# Introduction

This book will cover Object-oriented programming, introduce the basics of web development, and reinforce basic concepts of computer science. In order to use the samples in this book and the accompanying materials, you will need to have your environment set up correctly.

## In this introduction, we are going to introduce the organization of this textbook and help you set up your development environment.Overview

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.

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 used 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.

## Next Step

Let’s get your environment set up.

## Overview

This section helps you set up a working environment for this text. Through this process you will install and verify your development environment.

If everything goes well, this should only take you about 20 minutes. However, 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 complicated, so it is critical that your environment’s setup and the eventual structure of our web application follow these instructions precisely. If this seems limiting, that’s the point. Please try to stay within the bounds given 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).

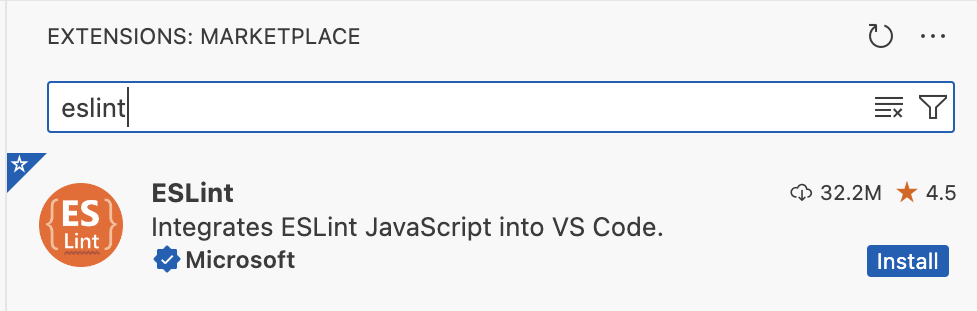
VS Code is an IDE (Integrated Development Environment) that you will program in.

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

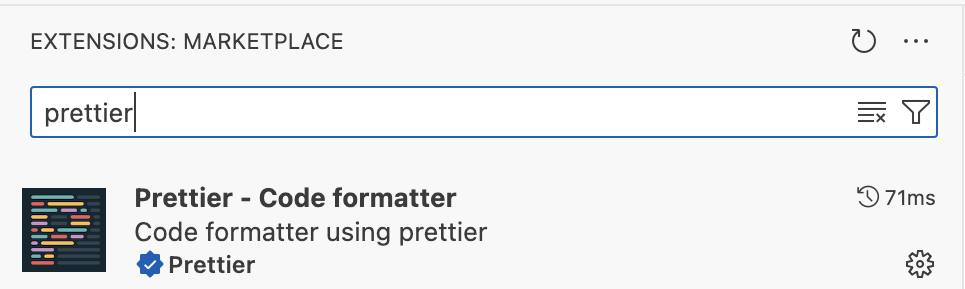
To open the extension menu, you can type Ctrl+Shift+X (Windows) or Cmd+Shift+X (Mac). There is also a navigation bar on the left side of your screen, and you can click the extensions menu that looks like this:

[INSERT IMAGE - I don’t think this one copied correctly]

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



Then, search for Prettier and click install.



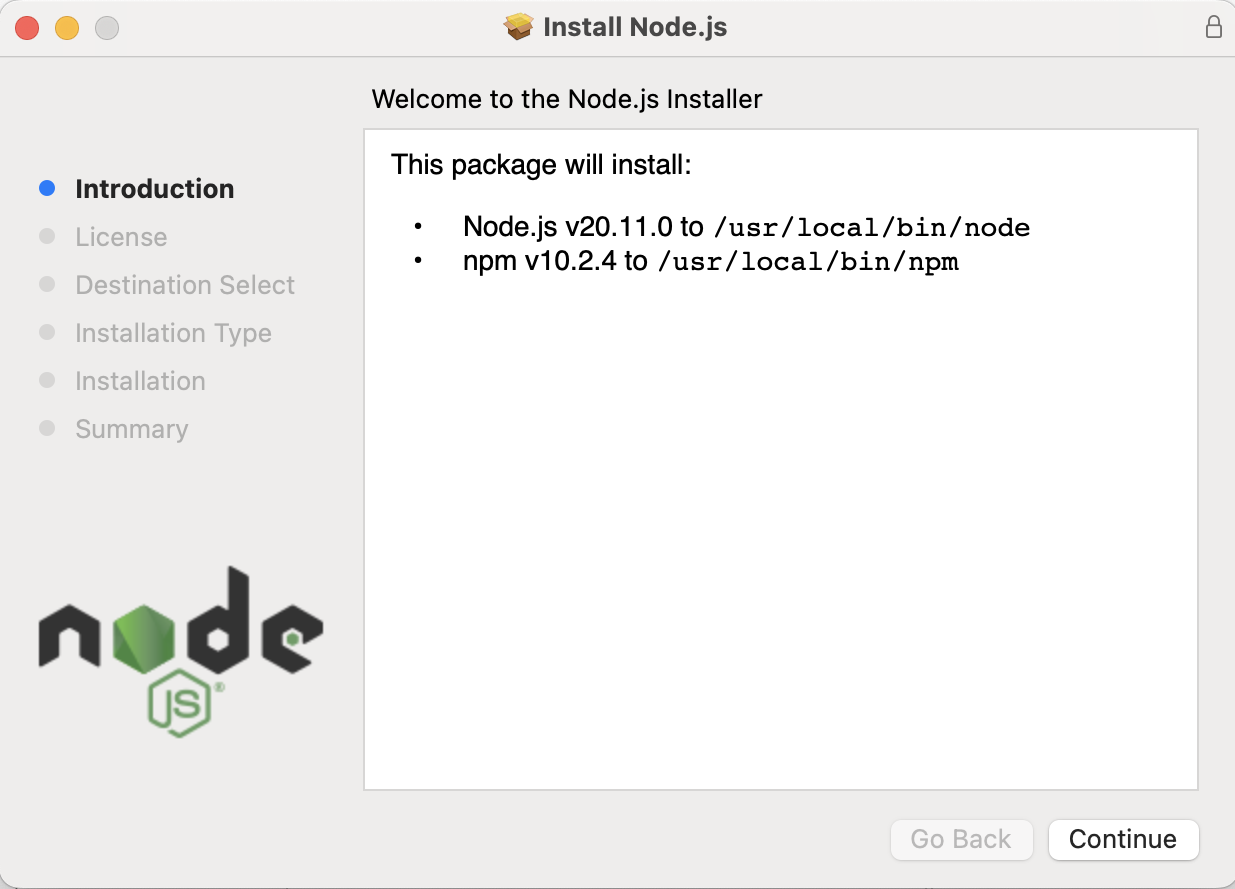
Make sure you have installed the extensions that are in the images above. These are the correct versions!

### Get Node

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

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.

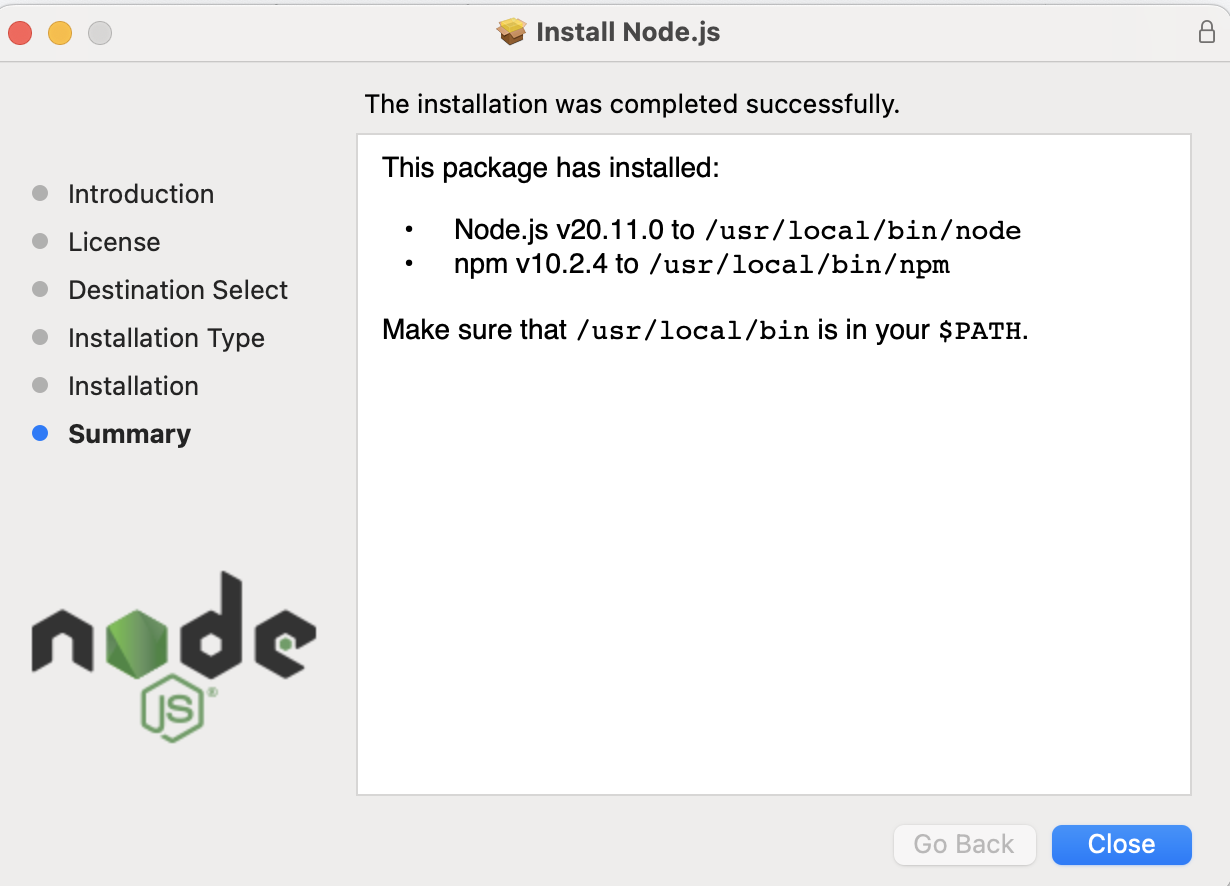


You may notice that it is installing both Node and something called npm. Node Package Manager (npm) will make it easier to manage, install, and update node packages. You need both of these.

