin around all of the input elements in the div.

How about the buttons. They were both defined with the class `btn` and because we want them to look the same, we can use the class name to style both of them. For classes we specify the style rule by preceiding the class name with a ".", so `btn` is references as `.btn`

```
 .btn{
    background-color: blue;
    color: white;
    padding: 10px 20px;
    font-size: 16px;
    border-radius: 10px;
    margin:15px 0;
}
```



We are setting the background and foreground colors. The padding inside, the size of the text, spacing around the buttons, and making the corners round. Notice it affects both buttons.

What a difference it makes when we add just a little bit of styling to our tags.

Box Positioning

As part of styling, we have a few very important styles we will use constantly.

position: This sets how the element is positioned within its parent.

- relative: The most common. It doesn’t affect the object it is applied to, but it causes everything inside to be positioned relative to the object to which it is * applied. By default, everything is page relative (ignores the parent) unless this is set.
- absolute: Positions the object outside the normal model. This object has no affect on other objects (i.e. next object could be at the same position).
- fixed: Positioned relative to the window. It stays there.
- sticky: Positioned with a scrolling window, stays in position relative to the scroll position.

display: (Too many to list, here are the ones you will probably use)

- inline: Next element will be right after it (or on the next line if no room). Sizes to the content
- block: Element will be displayed by itself vertically and can be sized manually with width and height, or top, button, left, and right.
- inline-block: Best of both worlds. Sizable but still can be

next to each other.

Don’t Panic

There are many html tags, and styles. I am not sure anybody knows them all. They are not even 100% consistent across web browsers. You only need a few to do everything.
We will provide you with resources to look up what you need and vs code has excellent IntelliSense with html and styles.
This is an incredibly useful skill that is worth learning for your futures.

Summary

Learning HTML is an important skill as web applications are pervasive both for internet and local applications. While the number of tags and styles that are available is large, with a few tags and styles we can build beautiful web application displays. We will learn much more about this as we begin to build applicaitons instead of static web pages.

9.2) Beginning WebZ

WebZ is a lightweight web framework designed for this book. It uses many of the same principals as more advanced web frameworks such as React and Angular, but simplifies operations to focus on Typescript development

Working Example of Webz

```
 import {
    BindValue,
    BindStyleToNumberAppendPx,
    Click,
    WebzComponent,
    Timer,
} from "@boots-edu/webz";

const html = `
    <div>
        <span>Hello <span id="name"></span></span>
        <button id="button">Click me!</button>
        <div>
            <span>Count: <span id="count"></span></span>
        </div>
    </div>
`;
const css = `
button {
    background-color: blue;
    color: white;
}
`;

class MainComponent extends WebzComponent {
    @BindValue("name")
    name: string = "World";

    @BindValue("count")
    count: string = "0";

    @Click("button")
    onClick() {
        this.count = (parseInt(this.count, 10) +
1).toString();
    }

    constructor() {
        super(html, css);
    }
}

_setIframeVisible(true);
let mainInstance = new MainComponent();
let mainElement = window.document.body;
mainElement.innerHTML = "";
mainInstance.appendToDomElement(mainElement);
console.log(mainInstance.name);
```

Overview

Once we have some html, we would like it to do something. That’s where the typescript comes in.
You can create a web application without a framework, but it can be difficult and requires a deeper knowledge of how a web browser works.
Many frameworks exist, but because they are for commercial purposes, they are large and have steep learning curves (angular, vuejs, react, etc.)
We created WebZ to be a lighter weight, easier to learn framework that will prepare you for more complex frameworks that may come later and allow you to create impressive applications without a steep learning curve (still a curve, just not as steep).

The WebZ Model

The WebZ model uses standard html and css like we talked about in the last section inside the basic unit of a component.
Every project starts with 1 component called MainComponent. It has an html file, a css file, and a ts file to get you started (and a file for your tests).
Additional components can be created and inserted into the MainComponent to build an object-oriented web application.
Some Key design points:

- The html is plain html.

- The css is plain css.
- The ts file uses decorators to attach methods and properties of the class to the html by the element's id attribute (I told you we would need it later).
  The finished product is compiled into a website that can be published on any web server.

Getting Started with WebZ

To get started, we need to install the WebZ command line tool from NPM.

```
npm i -g @boots-edu/webz-cli
```

To create a new project called Example Project, we can use the cli to build (scaffold) the code.

```
webz new first-example
```

This creates a fully working website with one component in it (MainComponent)

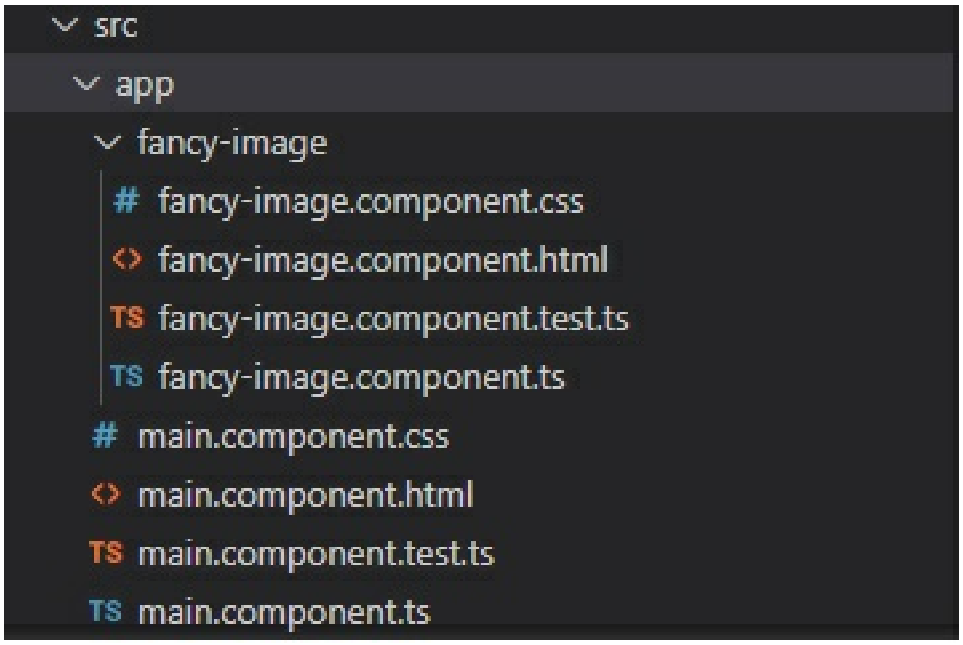
This installs a basic WebZ project with a single component in it that you can edit, and a lot of support files that you can ignore.

You are only interested in what is inside the src/app folder (src\app on Windows)

WebZ is a component-based system. Individual elements should be broken up into components and attached to the web document in the constructor.
If we navigate to the src/app folder at a command prompt, we can add more components using the CLI interface.

```
webz component fancy-image
```

This will create a folder with the 4 files in it (Just like MainComponent)



```
src
  app
    fancy-image
      fancy-image.component.css
      fancy-image.component.html
      fancy-image.component.test.ts
      fancy-image.component.ts
    main.component.css
    main.component.html
    main.component.test.ts
    main.component.ts
```

Here you can see the structure created for us. This does not attach the component to anything, it just creates the files for us.

To add the FancyImageComponent somewhere inside MainComponent we edit the files for MainComponent

First the html (main.component.html) (note the div is where the new component will go (image-holder). The buttons are to allow us to navigate later in the example):

```
 <div id="image-holder"></div><br/>
<button id="prev">Previous</button>
<button id="next">Next</button>
```

And the typescript class (main.component.ts):

```
 export class MainComponent extends WebzComponent {
     private fancyImg:FancyImageComponent = new
FancyImageComponent();
     constructor() {
         super(html, css);
         this.addComponent(this.fancyImg,"image-holder");

     }
}
```

The div `#image-holder` is where we attach our component. We have also added two buttons which we can use to control our fancy image component.

> *We can run this to make sure it worked with npm run start. We should see the default text for fancy-image and 2 buttons on the screen.*

Now we will add the html and css for the fancy-image component. We will also put two images img1.jpg and img2.jpg into the assets folder.

Replace the html with:

```
<img id="image"></img>
```

Add CSS to set it’s size:

```
#image{
    height: 300px;
}
```

How we have a place for our image, and we have set its size. What we want to do is have a variable in our class that is the name of the image we want to display. The *image* property in the following code.

```
 export class FancyImageComponent extends EzComponent {
    public imagePath: string = "assets/img1.jpg";

    constructor() {
        super(html, css);
    }
}
```

To connect html elements and class properties, we use typescript decorators to specify how to attach that variable to the html. In this case we want to set the src attribute of the element with id image. This will cause the src attribute of the element with id image to contain the text in the member property imagePath.

```
 export class FancyImageComponent extends EzComponent {
    @BindAttribute("image", "src")  //this is the decorator
binding src attribute of element with id image
    public imagePath: string = "assets/img1.jpg";

    constructor() {
        super(html, css);
    }
}
```

> *If you run this code with `npm run start` you will see the image displayed.*

Decorator transforms

While this is nice, I would rather use a numeric value (1 or 2) to select my image. I can do that in WebZ by using a custom transform.

```
 @BindAttribute("image", "src", (imgNum: number): string =>
{
        return `assets/img${imgNum}.jpg`;
})
public image: number = 1;
```

> *Notice that we pass an anonymous function to the bind decorator that takes a number and returns a string.*

Now we can just change the image number and it will just modify the img tag to load the correct image.

Remember the buttons we added to MainComponent. What do we want them to do:

- If we are at the first image, disable the previous button.
- If we are at the second image, disable the next button.
- If next is pushed increment the image number
- If previous is pushed decrement the image number

So first we need variables to bind to the disabled attribute of the buttons so we can disable them. There is a special decorator in WebZ, `@BindDisabledToBoolean` that greatly simplifies this process for us.

```
 export class MainComponent extends EzComponent {
    private fancyImg: FancyImageComponent = new
FancyImageComponent();
    @BindDisabledToBoolean("prev")
    public prevDisabled: boolean = true;
    @BindDisabledToBoolean("next")
    public nextDisabled: boolean = false;
    . . . //rest of class omitted for brevity
```

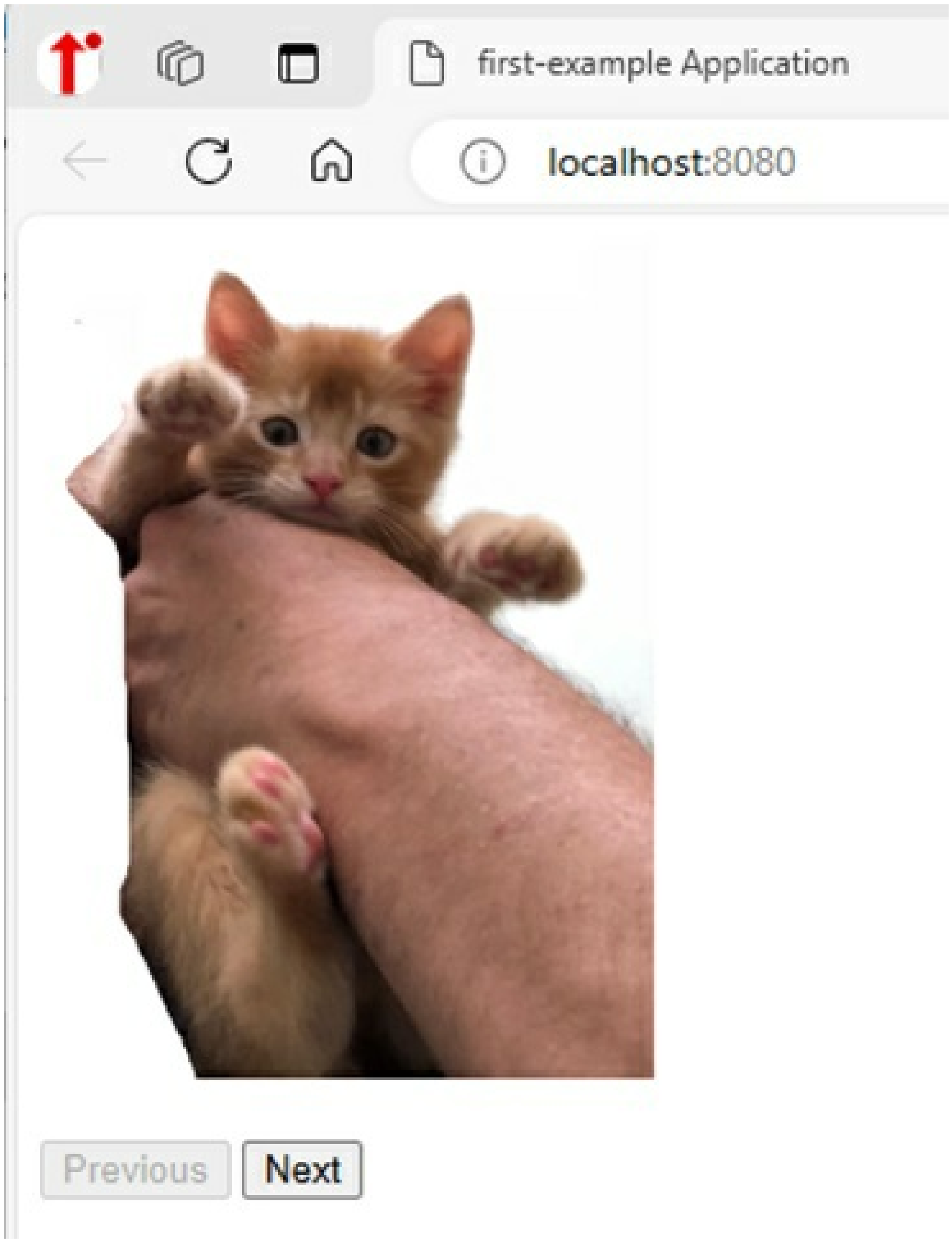
Then we need to bind the button’s click events to a function that increments/decrements the value and properly sets the values of prevDisabled and nextDisabled. Notice that the html id of the buttons is prev and next. We use that in the @Click decorator to specify which button we are binding.

```
     @Click("next")
    onNext(){
        this.fancyImg.image++;
        this.prevDisabled = false;
        if(this.fancyImg.image === 2){
            this.nextDisabled = true;
        }
    }
    @Click("prev")
    onPrev(){
        this.fancyImg.image--;
        this.nextDisabled = false;
        if(this.fancyImg.image === 1){
            this.prevDisabled = true;
        }
    }
```