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

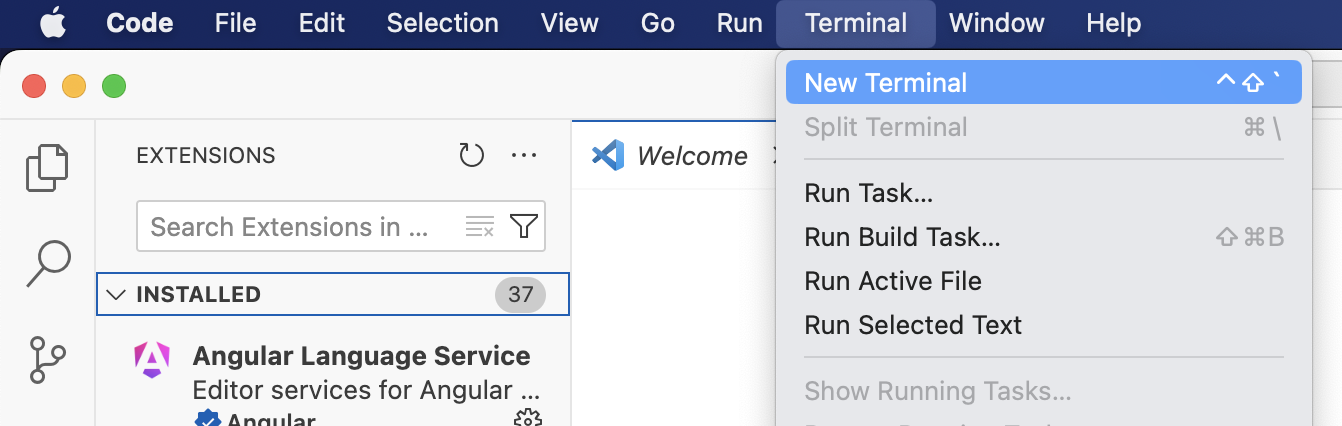
* For Windows: C:\Program Files\nodejs
* For Mac: /usr/local/bin/node

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

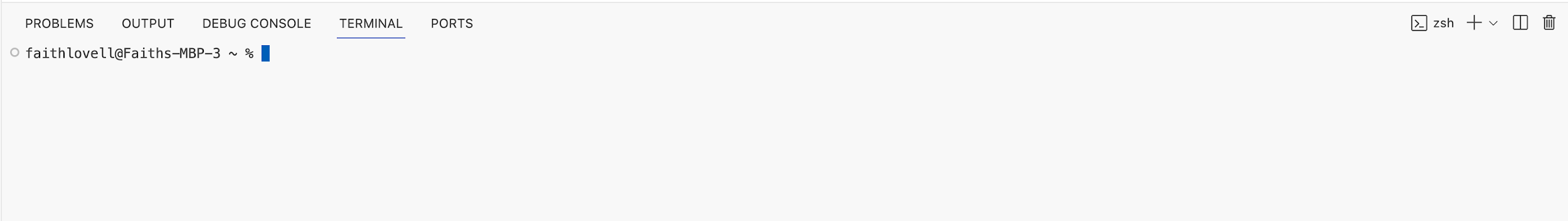


Now we need to verify that the installation was successful. Navigate back to VS Code and open a terminal.

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:



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 you 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):

Note: On some Windows systems, security prevents the running of node from powershell. If this is the case you can open a *command prompt* terminal window instead.

$ 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.

### 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:

[INSERT CODE BLOCK]

When it asks for a 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 for words to appear, and then, when able, type:

[INSERT CODE BLOCK]

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) (if you don’t already have one).

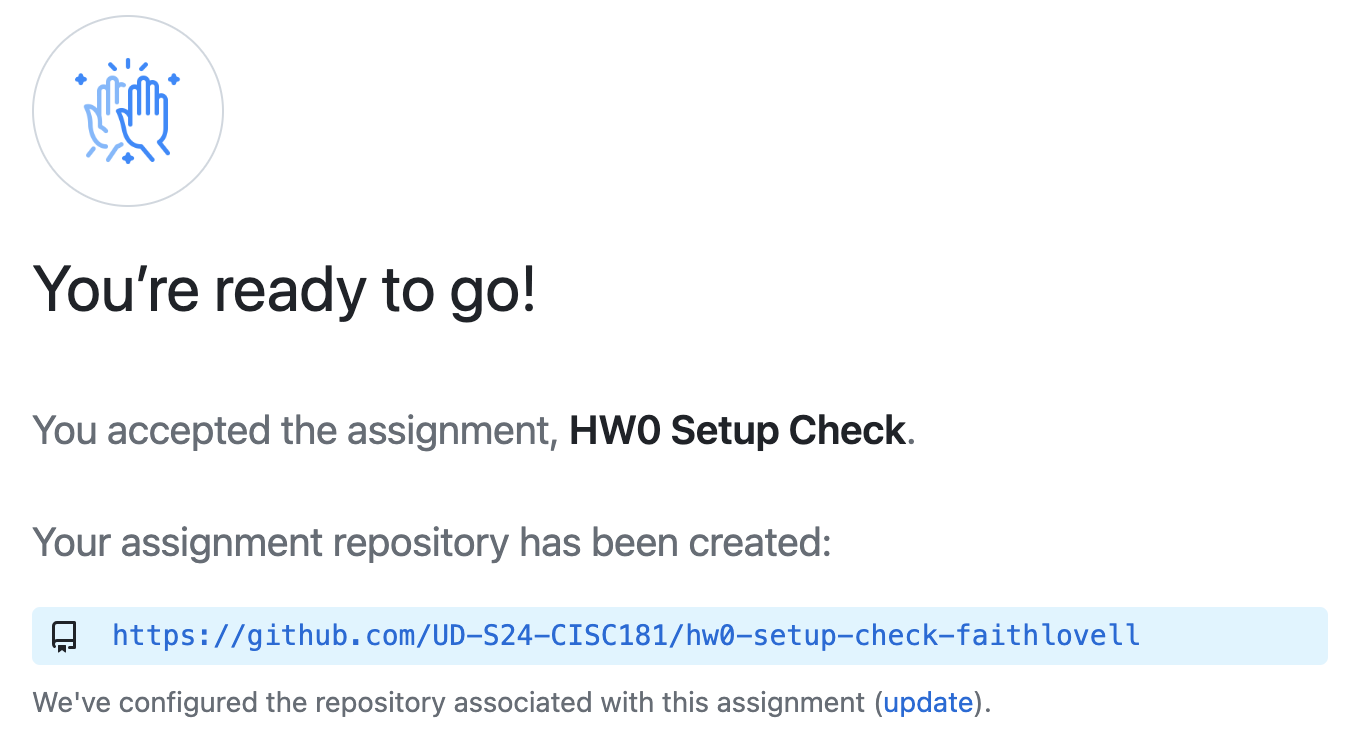
This account is an important part of your professional identity. You will use it to store your code, collaborate with others, and show off your work to potential employers. Choose a username that is professional and easy to remember. You are likely to use this account for a long time, so choose wisely!

## Clone Assignment

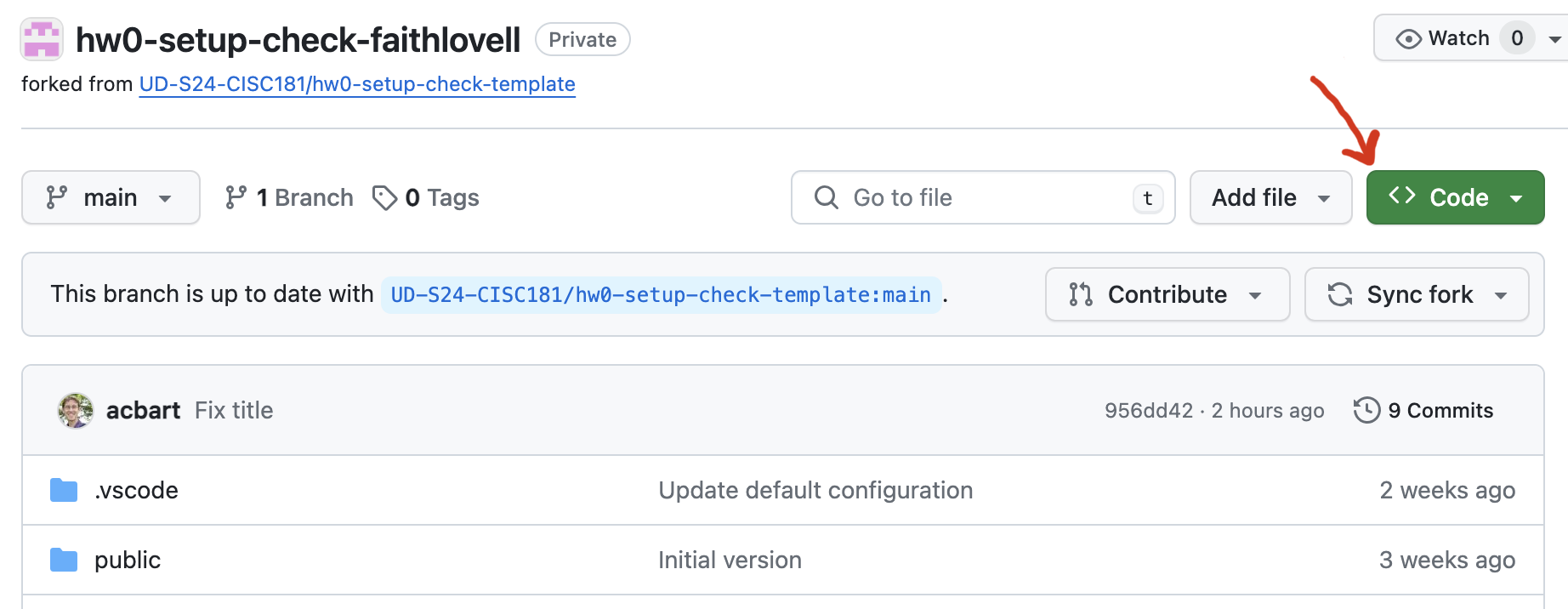
Now that you have a programming environment, it is time to complete the first coding exercise to test everything.

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.

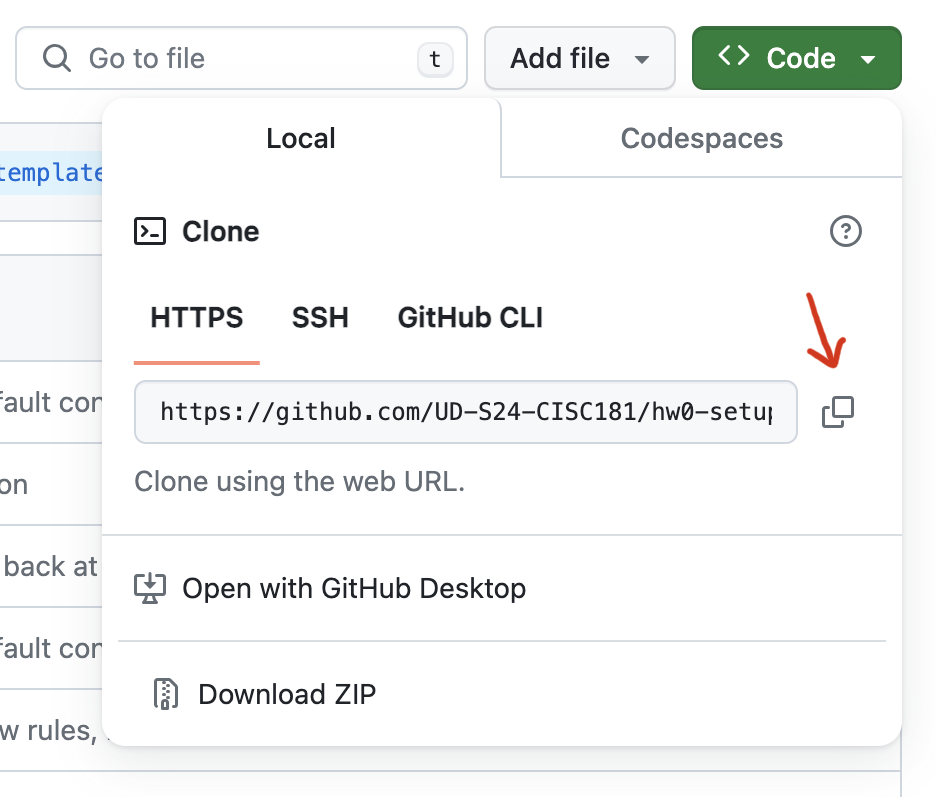
You may need to reload the page manually. The process should not take long. When the repository is ready, you should see a new link:



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



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.

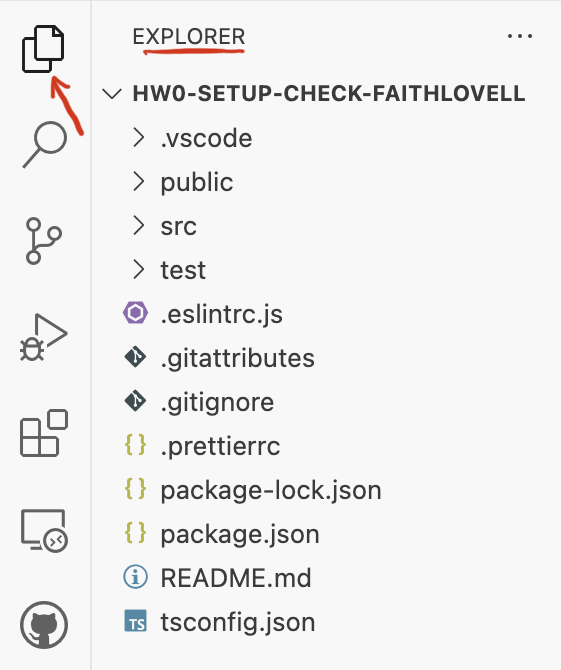
You will need to run the “Git: Clone” command in VS Code:

* 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.
* If you are asked to authenticate on GitHub, do so.
* A folder select window will prompt you to “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.