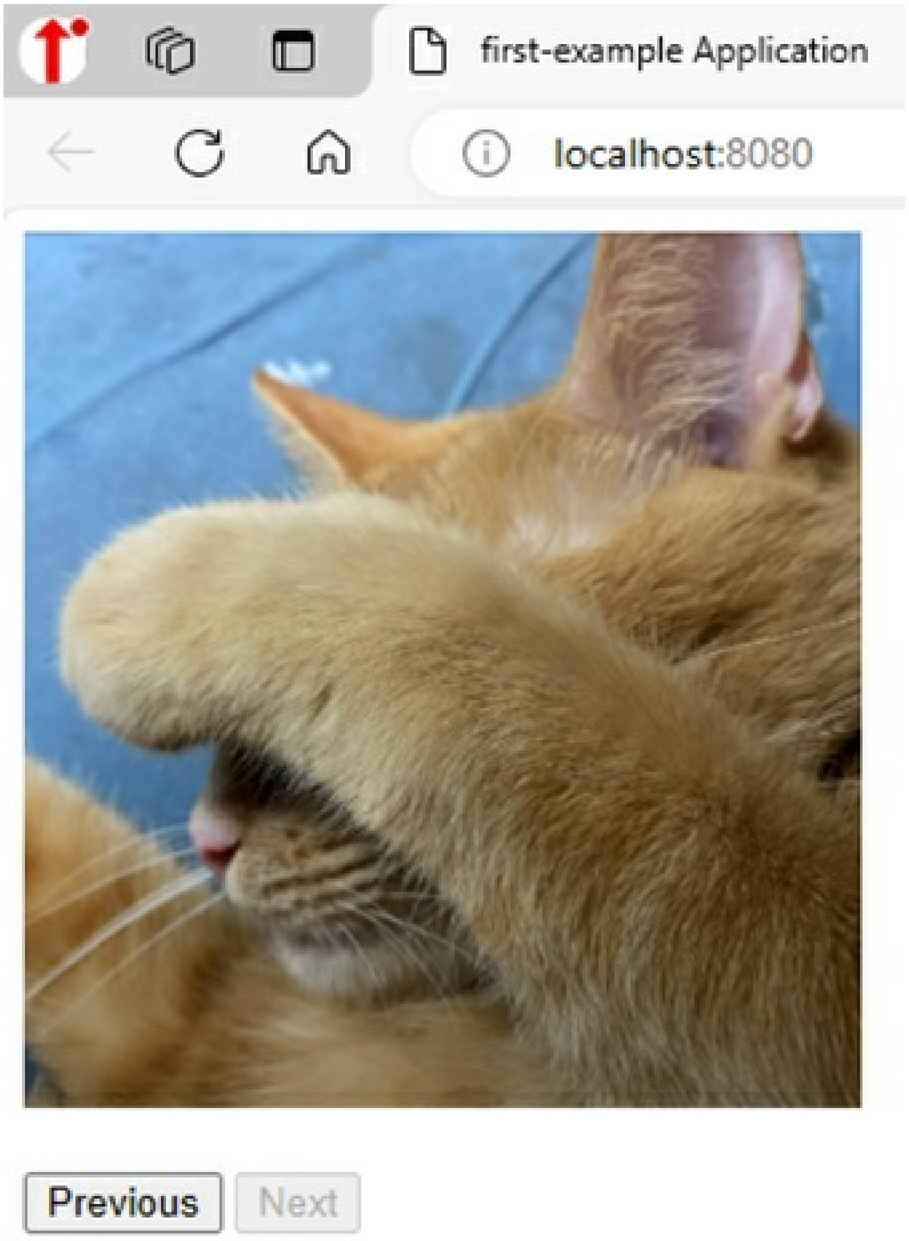
If next is pushed:

- Increment the image number
- Enable the previous button
- Disable the next button
  if prev is pushed:
- Decrement the image number
- Disable the previous button
- Enable the next button

If we run this with `npm run start` we will initially see imag1.jpg displayed and our buttons.



Clicking hte next enables the previous button, disables the next button and displays img2.jpg.



There really isn’t much more to it. Bind decorators connect properties to elements. If we change the property, the element changes (NOT THE OTHER WAY AROUND).

Event Decorators capture events from the web page allowing us to react to those events. These are decorators

like @Click(…)
We will cover some more advanced features in the next chapter, but these are the basics.

Decorators in WebZ

Bind Decorators

General:

- @BindAttribute(id,attr,?trans)
- @BindCSSClass(id,class,?trans)
- @BindStyle(id,style,?trans)
- @BindValue(id,?trans)

Specialized:

- @BindValueToNumber(id,?append)
- @BindCSSClassToBoolean(id,class)
- @BindDisabledToBoolean(id)
- @BindVisibleToBoolean(id)
- @BindStyleToNumber(id,style,?append)
- @BindStyleToNumberAppendPx(id,style)

Event Decorators

General:

- @GenericEvent(id,eventType) (e:Event)=>{}
- @WindowEvent(eventType) (e:WindowEvent)=>{}
- @Timer(milliseconds) (f:TimerCancelFunction)=>{}

Specialized:

- @Blur(id) (e:Event)=>{}
- @Change(id) (e:ValueEvent)=>{}
- @Click(id) (e:MouseEvent)=>{}
- @Input(id) (e:ValueEvent)=>{}

> *Note: You can only use these on a class derived from WebzComponent. All classes created with the CLI will automatically subclass WebzComponent.*

References

HTML References

- Intro: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basi

- Reference: https://www.w3schools.com/tags/default.asp

CSS References

- Intro: https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/Getting_started
- Reference: https://www.w3schools.com/cssref/index.php

Playgrounds

- https://playcode.io/html
- https://www.w3schools.com/tryit/
- https://jsfiddle.net/

Summary

In this section we learned about the Webz framework and how we can build a simple interactive application. The CLI can be used to generate new projects, and add components to an existing project. We can then attach code and variables to our html using the various decorators outlined in this chapter.

Chapter Summary

In this chapter we have learned the basics of web development including html and css. We have introduced ***WebZ*** which is a framework developed for this book. By binding variables to element attributes and functions to element events, we can build complex web applications.

10) Advanced WebZ

10.1) Dynamic Components

We can create and attach components dynamically in order to create complex applicaitons.

Building dynamic applications

Say we wanted to make a simple point of sale system.

- Getting the customer name and order number is easy. We just create some input boxes and bind the @Input event to a function that updates an internal variable.
- We can easily bind the @Click event of buttons to add new items to our order.
- How do we deal with a variable number of line items in the order?

Let’s start by creating a simple page in html/css for the things we know how to do, and a div to hold our line items once we create them.

```
 <div class="form-container">
    Customer Name: <input type="text" id="customerName" />
<br />
    Order Number: <input type="text" id="orderNumber" />
    <div class="detail-header">
        Order Details:         <button
id="addItemButton">New Item</button>
        <button id="addCommentButton">New Comment</button>
        <div id="counter">0</div>
    </div>
    <div id="orderDetails"></div>
</div>
```

```
 .detail-header {
    font-size: 20px;
    color: white;
    margin-bottom: 20px;
    background-color: black;
    padding: 10px;
}
#counter {
    display: inline-block;
}
```

Now in the typescript file, we need to create variables to hold our order number and customer name. These will be updated, but not bound directly (they could be).
We also need functions that are bound to the @Input event of these text boxes. Finally we need functions bound to our add buttons.

```
     orderNumber: string = "";
    customerName: string = "";
    @BindValueToNumber("counter", " items in cart")
    count: number = 0;
    @Input("orderNumber")
    onOrderNumberChange(e: ValueEvent) {
        this.orderNumber = e.value;
    }
    @Input("customerName")
    onCustomerNameChange(e: ValueEvent) {
        this.customerName = e.value;
    }
    @Click("addItemButton")
    onNewItemClick() {
        //Add the item here
    }
    @Click("addCommentButton")
    onNewCommentClick() {
        //Add the comment here
    }
```

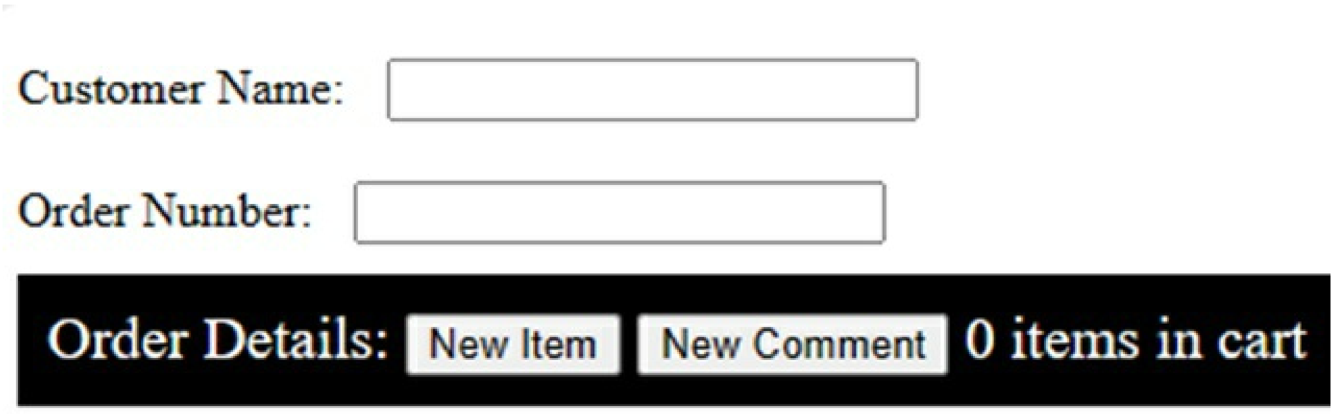
Note we have not implemented onNewItemClick or onNewCommentClick, but we have the methods hooked up to the buttons so we just have to write the contents.

So what do we do inside the click handlers? Assume we have created components for one line item or one line comment already using the cli. Then we:

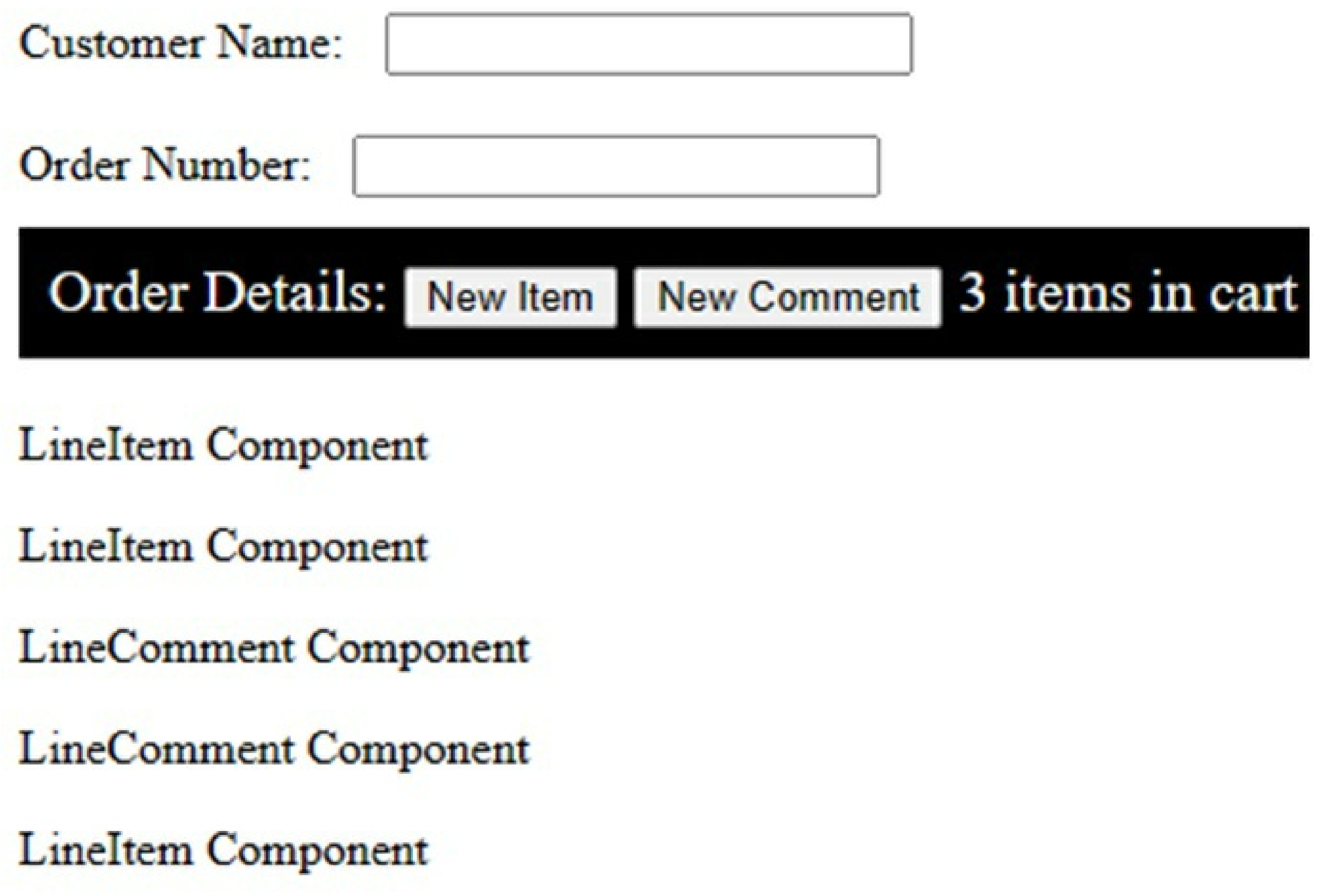
- Create the correct type of child component (info or comment) and store it somewhere so we can reference it later.
- Add it to our orderDetails div so they show up in order created where we want them.
- Increment the counter if it’s an item

> *Note: Since count is already bound to the counter div, all we have to do is update the variable to update the counter on the screen.*

```
     items: LineItemComponent[] = [];
    comments: LineCommentComponent[] = [];
    @Click("addItemButton")
    onNewItemClick() {
        const item = new LineItemComponent();
        this.items.push(item);
        this.addComponent(item, "orderDetails");
        this.count++;
    }
    @Click("addCommentButton")
    onNewCommentClick() {
        const comment = new LineCommentComponent();
        this.comments.push(comment);
        this.addComponent(comment, "orderDetails");
    }
```



This is what our website looks like when we run it. If we type in the text boxes, our member variables are automatically updated. If the count property of the class is changed, the number of items in the cart will change. If we click on our buttons, our event handlers are called. Those click handlers create a new component and add it to the orderDetails in the order they are created.