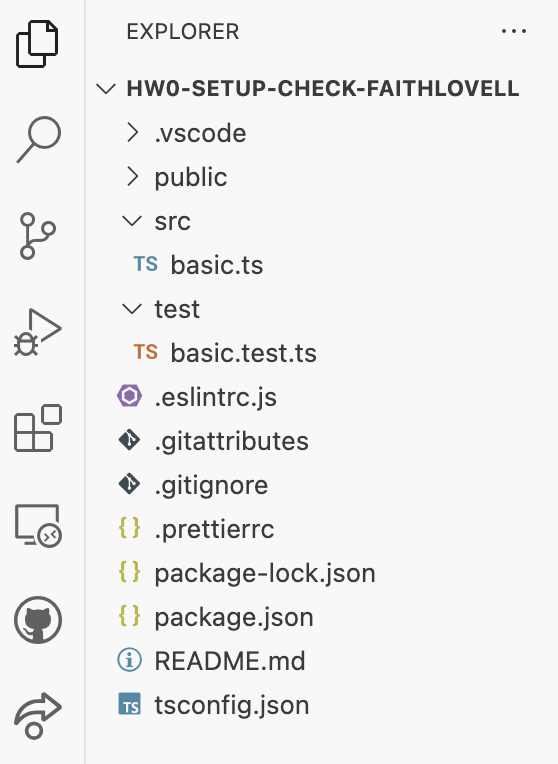
## Inspecting the Project

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

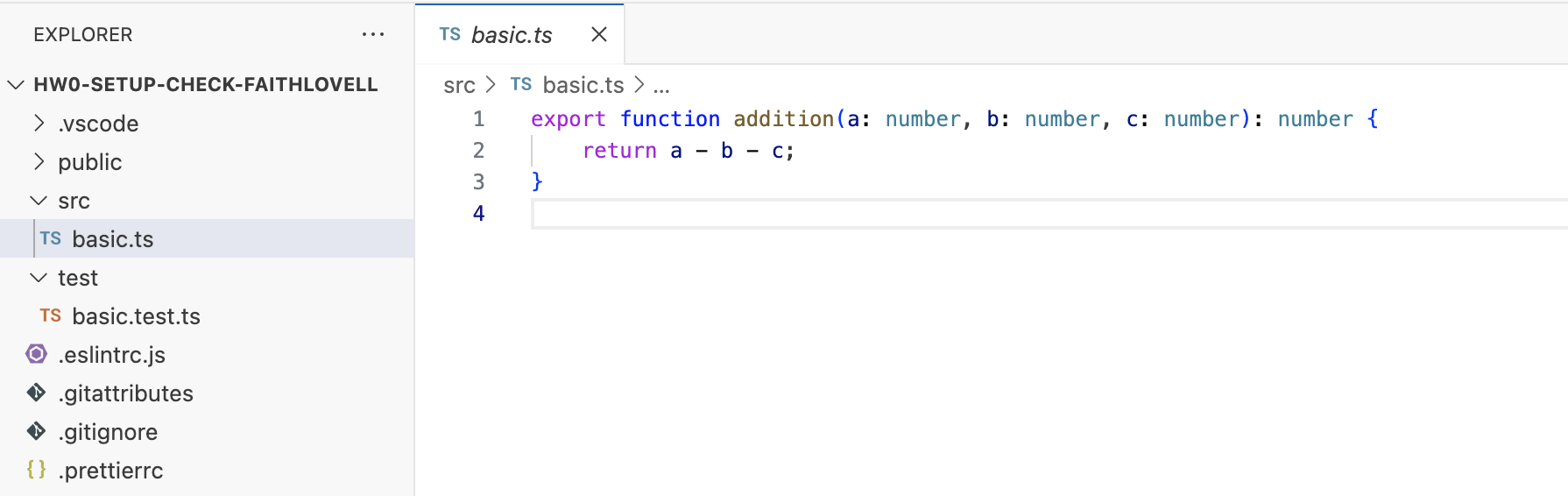
There are a lot of files already in this repository, but we only need to look closely at two of them. If the files are not already visible, click the document icon in the top left of the left navigation bar to see the “Explorer View.”



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.



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.

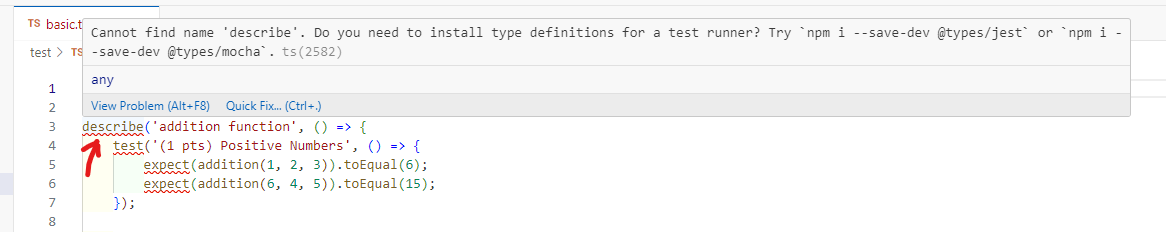
Click on the basic.test.ts file to view the file’s contents:



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.

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.



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.

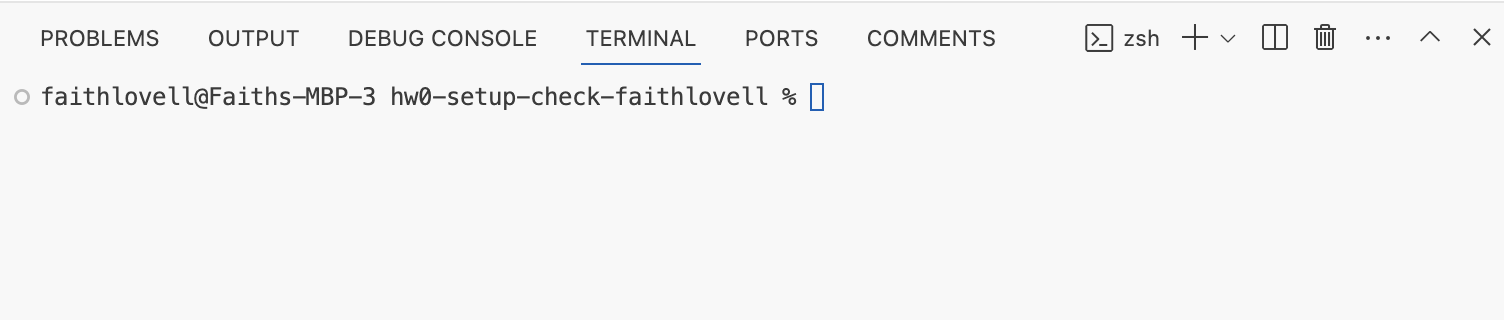
In this case, the system is letting us know that we never installed the necessary modules required to run our program. This is one of the first steps when starting a project - to install the necessary dependencies onto your system. We’ll need to do this every time we clone a new assignment repository.

## 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.

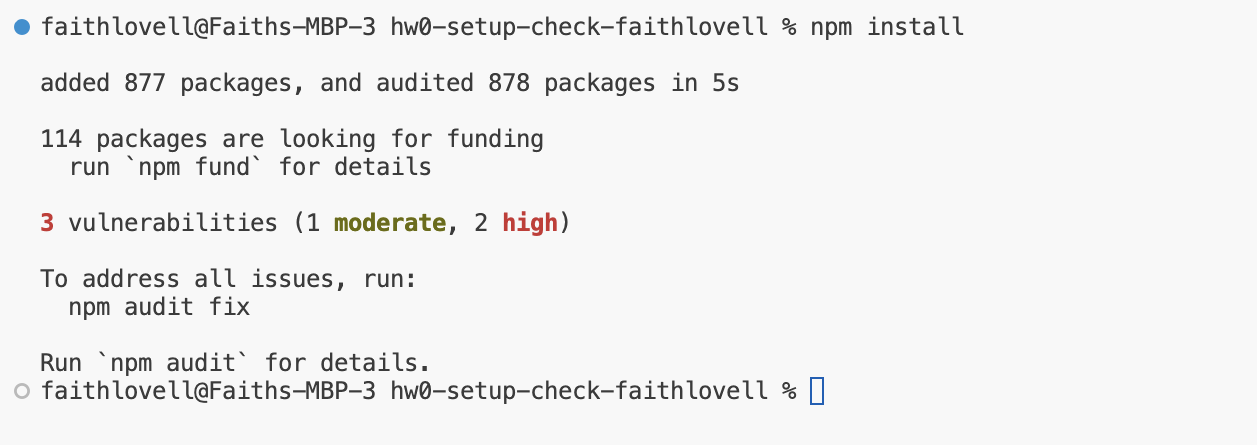
A new terminal will appear at the bottom of your screen, and it will look something like this (notice you are in hw0-setup-check):



Run the command (without the dollar sign).

[INSERT CODE BLOCK]

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



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

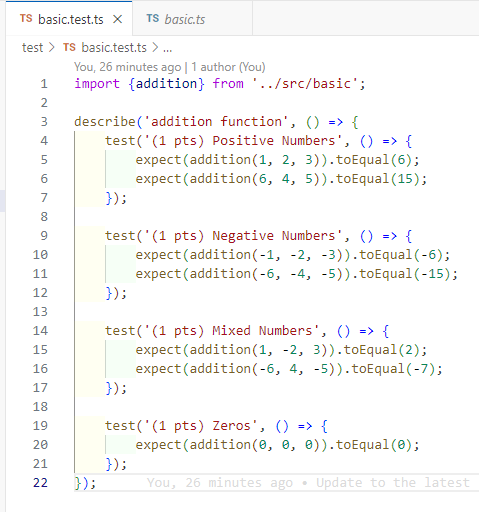
The message also says that “3 vulnerabilities” were detected in the packages, and offers a command to fix them. Again, you might be wondering if you should take the advice of a machine. Remember: Do not run commands you do not understand.

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.



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 exported, 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.

[INSERT CODE BLOCK]

[INSERT CODE BLOCK]

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):

[INSERT CODE BLOCK]

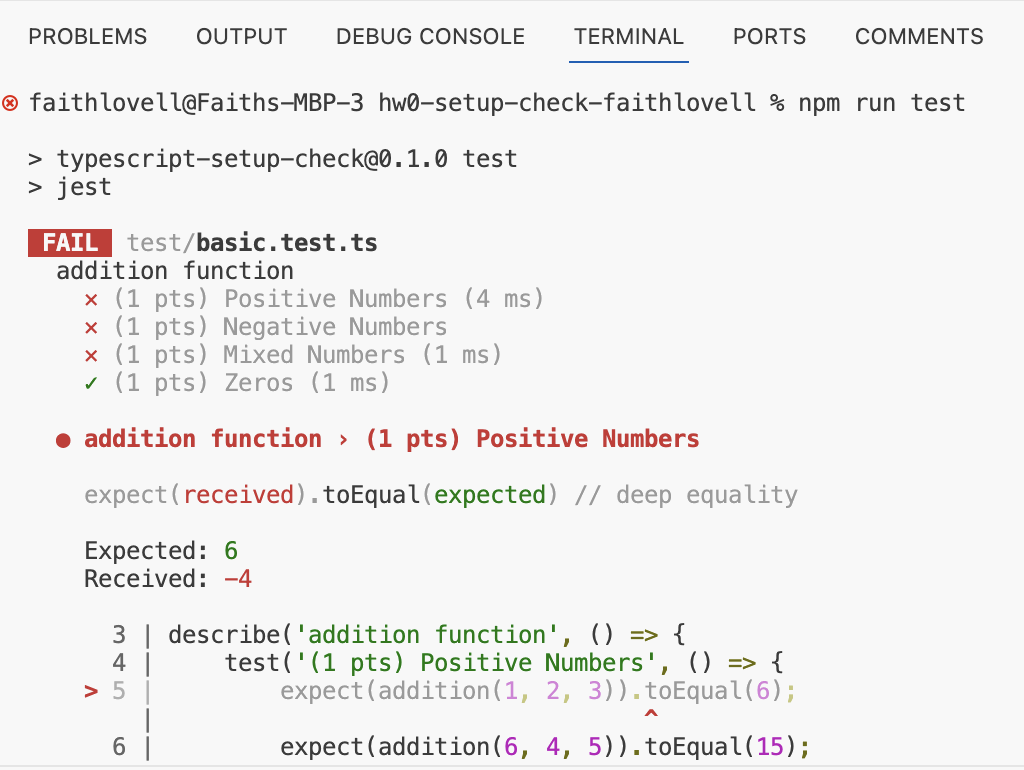
The output might take a little while the first time, and may be so long that it scrolls offscreen. The final output might look something like this:



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”)

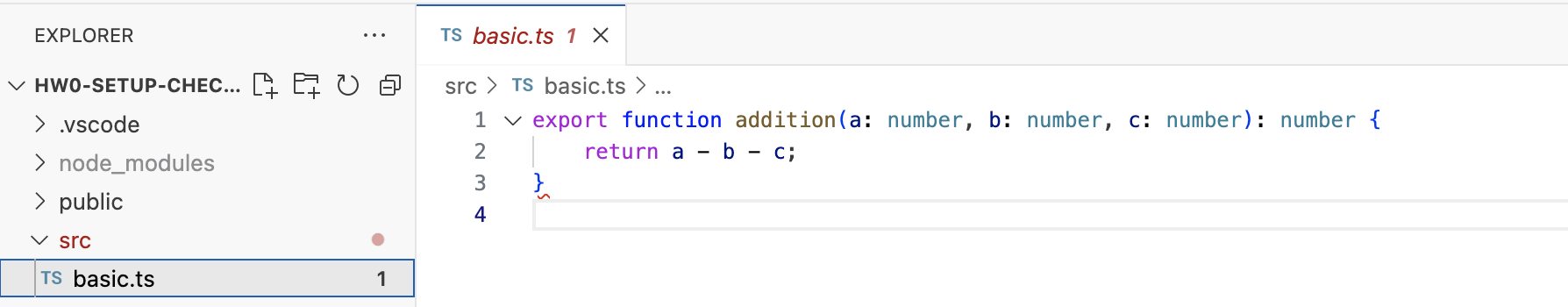
Scrolling up through the output, you can see more details about exactly which tests failed and the specific expect assertions that went wrong.



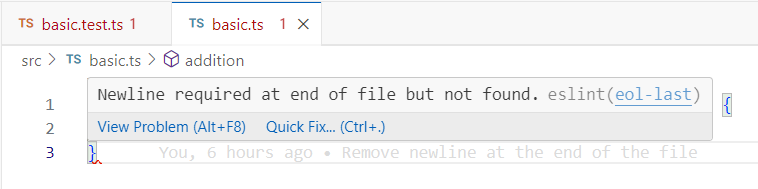
According to the output, the Positive Numbers test called the addition function with the arguments 1, 2, 3 and expected the result to be 6. However, instead the result was -4. There seems to be an error in our code.

## Fix the Code

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



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

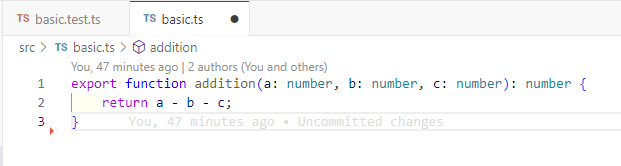


The feedback from the environment has nothing to do with the correctness of our code. Instead, this is the linter (eslint) complaining about the formatting of our file. Specifically, the system wants us to add a newline at the end of our file.

Sometimes, you may disagree with what the linter says. There are 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. 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!

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.



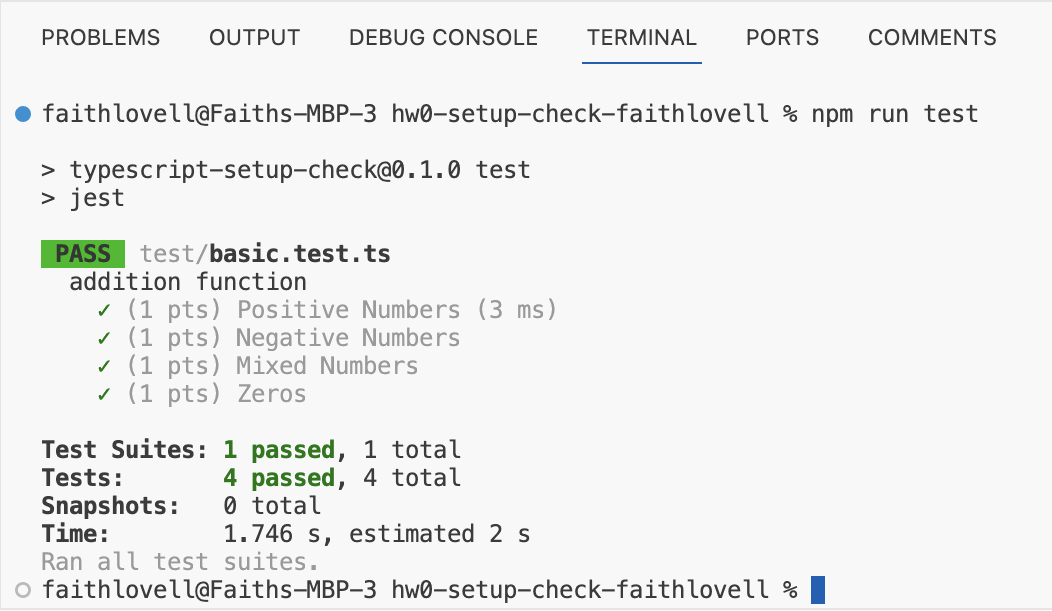
You may notice a black dot next to the name of the basic.ts file in the tab. This indicates that the file has not been saved. Save the file now (either using the appropriate keyboard shortcut or the file menu). If you do not fix AND save the file, then the next step will not work.

## 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.

[INSERT CODE BLOCK]

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



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

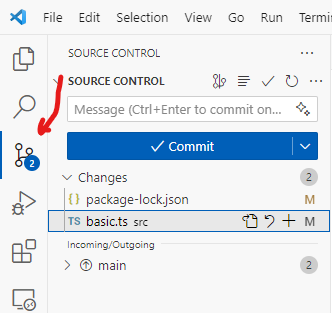
## Stage/Commit/Push to GitHub

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.

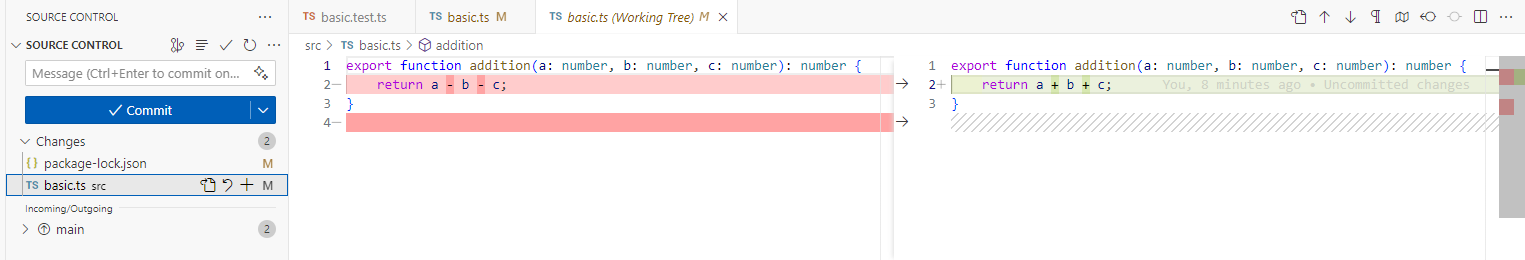


The source control panel gives us a graphical user interface for running Git commands. We could also run them from the terminal, but for now it will most likely be easier to use this interface.

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.

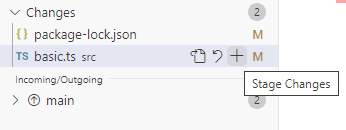
Instead, focus on the basic.ts file. Click on the filename in the Source Control panel and VS Code will show you a diff (“difference”).



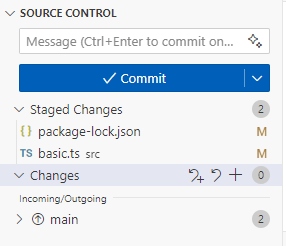
The dark red line was deleted from the left, so it shows up as a grey hatched line on the right. Similarly, the light red line was modified on the left, so it shows up as a green line on the right. These were indeed the changes we made to the file.