Customer Name:

Order Number:

Order Details: New Item New Comment 3 items in cart

LineItem Component

LineItem Component

LineComment Component

LineComment Component

LineItem Component

This is what our website looks like after we press new item twice, comment twice, then new item a third time. Since the click handler updated count, the correct count is displayed. The different types of line items are interspersed. They are displayed in the page where we want them since we added them to the orderDetails element.

Summary

We can create dynamic components by adding them to an existing component using the `addComponent(...)` method. Adding components dynamically allows us to create interactive web applications and reuse components (like our line item component) over and over again as appropriate.

10.2) WebZ Events

We can pass events between components so that our components can communicate.

Component heirarchy.

We can view the component hierarchy as a tree where MainComponent is the root. Each time MainComponent creates a new component, it is a *child* of MainComponent. Those children can themselves create and attach new components. What we are left with is a heirarchy of components related to each other as *parent* and *child*

Talking to our children

Talking to our children is easy. We created them, so we probably have (or at least we should have) a reference to them. Through this reference we can modify public properties and call public methods on the child. In this way we can communicate important information (that something has changed or some action is required) to the child. For deeper heirarchies, we can have each parent notify its child down the heirarchy until the child we wish to notify is reached.

Talking to our parents

Unlike communicating with children, a child likely does not have a reference to the parent object. This means we need a mechanism for a child to send inforamtion to its parent that some event has occurred.

The WebZ Notifier

We can create a public Notifier member on the child class and subscribe to that event in the parent class. That event can then be triggered in the child to notify the parent that something has changed. The contents of the Notifier can be anything we want from a value to a class to an array of either.

We will talk about generics in detail later, but we need a basic understanding in order to use Notifier.
Notifier is a Generic Class in that we can change the internal type of the class by specifying what type of event object the Notifier emits.
First, let's look at the Notifier class that we will be using:

- It has a method notify(data) that fires the event (usually called in the child component)
- It has a method subscribe((data)=>void) that attaches a function to the event when it is called.

What type of object is data? Because Notifier is Generic we get to choose.

When we create a variable of type Notifier we can supply a type parameter to tell us what type of object Notifier emits.

```
     event:Notifier = new Notifier();
    event2:Notifier<number> = new Notifier<number>();
    event3:Notifier<string> = new Notifier<string>();
    event4:Notifier<SomeClass> = new Notifier<SomeClass>();
    event5:Notifier<string[]> = new Notifier<string[]>();
```

- The first line creates an Notifier that does not emit data, just a notification.
- The second line creates an Notifier that emits a number.
- The third line creates an Notifier that emits a string.
- The fourth line creates an Notifier that emits an instance of a custom class
- The fifth line creates an Notifier that emits an array of strings.

The <> syntax is used to specify one or more ***type parameters*** that alter the class internally to support that type.

Now we can pass that type of data to the notify method.

```
 this.event.notify();
this.event2.notify(1);
this.event3.notify("hello");
this.event4.notify(new SomeClass());
this.event5.notify(["hello","eventsubject","world"]);
```

> *The type parameter specifies the type expected for the next method. Using the wrong type will be an error.*

Let’s look at the two methods we will be using (there are others, but we don’t need them yet).

notify(data:T):void

We call notify to ask the Notifier to emit our data (call any subscribed methods).
We pass it the data we want to emit which must be of the type specified in the type parameter we used to create the property.
Calling this repeatedly will repeatedly call the subscribed methods. In other words, the subscriptions last until they are unsubscribed.

subscribe((data:T)=>void,?(err:Error)=>void)

When the child calls `notify(...)`, the method passed in the first parameter is called. When the `error(...)` method of the Notifier is called, the function in the second parameter is called. The second parameter is optional.

In the parent, we can call subscribe to attach an anonymous function that will run each time the child calls `notify(...)` and optionally, another to hand when the child calls `error(...)`.

```
 this.event2.subscribe(
     (value: number) => {
         console.log(value + 1);
     },
     (err: Error) => {
         console.error(err);
     },
);
```

- event2.next(4) called in the child would print 5 from the parent.
- Event2.error(new Error("Bad stuff")) would print the error object as an error from the parent.

> *Note: This second parameter is optional if you don't want error notifications.*

Let’s apply this to our LineCommentComponent from the previous seciton. First we need to define it:

```html
 <div class="line-comment">
  Comment:
  <input id="comment" type="text" />
</div>
```

```css
 .line-comment {
    border-bottom: 1px solid black;
    padding: 10px;
}
```

export class LineCommentComponent extends WebzCmponent {
constructor() {
super(html, css);
}
@Input(“comment”)
onItemInputChange(e: ValueEvent) {
//emit the value to parent
}
}

```
 Keeping it simple, we are just getting the input event
from the text input box and calling a method
(onItemInputChange).

Now we can add our Notifier to the class and use it to
notify when the text changes:
```

```
export class LineCommentComponent extends
WebzComponent {
commentChange: Notifier = new Notifier();
constructor() {
super(html, css);
}
@Input("comment")
onItemInputChange(e: ValueEvent) {
this.commentChange.notify(e.value);
}
}
```

```
 * The public property commentChange is of type Notify and
emits a string.
* The method now calls next with the new value.

And that’s it, our child class now emits to its
subscriptions whenever the user types and this will happen
for each line comment we create.
We would do something similar for each of the fields in a
line item.

Now we have to subscribe to the event in the parent.  We
can do this when we create the child so that we will be
notified about changes to each comment.
```

```
@Click(“addCommentButton”)
onNewCommentClick() {
const comment = new LineCommentComponent();
this.comments.push(comment);
this.addComponent(comment, “orderDetails”);
comment.commentChange.subscribe((comment:string) => {
console.log(comment);
});
//Add the comment here
}
```

```
Since we never threw an error, we did not bother with a
method to handle the error.  Now each time the user types
in any of the input boxes for comments, the value will be
logged by the parent.  Of course so far, we are just
looging the comment.  Let's create an array and counter to
store the comments and update them when they change.
```

```
private commentText:string[]=[];
private commentCount:number=0;

@Click("addCommentButton")
onNewCommentClick() {
const comment = new LineCommentComponent();
this.comments.push(comment);
this.commentText.push("");
this.addComponent(comment, "orderDetails");
let index = this.commentCount++;
comment.commentChange.subscribe((comment: string) =>
{
this.commentText[index] = comment;
console.log(this.commentText);
});
}
```

```
Now we can push a new empty string onto our new array when
we create the comment and update it in our subscribe
```

```
callback.
Now at any point, commentText in the parent contains the
current value of all the comment children.

We could do something similar for our line items and store
them in an array itemList.
If we added a save button in the parent, we would have the
commentText and itemList which we could save in any way we
want.
As we type in the child component, it catches the Input
event and emits the current value through the Notifier of
each line item.
Each time the Notifier emits a value, we update the
appropriate element in our value array.
Now we don’t even need to think about what is going on in
the child in order to get the values from them.
We could do this by querying each child component for its
value when we need it, but this is a much nicer solution
and gives us real time updates in the parent class.



## Summary
Passing information between components is critical to
developing web applications.  Passing information is simple
from parent to child because the parent created the child
and thus has a reference to it.  Passing information from
child to parent can be accomplished in WebZ using the
***notifier*** class.  A child calls the ```notify(...)```
method, and the parent can subscribe to that Notifier.



<div style="page-break-before: always;"></div>

# 10.3) WebZ Dialogs
```

```
Popup windows and dialog boxes can be challenging in web
development.  WebZ provides an easy mechanism for creating
them.

## Overview
Sometimes we want to create an overlay window that sits on
top of our page, prevents us from clicking elsewhere on our
page, and has its own content and behaviors.
WebEz provides two methods for doing this:
* Popup: Creates a popup window with a title, some text,
and buttons that returns the text on the button through a
Notifier.
* Dialog: Creates a popup window whose content is
determined by a component.  These can be created with the
cli (webz dialog my-dialog).

## Popups
The popup window is provided as an easy way to interact
with your user for a quick message or question.  This is
similar to the javascript alert/confirm methods, but looks
a lot better and is more flexible.  To show a popup we
simply call the popup method of the WebzDialog class.  This
is a static method (means it does not exist on an instance
of WebzDialog but rather can be called directly on the
type).
```

EzDialog.popup(attachTo: EzComponent, message: string, title?: string, buttons?: string[], btnClass?: string):Notifier

```
We can call this method to show a dialog box:
```

WebzDialog.popup(this,"Hello World","I am the title",[“Yes","No”],"btnClass");

```
 Popup returns a Notifier<string> which emits the text of
the button pressed.
We can subscribe to the returned value to be notified when
the popup closed.

Let’s examine this in detail:
* attachTo (required): is the component that you want to
attach the element to.  Usually you will pass in this to
specify the current component.
* Message (required): The text inside the popup
* Title (optional): The title for your popup, displayed at
the top.
* buttons (optional): An array of strings that are the
labels of the buttons that you want to display.  By default
there is a single OK button.
* btnClass(optional): An optional css class string to style
the buttons.  This allows you to optionally attach a css
class to the button for styling.
* Returns: An event subject that emits the label of the
pressed button.
Pressing a button, closes the popup.

Back to our point of sale example, we can use a popup to
notify the user that a comment was added.
```

@Click(“addCommentButton”)
onNewCommentClick() {
const comment = new LineCommentComponent();
this.comments.push(comment);

this.commentText.push(“”);
this.addComponent(comment, “orderDetails”);
let index = this.commentCount++;
comment.commentChange.subscribe((comment: string) =>
{
this.commentText[index] = comment;
console.log(this.commentText);
});
WebzDialog.popup(this,"Item added.")
}

```
 Here we have added a popup with the default title “Alert”
and the default buttons [“Ok”].
We have not subscribed since I do not need notification of
when the window closes, and there is only one button the
user could have clicked.
We could have subscribed if I needed to know that the popup
was closed.

![]
(/home/runner/work/textbook/textbook/source/assets/images/w
ebz_6.jpg)

Notice how the popup greys out the underlying website.  You
cannot click buttons or enter text while the popup is open.
Once it is closed, the gray background goes away and the
rest of the page will again accept input.
With this screen, the only thing I can do is click ok.

Let's look at a more complex example:
 @Click("addCommentButton")
onNewCommentClick() {
    WebzDialog.popup(this, "Are you sure?", "Add new item",
["Yes","No"],
    ).subscribe((result: string) => {
        if (result == "Yes") {
            const comment = new LineCommentComponent();
            this.comments.push(comment);
            this.commentText.push("");
            this.addComponent(comment, "orderDetails");
            let index = this.commentCount++;
            comment.commentChange.subscribe((comment:
string) => {
                this.commentText[index] = comment;
                console.log(this.commentText);
            });
        }
    });
}
```

```
Here we have added a popup to ask the user if they are sure
before adding the item, and then only adding it if they
click the “Yes” button.
We subscribe to the EventSubject returned by the popup
method to see when the window closes and which button was
pressed.

> Note: We moved all of the code inside the anonymous
function so that it will only be called after the dialog is
closed.

![]
```

```
(/home/runner/work/textbook/textbook/source/assets/images/w
ebz_7.jpg)

## Dialogs

Dialogs work similarly, except they do not have a pre-
defined structure.  You can create them as a component
where you control the layout and any Notifiers you want to
implement.
Creating a new dialog is as simple as: ```webz dialog
myDialog```

Like webz component, this creates a new component, but it
will behave and look like a popup window, only its content
will be your new component.
The default implementation is a simple popup with an ok
button that closes when the user clicks it.  We can close a
window by calling the member method this.show(true/false).
We add it just like any other component using addComponent,
then display it by calling show(true).
```

dialog: MyDialog = new MyDialog();
constructor() {
super(html, css);
this.addComponent(this.dialog);
}
showDialog(){
this.dialog.show(true);
}

```
hideDialog(){
this.dialog.show(false);
}
```

~~~~
First we create a variable to hold our dialog:
We can then add it to the component:

> Note, if you want it to display immediately, then you can
call show with true ```this.dialog.show(true);```

Whenever we want to show the dialog, we just pass true to
it’s show method.  To hide it we pass false.

If we want to get an event to subscribe to when the window
is closed, or something happens in the dialog, we can
implement our own Notifiers and subscribe to them in the
parent.

### An Example
A simple please wait dialog with no buttons.
To make this simple, I am just going to use text, but you
could use an animated gif or do some css magic to add some
movement to this dialog (we will do that in a few minutes
with a timer).
First we will create a new dialog with the cli: ```webz
dialog Pleasewait```

For the body of our dialog, we will just center a string
that says “Please Wait…”
```html
<div class="content">
    <div class="body">Please Wait...</div>
</div>
~~~~

```
.content {
    width: 600px;
}
.body {
    text-align: center;
    font-size: 40px;
    line-height: 100px;
}
```

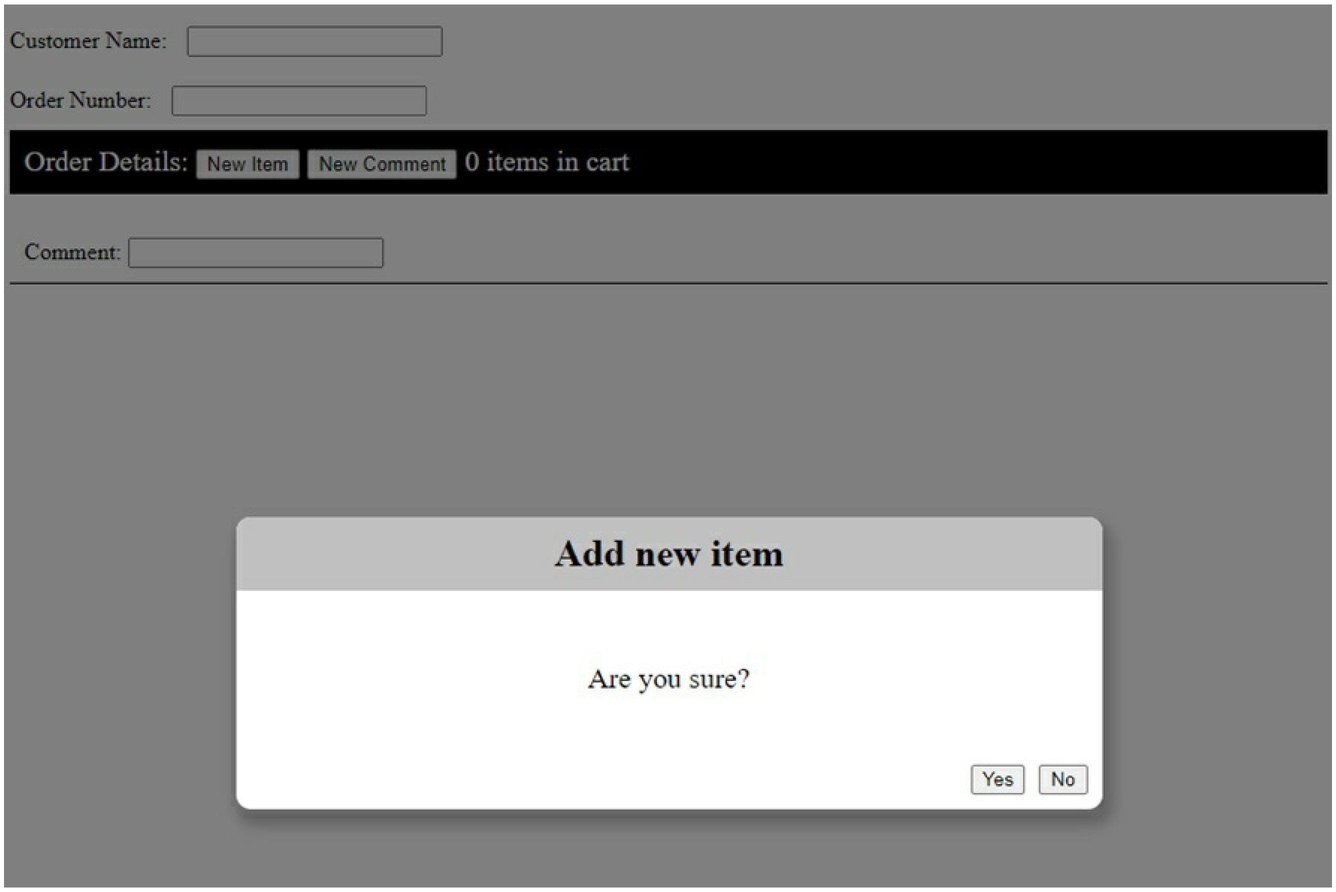
In the parent, we create a property for our dialog and add it to the component.

```
plsWait: PleaseWaitDialog = new PleaseWaitDialog();

constructor(){
    this.addComponent(this.plsWait);
}
```

When we want the dialog we can simply display it while some time consuming task is occurring, then hide it after.

```
this.plsWait.show(true);
//do something that takes a while
this.plsWait.show(false);
```



Here you can see the output after a call to plsWait.show(true).
Just like the popup, the rest of the website is grayed out and cannot be interacted with.

Summary

Creating a good user interface is critical to having your software accepted by users. Dialogs and popups are an excellent mechanism for communicating and querying simple information from the user. WebZ provides a simple ***popup*** method for simple interactions, and a ***dialog*** class to derive from for creating custom layouts.

10.4) WebZ Timers

We can cause things to happen periodically by using a ***timer***. Timers allow us to schedule actions to occur once per time interval.

Overview

Sometimes we want to do something periodically while our site is displayed

- Update a timer
- Refresh data from a backend
- Move a game element

- Animation
- Anything else we want to accomplish on an interval.

This can be useful to provide more interactivity to your site.

Using Timers

Returning to our PleaseWait dialog, we can use a timer to make it more interesting.
First, we will bind a variable to the text we are displaying:

```
 @BindValue(“displayDots")
displayDots: string = "";
```

We will modify the html and add a div with the id displayDots.

```
 <div class="content">
    <div class="body">
        Please Wait
        <div id="displayDots"></div>
    </div>
</div>
```

And style it so that it has a fixed width and will appear inline after the words Please Wait.

```
 #displayDots {
    width: 50px;
    display: inline-block;
    text-align: left;
}
```

The plan is to change displayDots to contain 1, 2, or 3 dots and change it once a second.

To implement the behavior, we will use the ***@Timer*** decorator to decorate a function hat we want called periodically.
Passing 1000 to the timer method causes onTimer to be called once a second while the page is displayed (forever: more on this later). 1 second= 1000 milliseconds
Each time it is called, we check a counter that will keep track of how many dots are displayed. When we get to 3, we set it back to 0. Otherwise, we draw the correct number of dots (count+1 because count goes from 0-2) by updating our displayDots property which is bound to the page.

```
 @Timer(1000)
private onTimer() {
    if (this.count === 3) this.count = 0;
    this.displayDots = ".".repeat(this.count + 1);
        this.count = this.count + 1;
    }
```

Summary

We can use a timer to cause a function to be called periodically. The @Timer directive takes the number of milliseconds between calls, and runs until the page exits.

11) Advanced Typescript

11.1) Typescript Generics

Generics allow for creation of reusable code that where internal types can be specified externally.

Generics in Typescript

In the last chapter we discussed the WebZ *Notifier* class. This class was a ***generic*** class that we could pass ***type parameters*** to during creation.

```
 event:Notifier = new Notifier();
event2:Notifier<number> = new Notifier<number>();
event3:Notifier<string> = new Notifier<string>();
event4:Notifier<SomeClass> = new Notifier<SomeClass>();
event5:Notifier<string[]> = new Notifier<string[]>();
```

This is a single class definition that works on any type of data. We can make our own generic functions, classes, interfaces, or type aliases by creating them with one or more ***type parameters*** that can be specified by the caller. Overall, this allows us to create reusable code that works on various types of data.

Motivation

Consider the following simple method.

```
 function printNumberResult(result:number){
     console.log('Result: ' + result);
}
printNumberResult(5);
```

This method prints *Result: 5* when called with a parameter of 5. What if we wanted to allow other types of data to be printed? One solution would be to write another function.

```
 function printStringResult(result:string){
     console.log('Result: ' + result);
}
printStringResult("Hello World");
```

While we could write different functions for each type we wish to support, it would be better if we could right a single method for all of them. Let's examine this code further:

Generic Functions

We know `console.log(...)` will print anything, so the only issue here is that our method expects a number. We can make this function a ***generic*** by adding a ***type parameter*** and using it as the type of the result parameter.

```
 function printResult<T>(result:T){
    console.log('Result: ' + result);
}
printResult<number>(5);
printResult<string>("Hello World");
```

Here we have added a ***type parameter*** (T), and we use that paramter to set type type of the function's parameter (result). When we call our function, we can specify the type of the data when we call it.

It turns out that typescript can *infer* the type from the parameter, so we can leave it out when we call the function (However it is not incorrect to include it).

```
 function printResult<T>(result:T){
    console.log('Result: ' + result);
}
printResult(5);
printResult("Hello World");
```

We are not limited to a single type parameter. If we need more than one, we can specify multiple type parameters.

```
 function makePair<T,S>(x:T,y:S):[T,S]{
    return [x,y];
}
const result=makePair<number,string>(1,"hello");
console.log(result);
```

The important point here is that the type checking occurs at compile time (not at run time). If we call it with the wrong arguments...

```
 function makePair<T,S>(x:T,y:S):[T,S]{
    return [x,y];
}
const result=makePair<number,string>("hello“,1);
console.log(result);
```

... you will get compiler errors. Try it and you will see the errors in the console.

> *It is much easier to fix compiler errors where the compiler gives us a line number and description then it is to fix run time errors where the program either crashes, or just gives the wrong answer.*

Controlling types

We can limit the types that are acceptable as a type parameter by using the extends keyword. In this example, the first parameter must be a string or a number, but the second parameter can be any type.

```
 function makePair<T extends string|number,S>(x:T,y:S):
[T,S]{
    return [x,y];
}
console.log(makePair(4,["Hello"]));
```

*Note: string|number is referred to as a **Union Type** which we will talk more about later, but basically we can combine types with a | and then either type would be acceptable.*

If we use *extends* with a class type, we could use elements of that class or any class that derives from the class specified in the ***type paramter's*** extends clause.

```
 class Shoe{
    constructor(public size:number){}
}
class Sneaker extends Shoe{
    constructor(size:number,private sport:string){
        super(size);
    }
}
class Boot extends Shoe{
    constructor(size:number,private height:number){
        super(size);
    }
}
function getShowSize<T extends Shoe>(shoe:T):number{
    return shoe.size;
}
console.log(getShowSize(new Boot(10,14)));
```

> *Note: We could do this without a generic if we made the parameter type Shoe as it would accept the derived classes. In this case either method is ok, but there are places where a generic is a better solution.*

Generic Classes

Just like functions, we can use generics for classes as well. Let’s consider a class for a list of numbers:

```
 class ItemList{
    public items:number[]=[];
    constructor(){}
    addItem(item:number):void{
        this.items.push(item);
    }
}
const list:ItemList=new ItemList();
list.addItem(4);
console.log(list);
```

What if we wanted to extend this so it worked on a list of any type, even a list of lists.
We could use a generic definition to make ItemList work on any type, and not just on numbers
As always we can limit the acceptable types using the extend keyword.

```
 class ItemList<T>{
    public items:T[]=[];
    constructor(){}
    addItem(item:T):void{
        this.items.push(item);
    }
}
const list:ItemList<number>=new ItemList<number>();
list.addItem(4);
const list2:ItemList<string>=new ItemList<string>();
list2.addItem("hello");
const list3:ItemList<string[]>=new ItemList<string[]>();
list3.addItem(["Hello","World"]);
console.log(list);
console.log(list2);
console.log(list3);
```

> *Note: T is defined on the class, and we can use it within the class as the type of any method parameter, return value, or member variable.*

We can create a homogeneous list of anything by specifying the type of object the list contains with a ***type parameter***.
Now we have created a class that works on any data, instead of just on numbers.
We can even add additional type parameters to the methods within our class to make them more reusable.

Default Types

Finally, we can provide a default value for our generic to describe how it behaves if no type parameter is provided:

```
 class ItemList<T=number>{
    public items:T[]=[];
    constructor(){}
    addItem(item:T):void{
        this.items.push(item);
    }
}
const list:ItemList=new ItemList();
list.addItem(4);
const list2:ItemList<string>=new ItemList<string>();
list2.addItem("hello");
const list3:ItemList<string[]>=new ItemList<string[]>();
list3.addItem(["Hello","World"]);
console.log(list,list2,list3);
```

If a parameter is provided, the default is ignored.
If no parameter is provided, then thetype must match the default if we use the class (i.e. we must pass a number, anything else will cause a type error at compile time).

Inside the WebZ Notifier class

Let’s return to the WebZ *Notifier* class and loot at the source code for it.

```
 export class Notifier<T = void> {
    constructor() {}
    subscribe(callback: (value: T) => void, error?: (value:
Error) => void) {
        //something goes here
    }
    unsubscribe(id: number) {
        //something goes here
    }
    notify(value: T) {
        //something goes here
    }
    error(value: Error) {
        //something goes here
    }
}
```

*T defaults to void if no parameter is provided.

- subscribe takes a function whose parameter has type T.
- notify takes a value of type T

This is as expected when you consider how we used Notifier previously.

- With no type argument its data is void (nothing)
- With a type parameter, the type it works with is the value specified for T.

Summary

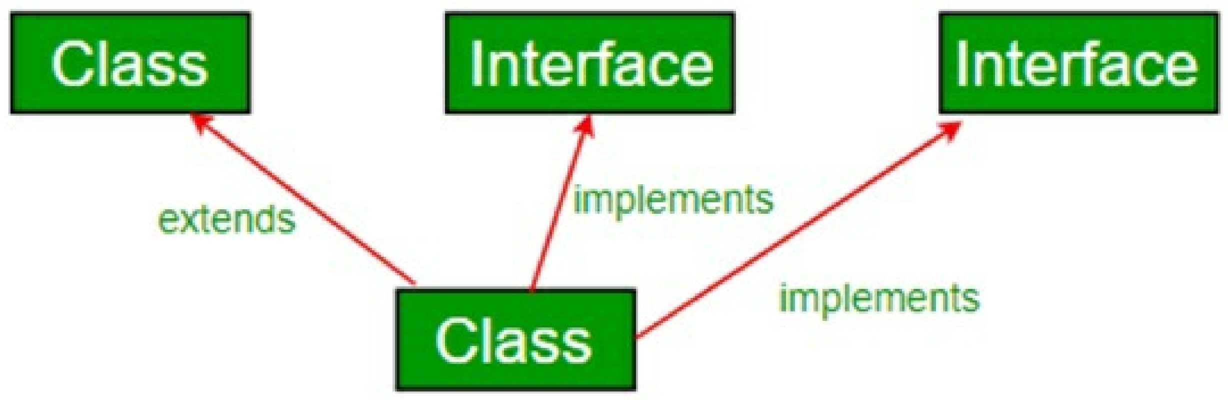
Using ***generics***, we can create more reusable code by allowing our code to work on many different types of data. We can apply this techinque to classes and methods so that our code works on various types of data.

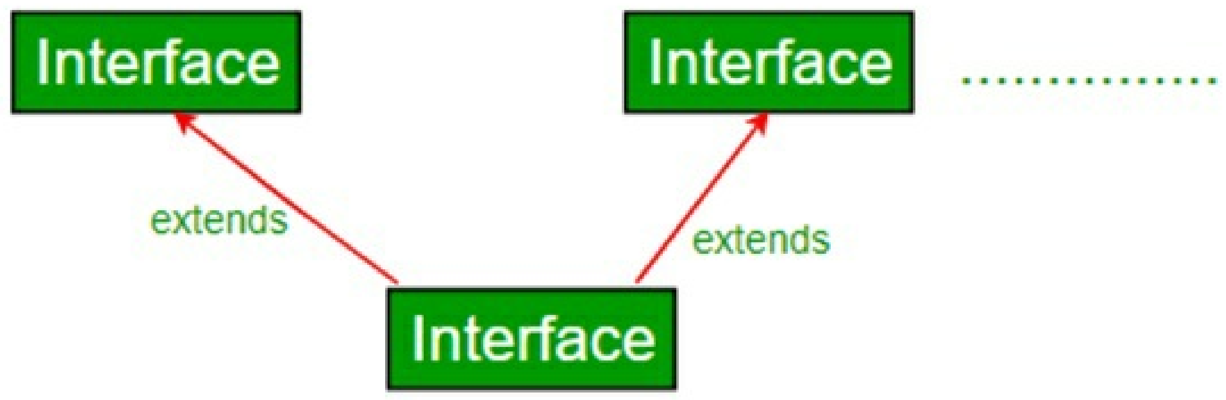
11.2) Typescript Interfaces

An ***interface*** is a contract that describes the shape of data without values or implementation.

Interfaces in Typescript

Sometimes we want to describe the shape of our data.
Sometimes we want to describe the methods and values that a class contains without detailing the entire class.
We can only extend one class, but we can implement many interfaces.





We say that it is a contract, because the object must implement the things in the interface, and users of the object are guaranteed that those things are implemented.

> ***Interfaces** can contain property or method signatures but not implementations.*

A simple example

Suppose we are building a drawing program and want to be able to pass around point structures {x:number,y:number}. We can declare this as an interface then use the interface name as a type.