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.

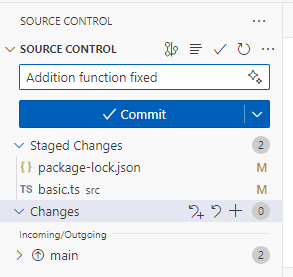


However, first we must write a commit message to explain what we have done. This message should be short, ideally fitting nicely into that box. If someone scrolls through the history of our commit messages, they should have a clear idea of what we did while writing the project.

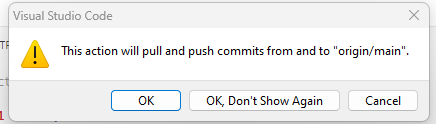
An example commit message here might be Addition function fixed.

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!

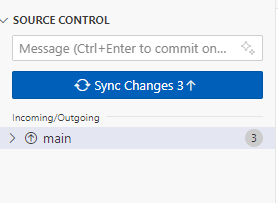
Once you have typed your message, click the “Commit” button to commit your staged changes.



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



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.

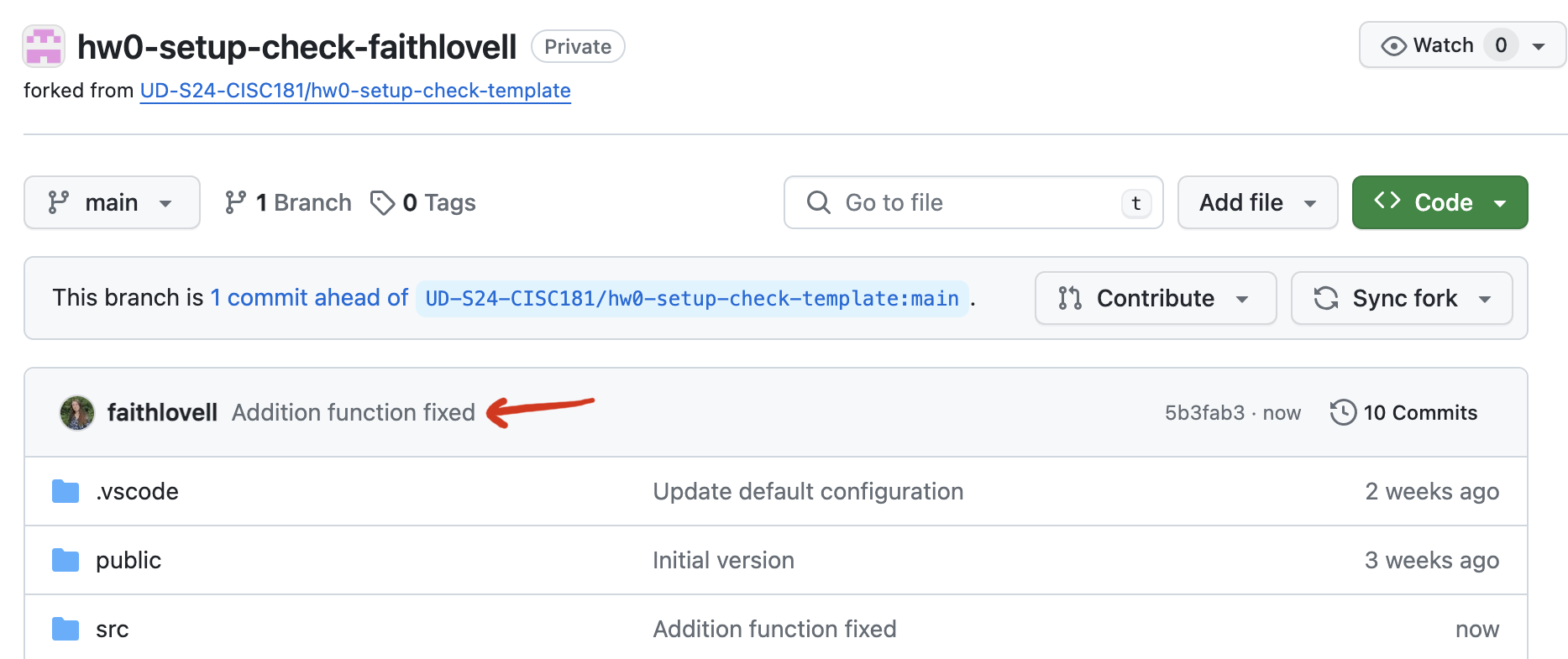
**“NEED TO CONFIGURE GIT”**

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.

[INSERT CODE BLOCK]

Make sure you replace YOUREMAIL with your UD Email, and YOUR NAME with your name (e.g., Austin Bart).

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



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/), 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 hw0.

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 attend to 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, committed, 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!

### Next Step

It has been a couple of months since you completed your CS1 course, so let’s review some concepts from last semester.

While Python and TypeScript (and many other languages) are different from what you have already learned, the basic concepts we learned in CS1 are still valid.

Concepts such as variables, operators, functions, conditionals and looping might have different syntax but largely operate in the same way between languages.

If you understand the basic concepts, picking up a new language like TypeScript is easier. This is a critical skill as new languages and paradigms come into fashion all the time. Learning how to apply your core knowledge to a new situation is critical to stay current throughout your career as a computer scientist.

Let’s start with [Variables](https://boots-edu.github.io/textbook/text/1-typescript/variables.html)

# Variables

## Key Idea

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:

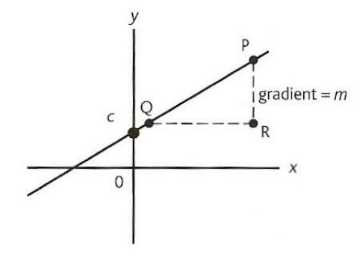
3+4

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.

**ASSIGN BEFORE USE**

Note, if we do not specify a variable’s initial value, then we cannot read the variable’s 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 the answer.

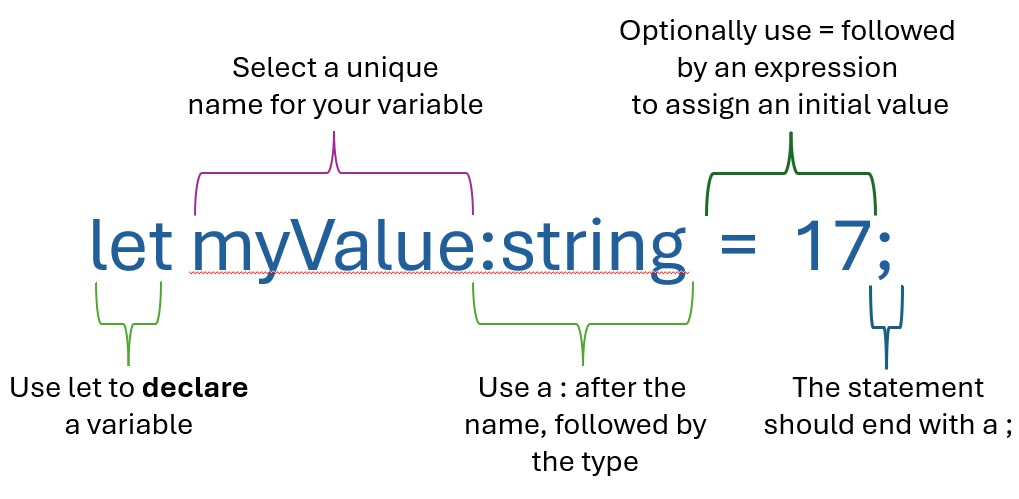
let myValue: number = 4;

let answer: number;

answer = myValue + 3;

console.log(answer);

## Declaration Syntax



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.

## The const Keyword

The const keyword in TypeScript is used to declare variables whose values are intended to remain constant once assigned. It provides developers with a way to ensure that the variable cannot be reassigned, promoting safer and more predictable code.

Immutable Binding: The const keyword ensures that a variable cannot be reassigned after it is initialized. This is particularly useful for defining constants or values that should not change throughout the execution of your program.

const PI = 3.14;

PI = 3.14159; // Error: Cannot reassign a const variable.

While this might seem unnecessary, using *const* allows the programmer to express intention which is both helpful to make code more readable, and useful in debugging.

An example of some defined constants.

const pi: number=3.1415927;

const sol\_ms: number=299792458; *//speed of light in m/s*

In some programming languages, using const can also lead to more efficient memory access as those values do not need to be changed.

## Combining Variables

*// Code to compute the area of a circle with radius 2.*

const 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.

const 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

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.

[INSERT CODE BLOCK]

## 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.

## Next Step

Next we’ll review Functions.

# Functions

## Key Idea

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.

**“CALL” FUNCTIONS, “USE” VARIABLES**

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

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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

### Example Function Calls

[INSERT CODE BLOCK]

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 use 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.

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

### Multiple Arguments to console.log

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

con[sole.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

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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.

## Next Step

Next we’ll review Conditionals.

# Conditionals

## Key Idea

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

[INSERT CODE BLOCK]

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

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

[INSERT CODE BLOCK]

### 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.

[INSERT CODE BLOCK]

### Nesting if inside of Functions

We can also nest if statements inside of functions.