```
 export interface Point{
     x:number;
     y:number;
}
let point:Point={x:1,y:2};
let point2:Point={x:2,y:3,z:4};  //this is an error
let point3:Point={x:3};  //this is an error
```

> *Note this will give an error because point2 and point3 don’t conform to the interface.*

We can’t create a point (with new) like a class, but the compiler will guarantee that the object contains the members of the interface and only the members of the interface.

We say that a class ***implements*** an interface if it contains all of the members of the interface (not necessarily only those members). Using the ***implements*** keyword guarantees this.

```
 interface Point{
    x:number;
    y:number;
}
class DrawPoint implements Point{
    x:number;
    y:number;
    constructor(x:number, y:number, private color:string){
        this.x=x;
        this.y=y;
    }
}
const point:Point=new DrawPoint(4,5,"red");
```

Now I can refer to the DrawPoint object as a Point and I know it contains an x and a y without having to know anything else about DrawPoint.

We are guaranteed that DrawPoint contains an x and a y member, because it implements point. If it doesn't, the code won't compile.

Interface methods

Interfaces can contain methods as well. They don't include the implementation, they are just stating that the class must contain that method in order to compile, so users of the class know it contains that method.

```
 interface Drawable{
    points: Point[];
    draw():void;
}
class Triangle implements Drawable{
    points:Point[];
    constructor(p1:Point,p2:Point,p3:Point){
        this.points=[p1,p2,p3];
    }
    draw(){
        //draw triangle
    }
}
```

Here we can see that Triangle contains an array of Point objects, and a draw method, therefore it correctly implements the Drawable interface.
If a class implements an interface, then the class can be referenced as an object of interface type and will always have the points array and draw method or it would not compile.

Multiple Interfaces

A class cannot extend more than one class in typescript, but it can implement many interfaces.

```
 interface Serializable{
    serialize():string;
}
interface Iterable<T>{
    next():T;
}
class MyList implements Serializable,Iterable<number>{
    values:number[]=[];
    pos:number=0;
    next():number{
        if (this.pos<this.values.length)
            return this.values[this.pos++];
        else
            return -1;
    }
    serialize():string{
        return JSON.stringify(this.values);
    }
}
```

Here we have a class that implements two interfaces. We can see that it provides the implementation that matches the signatures in all of the interface.

Now I can use it to write a function:

```
 function serializeAll(obj:Serializable[]){
    let result:string[]=[];
    for (let o of obj){
        result.push(o.serialize());
    }
    return result;
}
const obj:MyList[]=[new MyList()];
console.log(serializeAll(obj));
```

Using Interfaces

Interfaces have many uses, primarily:

- Describe the shape of data to guarantee that the data is in the right form.

```
 interface Point{x:number;y:number;}
```

- Describe certain features that we want to enforce when we create a class so that if we know the class implements the interface, we know that the interface members actually exist in the class and are implemented for us.

```
 interface Drawable{points: Point[];draw():void;}
class Triangle implements Drawable{ . . . }
```

- Using interfaces we can simplify coding by having multiple (very different) classes that all implement the interface, then we can call the interface methods on the objects even though they are otherwise

```
 class Elephant implements Serializable{ . . . }
class Tomato implements Serializable{ . . . }
```

Notes on Interfaces

- Interfaces allow us to further type our data by specifying what methods and properties an object must contain.
- Unlike extending classes (inheritance), we can implement multiple interfaces in a single class.
- If a class implements an interface, then that class can be stored in a variable whose type is the interface, and we can access the interfaces members through that variable.
- Interfaces can be very useful to describe typescript objects that are otherwise untyped (like complex data returned from an API call). Once described, the interface will enforce that the object is indeed the correct shape and contains all of the interface members (methods and properties).
- Interfaces are common in most Object Oriented programming languages and provide a convenient means

to strengthen typing within our code.

Summary

Interfaces provide another powerful mechanism for creating type safe reusable code. By specifying the *contract* that a class or method must adhere to, users of the class or type can be assured that the type contains the members specified in the interface. In this way, disperate objects can be used as if they are the same object so long as they implement a given interface.

11.3) Union Types

Union types are a way of declaring a variable that can hold values of two or more different types.

Combining types in Typescript

We know we can declare new types in typescript by creating classes and interfaces, and we can use these types in our programs.
What if we don’t know the type, but we know that it one of a finite number of types:

- It could be a number or a string
- It could be a class instance or null

We can combine types into a new either/or type by creating a ***union type***.
When a variable is declared as a ***union type***, it can take on either type of value, but the value must be one of those types.

Imagine we want to create a function that pads a string on the left.

We might want it to take a string to prepend

```
 function padString(value:string,padding:string){
    return padding+value;
}
```

We might want it to take a number and add that many spaces to the front

```
 function padString2(value:string,padding:number){
    return Array(padding + 1).join(" ") + value;
}
```

It would be great if we could combine these into one function, but not allow invalid types.

We can use a ***union type*** to combine the signatures Then check the type of padding and act accordingly:

```
 function padString(value:string,padding:string|number){
    if(typeof padding === "number"){
        return Array(padding + 1).join(" ") + value;
    }
    return padding+value;
}
console.log("Answer: "+padString("World",6));
console.log("Answer: "+padString("World","Hello "));
```

We can apply this to other types as well. Classes, interfaces, etc.

Union types with classes

Consider these classes:

```
 class Tiger{
    name:string="Tony";
    getDetails():string{
        return this.name + " is a tiger";
    }
    eat(){
        console.log("Yum");
    }
}
class Tree{
    name:string="Kevin";
    height:number=100;
    getDetails():string{
        return this.name + " is " + this.height + " feet
tall";
    }
}

let whatisit:Tree|Tiger;
whatisit=new Tree();
console.log(whatisit.name);
console.log(whatisit.getDetails());
```

We can unino these classes together and through the variable *whatisit*, we can access any members that both *Tree* and *Tiger* share in common.

> *We cannot access members that are not shared in common through the variable because its type only supports members that are in both.*

Type Aliases

We can create a ***Type Alias*** to combine types, then use our type alias in our programs to represent the union type.

```
 class Tiger{
     name:string="Tony";
     getDetails():string{
         return this.name + " is a tiger";
     }
     eat(){
         console.log("Yum");
     }
}
class Tree{
     name:string="Kevin";
     height:number=100;
     getDetails():string{
         return this.name + " is " + this.height + " feet
tall";
     }
}
declare type ThingsThatStartWithT=Tree|Tiger;
let whatisit:ThingsThatStartWithT;
whatisit=new Tree();
console.log(whatisit.name);
```

The declared type ***ThingsThatStartWithT*** can be used like any other type, but it represents the ***union type*** of combining *Tiger* and *Tree*.

Summary

A simple way to combine the functionality of multiple types is a ***Union Type***. ***Union Types*** represent an either/or relationship. Variables defined as a ***union type*** can be any of the types in the statement and get any properties or methods that are shared between all of the types in the statement.

12) Higher Order Methods

12.1) Higher Order Array Methods

A ***higher order function*** is a function that takes as an argument and/or returns a function.

Higher Order Methods in General

The term ***Higher Order Method*** simply refers to any method which takes a function as an argument, returns a function, or both.
This is nothing new for us. We have seen this before in the describe and test methods we use in our jest test code.

```
 describe("MainComponent", () => {
     describe("Constructor", () => {
         test("Create Instance", () => {
         });
     });
});
```

We also saw this in the WebZ library with the Notifier class' subscribe method:

```
 child.elementAdded.subscribe(
    (value:boolean) => {console.log(value);
});
```

In Typescript, when passing a function as an argument, it is often convenient to use an ***anonymous function*** which we have talked about already. You can always spot this because it will have the => operator.

Since functions in typescript are ***first order objects***, we can use them as parameters and return values.
We can specify the shape or signature of the expected parameter or return type when we declare the method.

```
 subscribe(callback: (value: T) => void, error?: (value:
Error) => void):number
```

In this signature for the subscribe method, the parameter named callback is of type (value: T) => void and the parameter named error is of type (value:Error)=>void where T is a type parameter used when creating an instance of the class and Error is the error type provided by Typescript.

This language feature of typescript (and many other languages where functions are ***first order objects***) allows for some useful and interesting ways to write code and typescript (javascript) provides some built-in functions that take advantage of this.

Use of these built-in methods will make your code shorter, simpler and more readable. There is nothing these can do that we could not write in some other way, but they simplify things considerably. We will examine several methods that can be applied to arrays including map, filter, reduce, reduceRight, every, some, find, findIndex, findLastIndex, flatMap, forEach, and sort,

Higher Order Array Methods

The forEach Method

The simplest and most straight forward higher order array method is ***forEach*** which takes a function as its only argument and executes that function on each argument in the list. If we wanted to do this without using the forEach method, we could certainly do it with a simple for loop:

```
 const arr:string[] = ['a', 'b', 'c'];
for (let value of arr) {console.log(value)}
```

We can use our new higher order ***forEach*** method to accomplish the same thing. Notice that the only difference is that we are passing a simple method to the ***forEach*** function which accomplishes whatever we want to do in the loop body by calling that function on each element of the array.

```
 const arr:string[] = ["a", "b", "c"];
arr.forEach((value) => {console.log(value)})
```

If we want to call a member function instead, we can simply call it in the body of the anonymous function.

```
 const arr:string[] = ['a', 'b', 'c'];
arr.forEach((value) => {this.doWork(value)})
```

> ***This does not mutate the array in any way.***

The every and some methods

The ***every*** and ***some*** method execute a function that returns a boolean on each element of the array and returns true if the passed function returns true for (every/some) of the elements in the array.

The ***every*** method:

- Returns true if the function returns true on all of the elements
- Returns false if the function is false on any single element

The ***some*** method:

- Returns true if the function returns true on any single element
- Returns false if the function returns false on all of the elements

```
 const ages= [21, 12, 19, 6, 15];
if (ages.some((age) => age > 18)){
    console.log('We have some adults');
}else{
    console.log('We have no adults');
}
if (ages.every((age) => age > 18)){
    console.log('We have all adults');
}
else{
    console.log('We have at least 1 kid');
}
```

> *These do not mutate the array in any way.*

The find and findIndex methods

The ***find*** method execute a function (Test method) that returns a boolean on each element of the array and returns the first element where the function returns true. The ***findIndex*** method returns the cardinal index of the element in the array instead.

find:

- Returns the first element where the test function returns true
- Returns undefined if the test function returns false on all elements

```
 interface Person {
    name: string;
    age: string;
}
const people: Person[] = [{ name: "John", age: "21" },{
name: "Jane", age: "22" }];
let jane = people.find((person) => person.name === "Jane");
if (jane !== undefined) console.log(`found ${jane.name}`);
```

findIndex:

- Returns the index of the first element where the test function returns true
- Returns -1 if the test function returns false on all elements

```
 interface Person {
    name: string;
    age: string;
}
const people: Person[] = [{ name: "John", age: "21" },{
name: "Jane", age: "22" }];
const index = people.findIndex((person) => person.name ===
"John");
if (index !== -1) console.log(`found ${people[index].name}
`);
```

There are also *last* versions of these methods that return the last element that matches. These are *findLast* and *findLastIndex*.

> *These do not mutate the array in any way.*

The filter method

The filter method executes a function (Test method) that returns a boolean on each element of the array. It returns a new array of the elements where the test function returns true.

```
 interface Person {
    name: string;
    age: number;
}

const people: Person[] = [
    { name: "John", age: 17 },
    { name: "Jane", age: 22 },
];
const adults = people.filter((person) => person.age > 18);
console.log(adults);
```

Filter returns an array with all of the elements (Jane in our example) where the function returns true.If the function returns false on all elements, it returns an empty array [].

Since it does not mutate the original array, you must capture the return value.

> *This does not mutate the array in any way.*

The map method

The map method executes a function that returns a new array consisting of the return values of the function applied to each element of the array.

```
 const ages:number[]=people.map((person) => person.age);
console.log(ages);  //outputs an array of ages.
```

In the example, the method is called on each person object, and returns the age of that person. The result is an array containing the ages of each person in the same order as the people in the original array.

It is not critical that the method USE the element of the array, suppose I wanted to create an array containing 0's for each element in our array.

```
 const zeros:number[]=people.map((person) => 0);
console.log(zeros)
```

Map is very useful for extracting data from an array of objects.

> *This does not mutate the array in any way.*

The flatMap method

The flatMap method executes a function and returns a new array consisting of the return values of the function applied to each element in a nested array.

```
 interface Person {
    name: string;
    groups: string[];
}
const people: Person[] = [
    { name: "John", groups: ["admin", "user"] },
    { name: "Jane", groups: ["editor"] },
];
let groups: string[] =
    people.flatMap((person) => person.groups);
console.log(groups);
```

In the example, the method is called on each person object, but the function returns an array which is then combined with the other arrays returned into a single array (merge). Here map would return `[[‘admin’,’user’],[‘editor’]`, but flatMap flattens it into `[‘admin’,’user’,’editor’]`

> *This does not mutate the array in any way.*

The reduce method

The ***reduce*** method takes a function of two parameters. The first is the array element and the second is an accumulator variable which gets passed to each function along with the array element.
The accumulator value is passed from one function call to the next allowing us to Reduce the array into a single value.
The ***reduce*** method returns a single value that is the accumulated result of all of the functions calls on each element of the array.

The ***reduce*** function ignores empty array elements.

```
 let vals: number[] = [1, 2, 3, 4, 5];
let sum = vals.reduce((acc, val) => acc + val, 0);
```

In the example we are summing up the numbers in an array by adding each number to acc. The initial value of acc is the second parameter to reduce.

Here is a product example (note that for this we set the initial value of accumulator to 1):

```
 let vals: number[] = [1, 2, 3, 4, 5];
let product= vals.reduce((acc, val) => acc * val, 1);
```

For something a little bit more interesting, we can compute some basic statistics on an array.
Note that we can do anything inside the function and any

changes we make to the accumulator parameter get passed on to the function call for the next element in the array.

```
 let vals=[1,2,3,4,5];
let max= vals.reduce((acc, val) => Math.max(acc, val), -
Infinity);
let min= vals.reduce((acc, val) => Math.min(acc, val),
Infinity);
let average = vals.reduce((acc, val) => acc + val, 0) /
vals.length;
let stdev = Math.sqrt(
    vals.reduce((acc, val) => {
        return acc + (val - average) ** 2;
    }, 0) / vals.length);
```

> *Notice that without the braces {} the value is returned automatically by the anonymous function (as in min, max, and average above), but with the braces I must explicitly call return (as in stdev above). This is true of all anonymous functions.*

We can exclude some values from our count, and also map some values first. Here we are summing up the odd integers in our array vals.

```
 let vals=[1,2,3,4,5];
sumOdd=vals.reduce((acc,val)=>{
    if (val%2)return acc+val
    else return acc;
},0);
```

Even though we are supposed to return a single value, that value can be a complex object. Here we compute all the statistics in a single pass through the array.

```
 interface Stats {
    max: number;
    min: number;
    average: number;
    stdev: number;
}
let stats: Stats = vals.reduce(
    (acc, val) => {
        return {
            max: Math.max(acc.max, val),
            min: Math.min(acc.min, val),
            average: acc.average + val,
            stdev: acc.stdev + (val - acc.average) ** 2,
        };
    },
    {
        max: -Infinity,
        min: Infinity,
        average: 0,
        stdev: 0,
    },
);
```

We can even use it to combine map and filter in a single step.

```
 let vals=[1,2,3,4,5];
let OddSqrs = vals.reduce((acc: number[], val: number) => {
    if (val%2) return [...acc, val * val];
    else return […acc];
}, []);
```

This example creates an array of the squares of the odd numbers in the array.

Consider the following array:

1 | 4 | 11 | 7

We will use this function to sum the array.

```
let sum = vals.reduce((acc, val) => acc + val, 0);
```

The first parameter is our function which takes the accumulator variable and a variable to receive each element of our array. The second parameter is the initial value of the accumulator.

Let’s step through the operation of this to make sure we understand what is happening.

On the first call (element with value 1) to our function, the values of the parameters and return value are:

Acc | 0 |
Val | 1 |
Returns | 1 |

On the second call, the return value of the first call becomes the value of accumulator.

Acc | 1 |
Val | 4 |
Returns | 5 |

On the third call, the return value of the second call becomes the value of accumulator.

Acc | 5 |
Val | 11 |
Returns | 16 |

On the fourth call, the return value of the third call becomes the value of the accumulator.

Acc | 16 |
Val | 7 |
Returns | 23 |

Since we have made calls on each element of the array we are done, and ***reduce*** returns the value returned by the last function call (23 in our example).

There is a variant of the ***reduce*** method that traverses the array in reverse order (i.e. right to left instead of left to right). This method is ***reduceRight***.

```
let sum = vals.reduceRight((acc, val) => acc + val, 0);
```

Obviously, for the examples so far, this makes no difference (sum and product are communative), but there are cases wehre it would.

Consider the following:

```
 let vals=[1,2,3,4,5];
firstEven=vals.reduce((acc:number,val)=>{
    if (val%2===0 && acc===0)return val
    else return acc
},0);
```

This returns the first even number. If we use reduceRight instead of reduce, it would return the last even number in the list.

```
 let vals=[1,2,3,4,5];
lastEven=vals.reduceRight((acc:number,val)=>{
    if (val%2===0 && acc===0)return val
    else return acc
},0);
```

> *This does not mutate the array in any way.*

The sort method

With no arguments, ***sort*** returns the elments in the array in ascending or alphabetical order.

```
 let vals=[1,2,3,4,5];
vals.sort();
console.log(vals);
```

If we provide a comparison function, we can define the sort order. The function should return:

- positive if the first value comes second in the sort order.
- negative if the first value comes after the second value.
- 0 if the values are the same.

```
 let vals=[1,3,2,6,5,4];
ascending=vals.sort((a,b)=>a-b);
descending=vals.sort((a,b)=>b-a);
```

Since we pass a function, we can sort arrays of complex objects or classes in any way we wish.

> *NOTE: This method is destructive and overwrites the array. If you don’t want this to happen, you have to clone the array first.*

Summary

High order methods are methods where we pass a function as an argument, or return a function. Specifically, we examined a number of ***high order methods*** for working with arrays of objects. These methods provide convenient, concise, and clear ways to handle various tasks which we might wish to accomplish on arrays.

13) Recursion

13.1) Description and Definition of Recursion

***Recursion** is a method in Computer Science where we state a problem in terms of a smaller instance of that problem, then write a function which calls itself to solve the smaller version of the problem.

Stating a problem recursively

In general, Recursion involves stating a large problem in terms of a smaller version of the same problem.

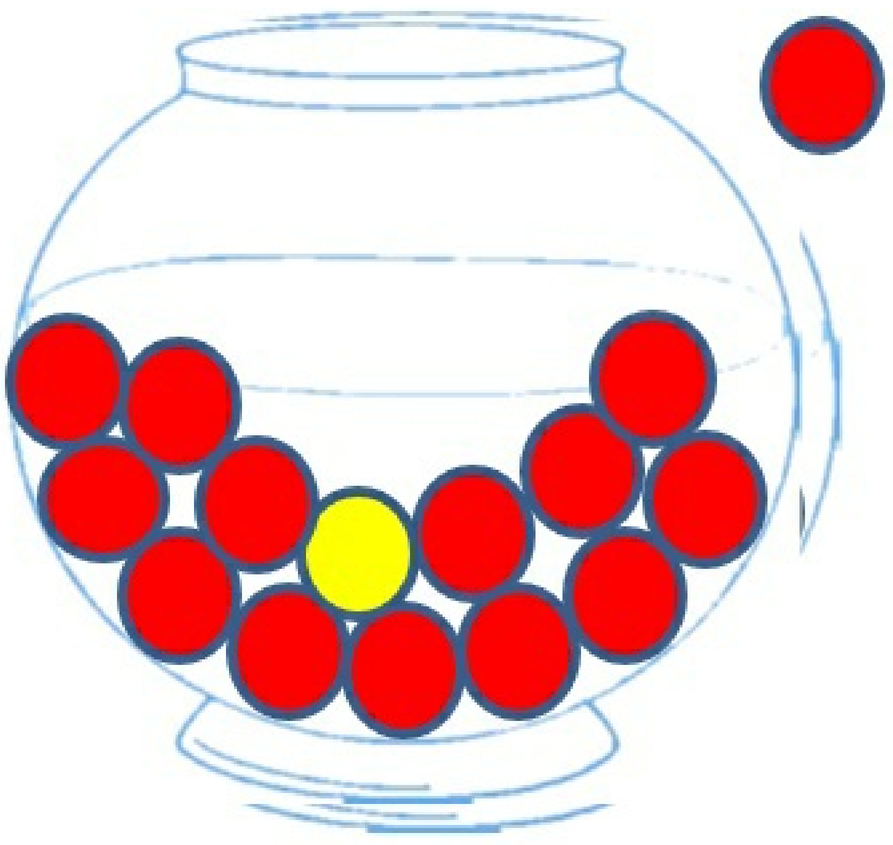
Consider the problem of teaching all of you a concept.
We can restate this problem as

- Teach one student
- Teach the rest of the students (this is a smaller group with 1 less student)

Eventually there are no more students to teach and we are done.
That is the basic idea behind recursion

Consider a problem where we have a container of balls that are all colored either red or yellow. If we want to know if there are any yellow balls in the container, we can state this problem recursively.

- Checking one ball is easy, so if we remove a ball we know if it is yellow or not.
    - If the ball is yellow, we are done, so return true.
    - If the ball is red, we can remove the ball from the set
        - Now we have the same problem, only there are fewer balls to look at.



Eventually, we will find a yellow ball, or we will empty the container making the problem trivial. *Are there any yellow balls in the empty container?*

Recursion Terminology

When we find a yellow ball we are done. The answer is “yes” there is a yellow ball. When the container is empty we are done. The answer is “no” because there are clearly no yellow balls in the empty container, and while making it empty, we didn't see any. These cases where the answer is trivial are know as ***stop conditions*** or the ***base case*** of the recursion.

So how do we get there? We need to make sure that whatever we do when the stop conditions are not met *approaches* the stop condition. If we keep removing balls one at a time, either we find the yellow ball or we reduce the number of balls by one. Clearly in all cases, this approaches the stop conditions of finding a yellow ball or emptying the container. The step that handles non-stop conditions and approaches the stop conditions is referred to as the ***recursive step***.

```
 public balls:string[] = [“red”, “red”, “red”, “red”
,”yellow”, “red”, “red”];
function findYellowBall(container:string[]):boolean{

    if (container.length==0){  //Yellow ball not found if
balls array is empty
        return false;
    }
    else if (container[0]===(“yellow”)){  //Yellow ball
found
        return true;
    }

else  {  //Yellow ball might still be in the container, but
not in the first element
        return findYellowBall(container.slice(1));
    }
}
```

The highlighted section above is the ***stop condition***. We first check if the array is empty, then we check if the first ball in the array is yellow. If either is true we are done and we know the answer (false/true respectively).

```
 public balls:string[] = [“red”, “red”, “red”, “red”
,”yellow”, “red”, “red”];
function findYellowBall(container:string[]):boolean{
    if (container.length==0){  //Yellow ball not found if
balls array is empty
        return false;
    }
    else if (container[0]===(“yellow”)){  //Yellow ball
found
        return true;
    }
    else  {  //Yellow ball might still be in the container,
but not in the first element
        return findYellowBall(container.slice(1));
    }
}
```

The highlighted section above is the ***recursive step***. Since we know it is not the first element, we simply reinitiate our search on the rest of the array (elements 2…n) by slicing the array and passing the result to our function.

Recursion Rules

- A recursive algorithm must have a base case or stop condition
- A recursive algorithm must change state and move towards the base case
- A recursive algorithm must call itself recursively

A simple Example

Consider the porblem of computing factorial.

Factorial is defined as: *n! = n * (n-1) * (n-2) * (n-3) * ... * 1*

```
5! = 5 * 4 * 3 * 2 * 1
```

We can restate this in terms of an easier instance of factorial: *n! = n * (n - 1)!*

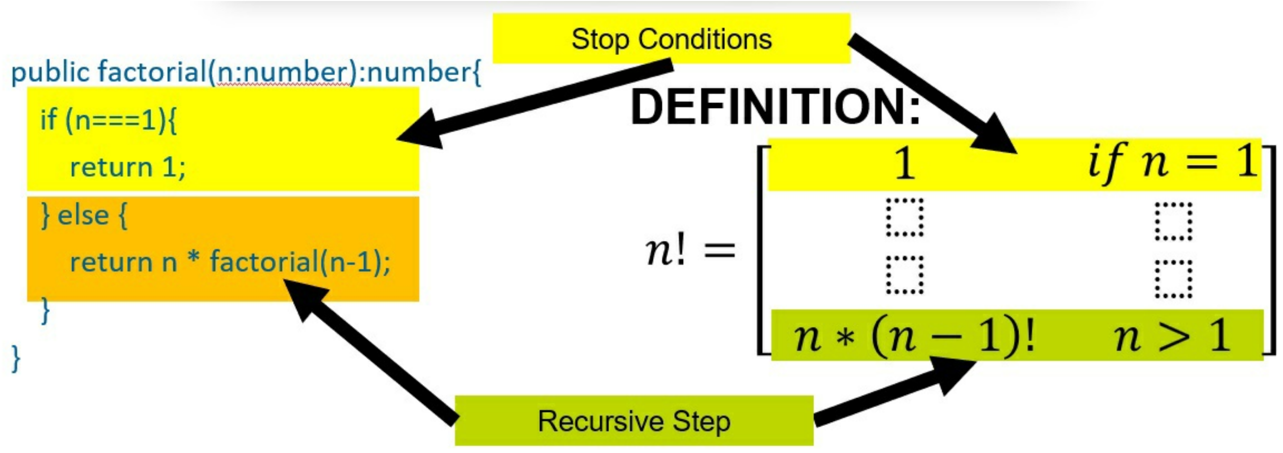
```
5! = 5 * 4!
```

Since we know that 1! is equal to 1, we can rewrite the definition as:

$$n! = \begin{bmatrix} 1 & if\ n = 1 \\ n * (n-1)! & n > 1 \end{bmatrix}$$

> *This is a recursive definition. It has a stop condition (n === 1), and a recursive step (n*(n-1)!)*

So how do we code this:



```
public factorial(n:number):number{
  if (n===1){
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

$$n! = \begin{bmatrix} 1 & if\ n = 1 \\ \vdots & \vdots \\ n * (n-1)! & n > 1 \end{bmatrix}$$