[INSERT CODE BLOCK]

### The else if construct

The following code will execute the first block if x is greater than 4, otherwise it will execute the second block. Inside of the second block, if x is greater than 2, it will execute the third block, otherwise it will execute the fourth block.

[INSERT CODE BLOCK]

Using the else if construct, we can rewrite this code to be more readable:

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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 operators, 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 *not*, we can build complex logic to test state to use in conditionals and loops.

## A Complex Example

[INSERT CODE BLOCK]

## 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.

## Next step

Now onto the next chapter: Strings.

# Strings

## Key Idea

A string is a 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';

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

Note: While you can use either single or double quotes, in practice, you should be consistent.

## 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!

[INSERT IMAGE]

### *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 a string containing the single 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*

Note that the index passed to charAt is 0 based (i.e. 0 is the index of the first charater in the string). In general all string (and array) operations in TypeScript (and most other languages) are 0 based.

### 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.

[INSERT CODE BLOCK]

### Indexes and Slices in Strings

[INSERT IMAGE]

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.

[INSERT CODE BLOCK]

### 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 and 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 split and 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”]

  
let myStr="Hello World";

console.log(myStr.split(" "));

substring(): Returns characters from the string between two defined indexes:

* myStr.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. If the second parameter is not provided, the remainder of the string is returned.

[INSERT CODE BLOCK]

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

* 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.

[INSERT CODE BLOCK]

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.

### String to Number 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:

1. 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 the special value 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";

myStr.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
  + 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.

## Next Step

Now onto the next chapter: Loops and Arrays

# Chapter 2 - Loops and Arrays

Loops and iteration are key concepts in programming. Time to learn how loops work and how to loop through arrays and objects in TypeScript!

# Loops

## Key Idea

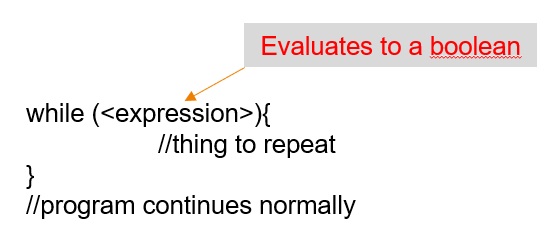
A loop is a control flow structure in programming that allows us to repeat a section of code until some Boolean condition is met.

## 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.



### 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?*

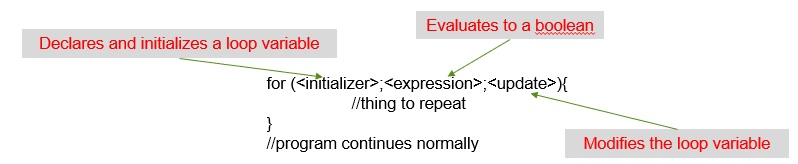
}

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.



* 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.

## Next Step

Next we’ll learn about Arrays

# Arrays

## Key Idea

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"];

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 elements 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]);

* 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 veggies: string[] = ["carrot", "potato"];

let allFood: string[] = [...fruits, ...veggies];

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 veggies: string[] = ["carrot", "potato"];

let allFood: string[][] = [fruits, veggies];

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.

[INSERT CODE BLOCK]

This works and is perfectly acceptable, 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.

## Next Step

Next we’ll learn about Data Classes

# Chapter 3 - Data Classes

Let’s learn about Data Classes.

Data classes allow the programmer to group data that belongs together into a single object that can be referenced through a single variable.

# Data Class Introduction

## Key Idea

*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

* + 1) *Interfaces* describe the data that goes into an object and its types but do not provide default values, or any additional logic.
  + 2) *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.

[INSERT CODE BLOCK]

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();

## Motivation

Some things belong together as they describe a more complex thing that we want to represent. Sometimes it is useful to combine data into a single unit which can be referenced together.

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.

By combining two numbers to represent the x and y coordinates of a Point object, we can reference a point and get its position.

## 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).

## Next Step

Next we’ll learn about creating Basic Data Classes

# Creating Basic Data Classes

## Key Idea

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.

[INSERT CODE BLOCK]

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();

And for a full example:

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

## Next Step

Next we’ll learn about Data Class Constructors

# Data Class Constructors

## Key Idea

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.

[INSERT CODE BLOCK]

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);

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

While much better, the definition of “color” still seems repetitive. While 100% correct, TypeScript gives us a shorthand.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

## Other Drawing classes

The other class we need is a line class. 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.

[INSERT CODE BLOCK]

## 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.

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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 preceded 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 a new instance of the class.

## Next Step

Next we’ll learn about Instances and References

# Instances and References

## Key Idea

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 object in memory. If we call new again, we get a second instance of the class and a second reference to a new memory location.

[INSERT CODE BLOCK]

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;

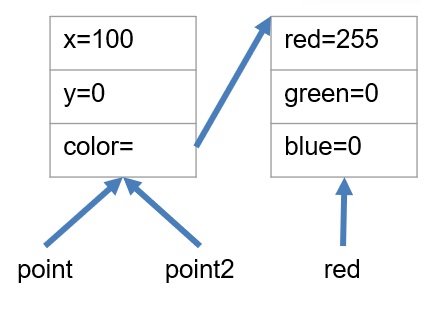
Graphically, this looks like:



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. Thus, they reference the same chunk of memory allocated by the one and only call to new Point(...).

Let’s see that in action.

[INSERT CODE BLOCK]

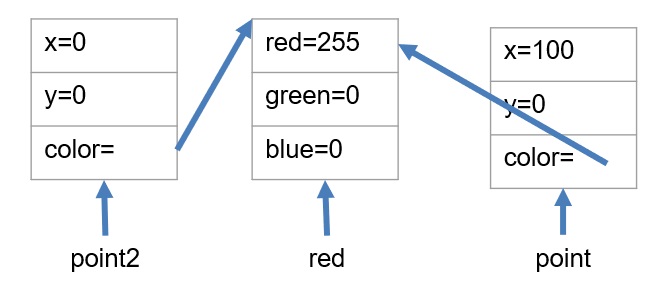


As you can see, updating point updates the memory location referenced by point which is the same memory location referenced by point2. In other words, we only have one point, but we have two references or aliases to that point. Changing either one alters the one and only object that the variables point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

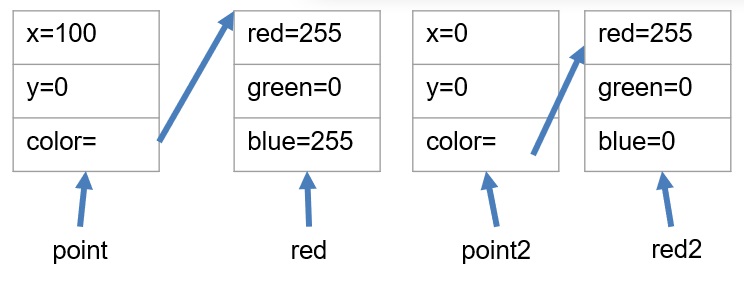
[INSERT CODE BLOCK]



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.

[INSERT CODE BLOCK]



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. This 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.

## Next Step

# Next we’ll learn about the *this* keyword:Instances and References

# Instances and References

## Key Idea

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

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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 contains (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.

## Next Step

Next we’ll learn about generalized Classes

# Chapter 4 - Classes

Let’s learn about more generalized classes.

Classes behave like data classes, but also allow us to create more complex behaviors by including methods within the class that operate on the properties of the class.

# Class Methods

## Key Idea

*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 to 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

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

Note that we don’t need to know how the line is represented to use this method. If we have a line and want to find its 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:

[INSERT CODE BLOCK]

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).

[INSERT CODE BLOCK]

## Another example

Let’s try to add a getArea() method to our Rectangle class. This should be straightforward since we have the corners.

[INSERT CODE BLOCK]

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



[INSERT CODE BLOCK]

## Exercises

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

[INSERT CODE BLOCK]

## 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;

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.

## Next Step

Next we’ll learn about data hiding in Public and Private.

# Public and Private

## Key Idea

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:

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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 that we wrote this for a client, and 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.

[INSERT CODE BLOCK]

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 relabeled 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:

[INSERT CODE BLOCK]

## 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.

## Next Step

Next we’ll learn about Object Cloning

# Object Cloning

## Key Idea

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 and 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 and 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 easy as a shallow copy is sufficient. The classes data items are all primitive types (numbers).

[INSERT CODE BLOCK]

We can create a new Color object from an existing one by calling the existing one’s clone method.