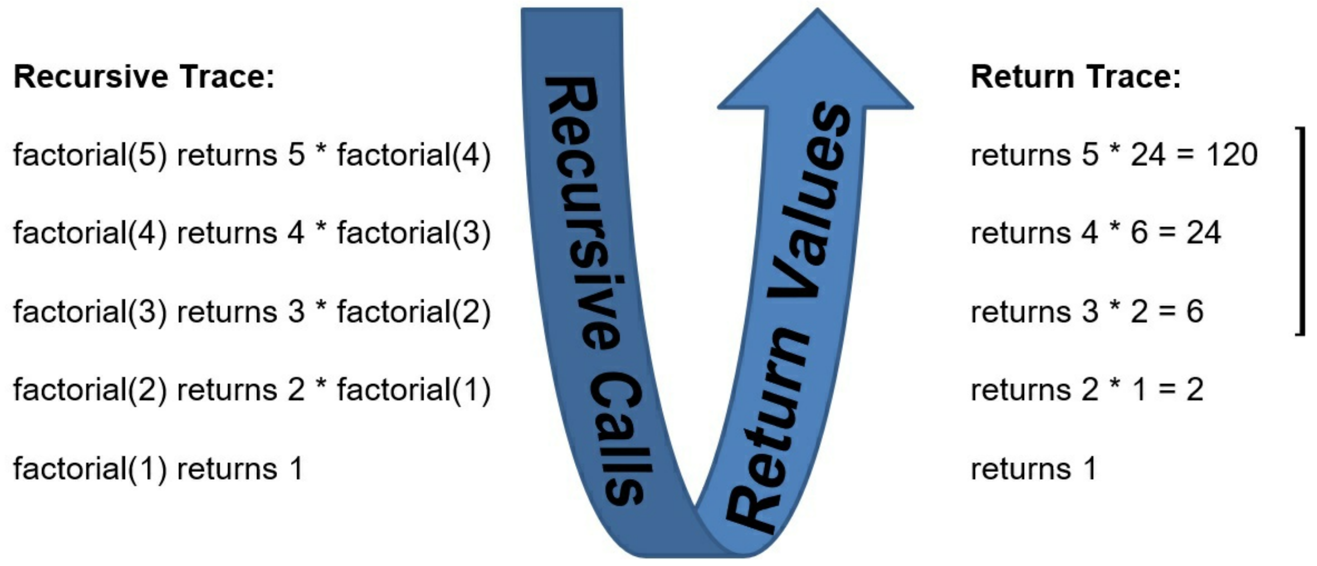
Let’s try it:

```
 factorial(n:number):number{
     if (n===1){
         return 1;
     } else {
         return n * factorial(n-1);
     }
}
console.log(factorial(5));
```

So what is actually happening:

- factorial(5) returns 5 * factorial(4)
- factorial(4) returns 4 * factorial(3)
- Factorial(3) returns 3 * factorial(2)
- factorial(2) returns 2 * factorial(1)
- factorial(1) returns 1

This process leads to the answer being computed during each return from the base case to the original function call.



But why?

In the jar of marbles and factorial examples, we could very easily solve these problems without recursion. A simple loop would be sufficient. While this is true of most/all problems, there are problems that are considerably easier to deal with by using recursion.

Let's look at a simple example of binary search.

- In binary search, we start with a sorted list. Instead of checking every element, we check the middle element.
- Since the list is sorted, if the value is less than the middle element, then we don't have to search the second half of the list. If it is greater, than we don't have to search the

first half.

Consider:

Find 4 in [1,2,3,4,5,6,7,8,9]

The middle element is 5, and since 4 is less, we can restrict further searches to [1,2,3,4]
In other words, if the middle element is not what we want, then we have reduced the problem to searching half the list. If it is what we want, then we are done.

Eventually the list will have 0 or 1 elements in it.

- If 1, it is either what we are looking for or not.
- If 0, then we did not find what we were looking for.

So to complete the example on the array [1,2,3,4,5,6,7,8,9] trying to find 4.

- 4 < 5 so we only search for 4 in [1,2,3,4]
- The middle element is either 2 or 3, os if we pick 3 4>3 so we search for 4 in [4]
- The list contians 1 element, and that element is the 4 we are looking for.

Another example, search for 11 in the same array.

- 11 > 5 so search for 11 in [6,7,8,9]

- 11 > 8 so search for 11 in [9]
- 11 != 9 so we did not find it.

> *This is a lot faster than searching every element one at a time.*

In the case of binary search our ***stop condition*** is:

- We stop when there are no elements in the list and return false
- We stop when the middle element is the one we are looking for and return true

Our ***recursive step***

- If the search value is greater than the middle value, we search the second half of the list
- If the search value is less than the middle value, we search the first half of the list

> *Each time, we are searching a smaller list, so eventually we will find what we want or the list will be empty.*

Our stop condition in code:

```
 if (list.length === 0){ return -1; }
let mid=Math.floor((list.length - 1) / 2);
if (list[mid] === target){ return list[mid]; }
```

Our recursive step simply calls itself on the correct half of the array:

```
 else if (list[mid] < target){
    return binSearch(list.slice(mid + 1), target);
}
else{
    return binSearch(list.slice(0, mid), target);
}
```

And the full search function:

```
 function binSearch(list: number[], target: number){
    if (list.length === 0){
        return -1;
    }
    let mid = Math.floor((list.length-1) / 2);
    if (list[mid] === target){
        return list[mid];
    }
    else if (list[mid] < target){
        return binSearch(list.slice(mid + 1), target);
    }
    else{
        return binSearch(list.slice(0, mid), target);
    }
}
let list=[1,2,3,4,5,6,7,8,9];
console.log(binSearch(list, 9));
console.log(binSearch(list, 11));
```

Summary

Recursion is a programming technique where a problem is restated in terms of a smaller instance of the same problem. Recursive functions must have a ***stop condition*** when the problem is solved or when the smaller instance becomes trivial. They must also have a ***recursive step*** where they call the same function on a smaller instance of the problem.

13.2) Trees

A ***tree*** in Computer Science is a data structure that represents data in a parent/child relationship.

Motivating Recursion

All of these would have been very easy to implement using a loop instead. One place where recursion is particularly useful is ***Trees***.
Trees are a basic data structure that we can use to represent data in a parent child relationship.

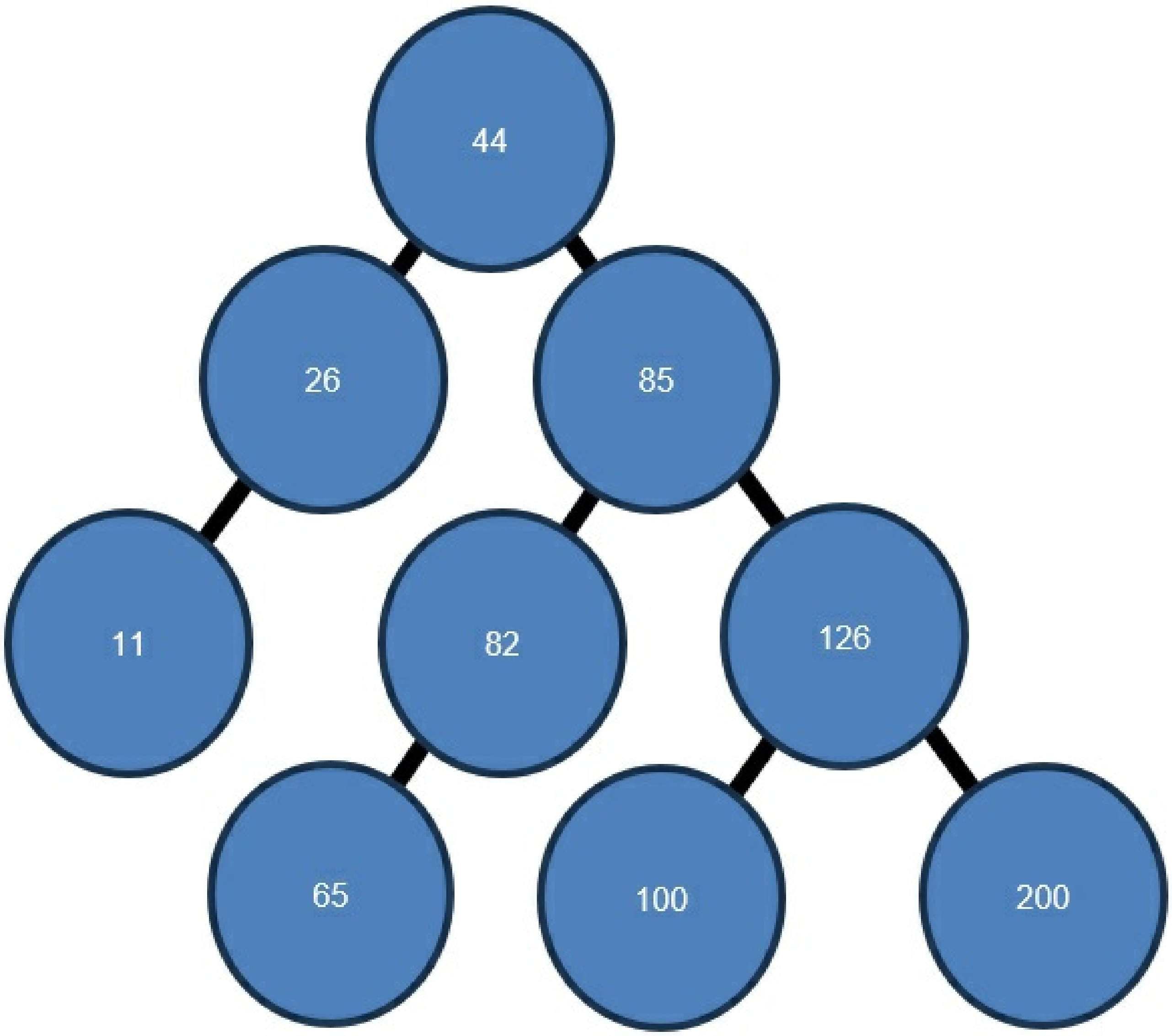
Many problems can be modeled as a tree. As a matter of fact HTML is actually a tree representation since a parent element can have multiple child elements.

Binary Search Trees

Consider a tree of numbers instead. If we look at the children of any node in the tree, they are themselves a tree. Just a smaller one.

Even for the nodes without children we can think of them as trees with no children (empty trees).
So in other words, we can treat each sub-tree of a tree with

a given root as if it were a tree.
This feels like a good candidate for recursion.



As it turns out, this is a special kind of tree called a ***binary search tree***.

It has some specific properties:

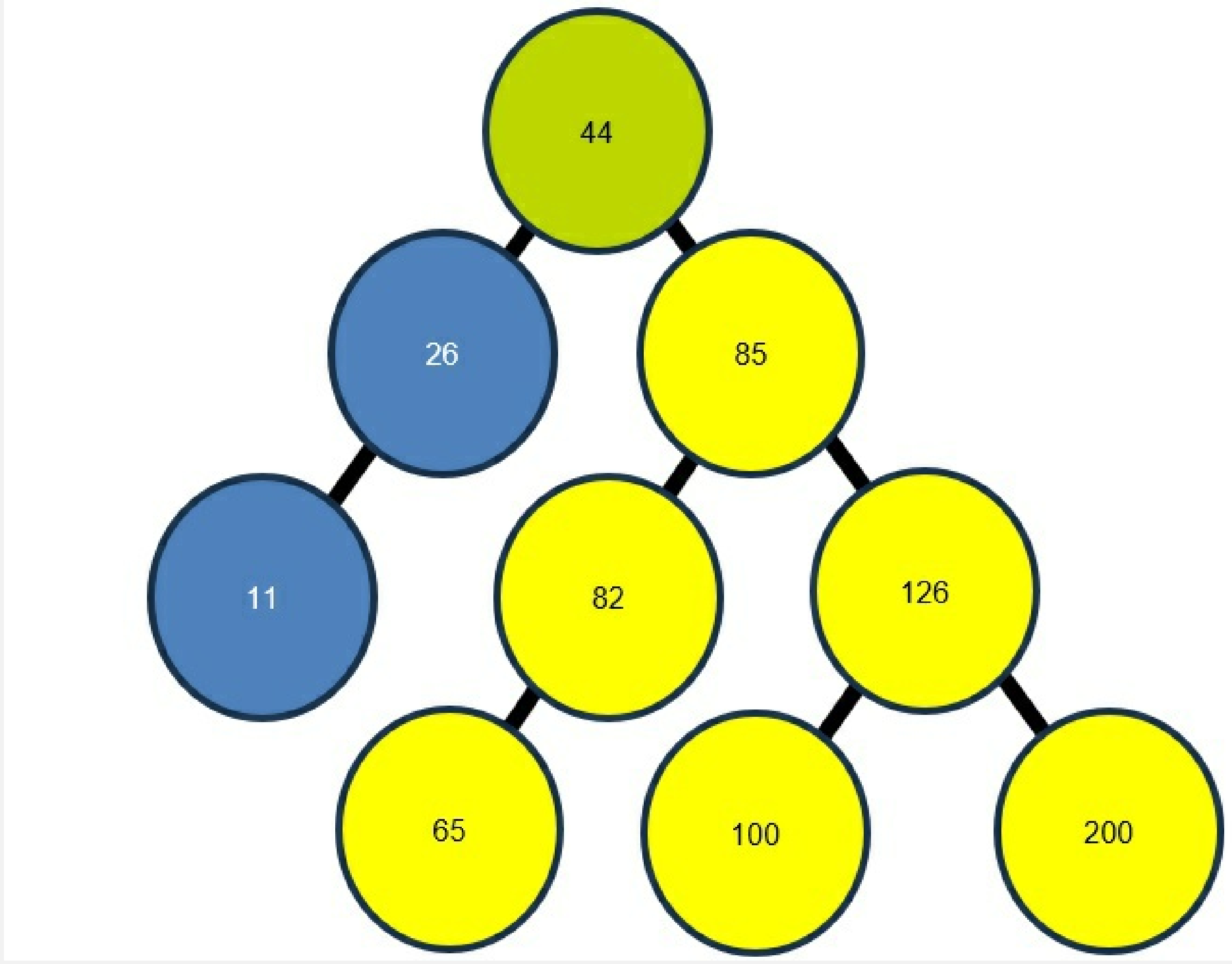
- A node will have 2 subtrees (possibly empty)
- Every number in the left sub-tree must be less than the value stored in the node
- Every number in the right sub-tree must be greater than the value stored in the node
- These must hold for the subtree rooted at every node in the tree.

We can search this structure by examining the root node then recursively searching the correct subtree based on the values.

Say we have a treeSearch method

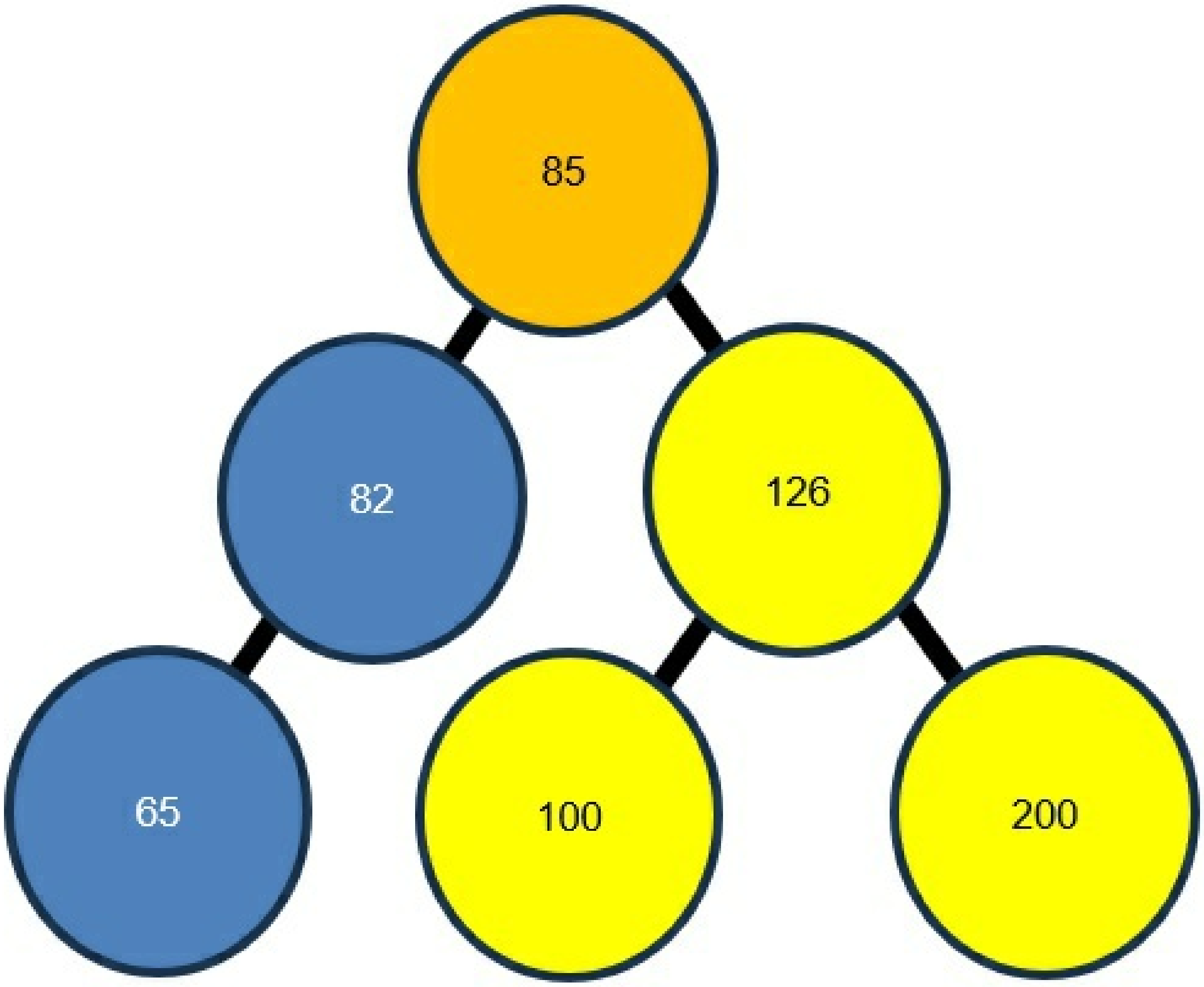
```
function treeSearch(tree: TreeNode, target: number){
```

and we want to find the number 100 in the tree. We woudl start by comapring it to 44. Since 100 > 44 we know the answer must be in the right subtree if this is a ***binary search tree***.



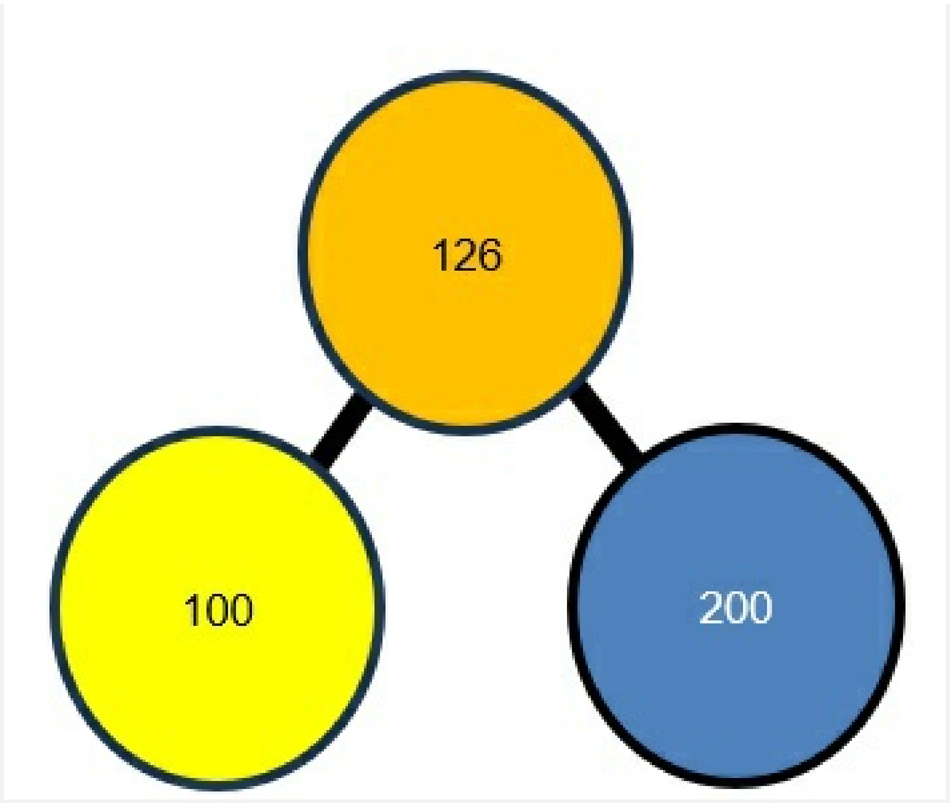
Root is in orange, subtree to search is in yellow.

We now call our treeSearch function on just the right subtree. Now we compare 85 to 100.
Again, 85 < 100 so we again search the right subtree.



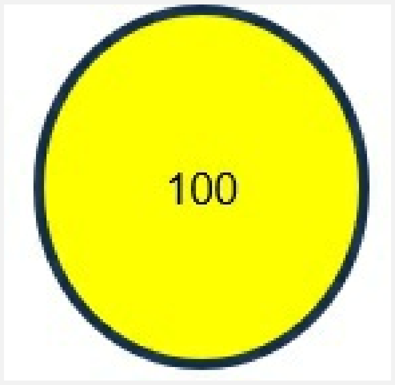
Root is in orange, subtree to search is in yellow.

This time, we see that the value 100 < 126, so we will call treeSearch on the left sub-tree.



Root is in orange, subtree to search is in yellow.

This time when we call on the left subtree, our root is the value we are looking for, so we can return that we found it.



Root is in orange, subtrees are empty.

> *Note: if we had been looking for 99, we would search the left subtree, which would be empty and we would return tat we did not find it.*

Implementing Binary Search Trees in Typescript

First we need a way to represent a tree in typescript. Since every node in a tree is itself a tree root, we should implement the node of a tree, then just keep a reference to that node as the root of the tree.

We will create a node class that will contain a number and 2 children. Those children themselves will be nodes (possibly empty).

```
 export class TreeNode{
    left: TreeNode|undefined=undefined;
    right: TreeNode|undefined=undefined;
    constructor(public value: number){}
}
```

Searching Binary Search Trees

Now we can right our treeSearch function to recursively search our tree.

Our function should have ***stop conditions*** when the tree is empty or when the value in the root of the tree is the one we are looking for.

```
 if (tree === undefined) {           //tree is emtpy
    return false;
} else if (tree.value === value) {      //root of tree is
our value
    return true;
}
```

For the ***recursive step***, we want to search either the left or right subtree based on whether value is less than or greater than the node’s value (we already checked === in our stop conditions)
When we look at a node, there are only 4 possibilities.

- The node is empty (undefined)
- It is the node we are looking for
- It is > than the node we are looking for
- It is < than the node we are looking for.

If the node is empty (undefined) then the node with the value we are looking for can’t exist, so we return false (did not find it).

If the node’s value is === the value we are looking for, then we return true (found it).

If the node’s value is < the value we are looking for, then if the value is in the tree, it must be in the right sub-tree, so we call treeSearch recursively to search that sub-tree.

If the node’s value is > the value we are looking for, then if the value is in the tree, it must be in the left sub-tree, so we call treeSearch recursively to search that sub-tree

```
 function treeSearch(tree: TreeNode | undefined, value:
number): boolean {
    if (tree === undefined) {
        return false;
    } else if (tree.value === value) {
        return true;
    } else if (tree.value<value){
        return treeSearch(tree.right, value);
    } else {   //must be > value, it is the only posibility
left
        return treeSearch(tree.left, value);
    }
}
```

Inserting into Binary Search Trees

We can also recursively insert a node into the tree. Search the tree until you find a node where the subtree you would search next is undefined and add a new node there. This is our stop condition. If the sub-tree we would insert into is not empty, then we just insert into that (smaller) sub-tree.

```
 function insert(tree: TreeNode, value: number): void {
    if (value===tree.value) return;
    else if (value < tree.value) {
        if (tree.left === undefined) {
            tree.left = new TreeNode(value);
        } else {
            insert(tree.left, value);
        }
    } else {
        if (tree.right === undefined) {
            tree.right = new TreeNode(value);
        }else{
            insert(tree.right, value);

        }
    }
}
```

Trees are a common data structure in Computer Science and recursion is a much more natural way to deal with them.

Complete Tree Example

```
 export class TreeNode{
    left: TreeNode|undefined=undefined;
    right: TreeNode|undefined=undefined;
    constructor(public value: number){}
};
function treeSearch(tree: TreeNode | undefined, value:
number): boolean {
    if (tree === undefined) {
        return false;
    } else if (tree.value === value) {
        return true;
    } else if (tree.value<value){
        return treeSearch(tree.right, value);
    } else {   //must be > value, it is the only posibility
left
        return treeSearch(tree.left, value);
    }
}
function insert(tree: TreeNode, value: number): void {
    if (value===tree.value) return;
    else if (value < tree.value) {
        if (tree.left === undefined) {
            tree.left = new TreeNode(value);
        } else {
            insert(tree.left, value);
        }
    } else {
        if (tree.right === undefined) {
            tree.right = new TreeNode(value);
        }else{
            insert(tree.right, value);
        }
    }
}
```

```
let treeRoot = new TreeNode(44);
insert(treeRoot,26);
insert (treeRoot,11);
insert(treeRoot,85)
insert(treeRoot,82)
insert(treeRoot,126)
insert(treeRoot,100)
insert(treeRoot, 200);
insert(treeRoot, 65);
console.log(treeSearch(treeRoot,100));
console.log(treeRoot);
```

THis example implements the ***binary search tree*** in the previous example, then searches it for 100.

> *If I inserted them in a different order, I would have gotten a different tree.*

Thought question: What happens if I insert them in sorted order?

▶
Answer

An Object Oriented Tree

This is nice, but it is NOT very object oriented.

A tree node should encapsulate the things we can do to a tree so we won’t need external methods.
For our implementation of insert, it is pretty straight forward.
We just remove the tree parameter, and instead call the member method on the appropriate subtree which is not null since we already checked that.

```
 class TreeNode{
    ...

    insert(value: number): void {
        if (value === this.value) return;
        else if (value < this.value) {
            if (this.left === undefined) {
                this.left = new TreeNode(value);
            } else {
                this.left.insert(value);
            }
        } else {
            if (this.right === undefined) {
                this.right = new TreeNode(value);
            } else {
                this.right.insert(value);
            }
        }
    }

    ...

}
```

For the search method, it is a little less straight forward. We need to check for a null subtree before we make the recursive call instead of stopping when the tree is null (otherwise we will not have an object to call search on).

```
    treeSearch(value: number): boolean {
        if (this.value === value) {
            return true;
        } else if (this.value < value) {
            if (this.right === undefined) {
                return false;
            } else {
                return this.right.treeSearch(value);
            }
        } else {
            if (this.left === undefined) {
                return false;
            } else {
                return this.left.treeSearch(value);
            }
        }
    }
```

Now we stop in the parent node if the child node is undefined instead of stopping in the child when it is itself undefined.

Here is a complete example of our tree proram:

```
export class TreeNode {
    left: TreeNode | undefined = undefined;
    right: TreeNode | undefined = undefined;
    constructor(public value: number) {}
    insert(value: number): void {
        if (value === this.value) return;
        else if (value < this.value) {
            if (this.left === undefined) {
                this.left = new TreeNode(value);
            } else {
                this.left.insert(value);
            }
        } else {
            if (this.right === undefined) {
                this.right = new TreeNode(value);
            } else {
                this.right.insert(value);
            }
        }
    }
    treeSearch(value: number): boolean {
        if (this.value === value) {
            return true;
        } else if (this.value < value) {
            if (this.right === undefined) {
                return false;
            } else {
                return this.right.treeSearch(value);
            }
        } else {
            if (this.left === undefined) {
                return false;
            } else {
                return this.left.treeSearch(value);
            }
        }
    }
```

```
}
let treeRoot = new TreeNode(44);
treeRoot.insert(26);
treeRoot.insert(11);
treeRoot.insert(85);
treeRoot.insert(82);
treeRoot.insert(126);
treeRoot.insert(100);
treeRoot.insert(200);
treeRoot.insert(65);
console.log(treeRoot.treeSearch(100));
console.log(treeRoot);
```

Summary

Trees are an important data structure in Computer Science. They allow us to store data in a structured way that represeents parent/child relationships. In other words, a parent can have many children, but a child can only have one parent. A ***binary tree*** is a tree where each node has at most two children. A special case of a ***binary tree*** is a ***binary search tree***. In a binary search tree each node in the left subtree of all nodes must be of lower value than the root, and each node in the right subtree of all nodes must be greater than the value of the root.