Our Point class 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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

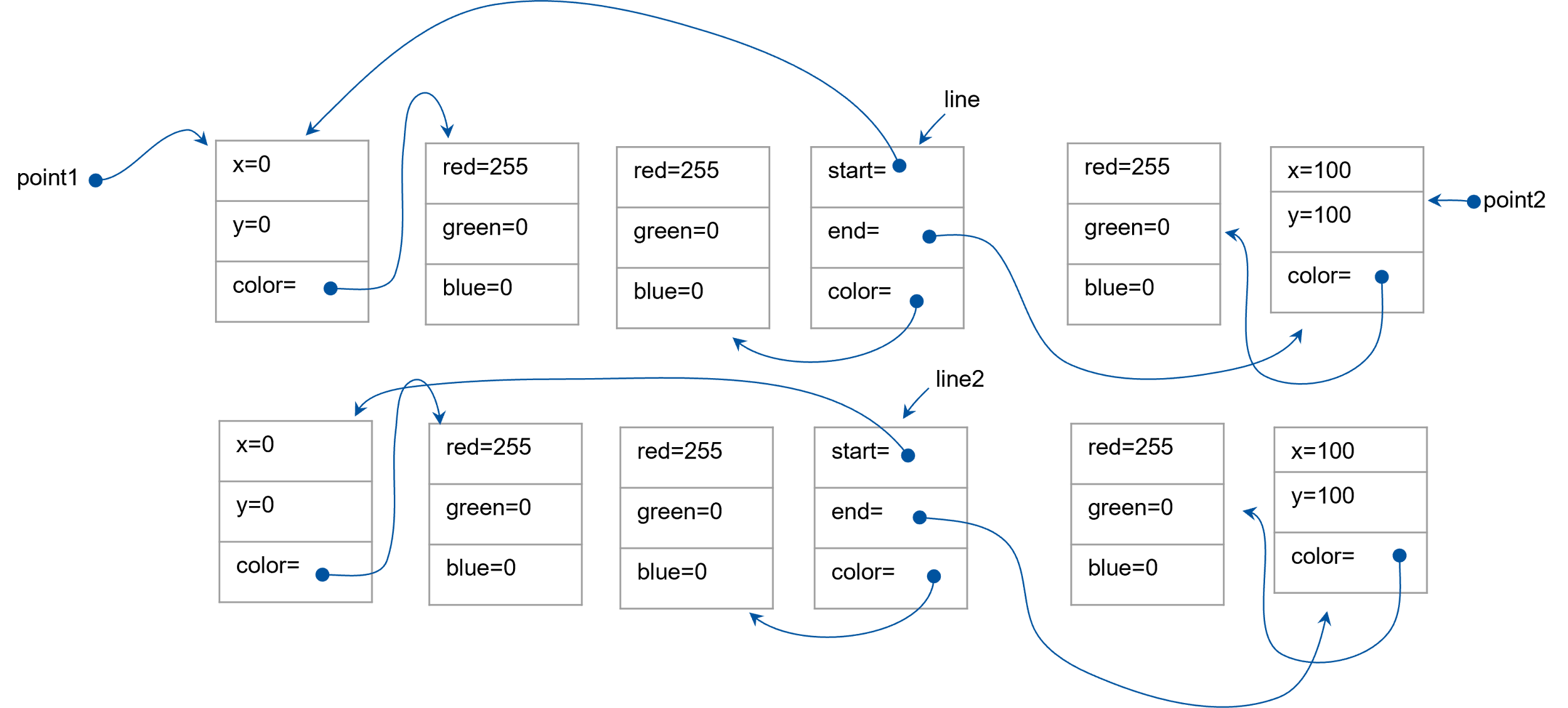
For the Polygon class, things are a little trickier. The class contains an array of references to Point. If we use the spread operator to create a new array, we will only get a shallow copy and the individual points will reference the same Point objects as the original Polygon. We will need to iterate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

[INSERT CODE BLOCK]

## 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.

[INSERT CODE BLOCK]

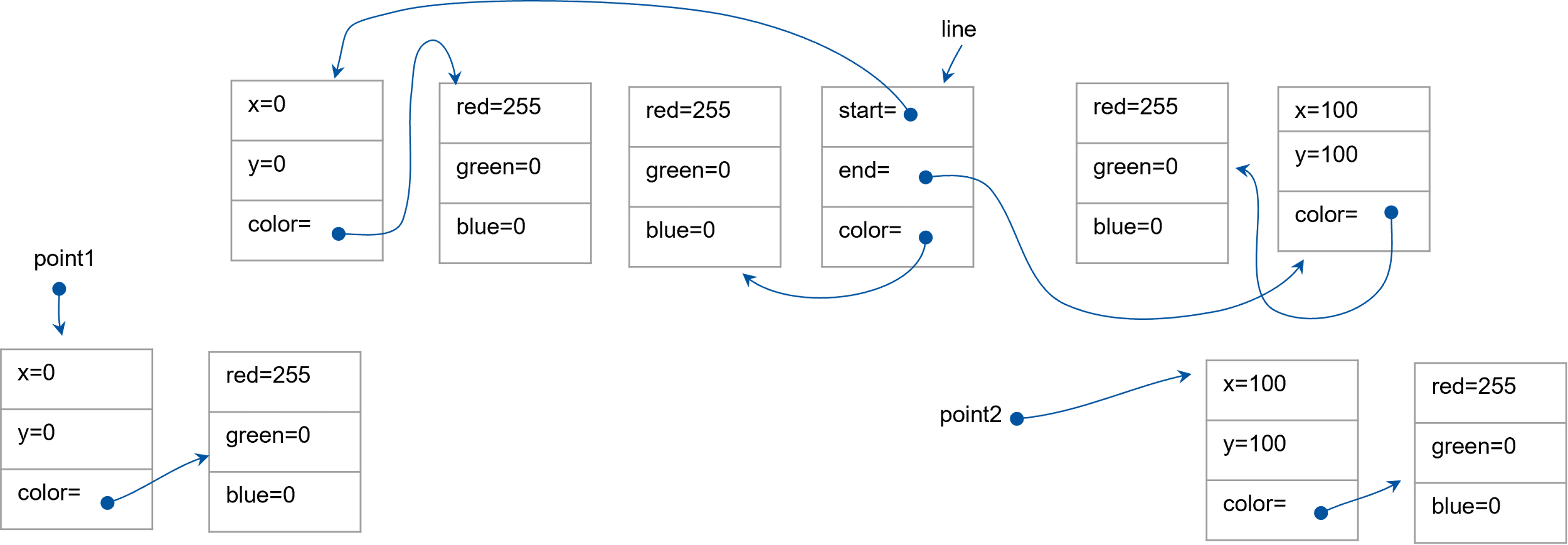


Notice that point1 and point2 are still the same references as we have in line. We can clone the points making them distinct.

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));



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.

## Next Step

Next we’ll learn about composition and inheritance.

# Chapter 5 - Composition and Inheritance

# Building complex objects can be accomplished by combining classes in various ways. Let’s learn the primary methods for doing this *composition* and *inheritance*.Composition

## Key Idea

Using *composition*, we can build complex objects in order to define new types that have 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, then Class2 is composed of Class1.

Here is a visual representation of composition, where a Point class contains a Color class:

Pointx: numbery: numbercolor: ColorColorred: numbergreen: numberblue: number

This would translate into the following code:

[INSERT CODE BLOCK]

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 represent 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.

[INSERT CODE BLOCK]

Compositionallows 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.

## Next Step

Next we’ll learn about inheritance.

# Inheritance

## Key Idea

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 basket contains fruit. While useful in many situations, we often want 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.

[INSERT CODE BLOCK]

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 {}

We can now define objects of type student and teacher, and instantiate them with new and they work just like our users class.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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?

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

Here is a completed example:

[INSERT CODE BLOCK]

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 inheritance through the extends keyword, we can express both the similarities and differences 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.

## Next Step

Next we’ll learn put these concepts to use: Putting it All Together

# Putting It All together

## Key Idea

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

## Terminology 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*.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

Remember that “drawable” is the *superclass, base class, parent class*. “Line” is the *subclass, child class.*

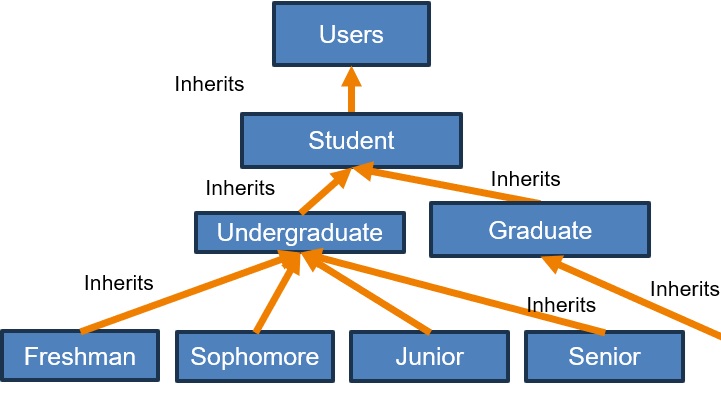
[INSERT CODE BLOCK]

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 undergraduate 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 a *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.

## Next Step

Next we’ll learn about Overrides and Polymorphism.

# Chapter 6 - Overrides and Polymorphism

While classes are a powerful tool for software engineering, the real power comes from the ability to alter the behavior in a derived class. This is accomplished by *overriding* members of the superclass in the subclass to change the behavior. This opens up a type of *polymorphism* which will allow us to write concise programs that behave differently based on the *overridden* member.

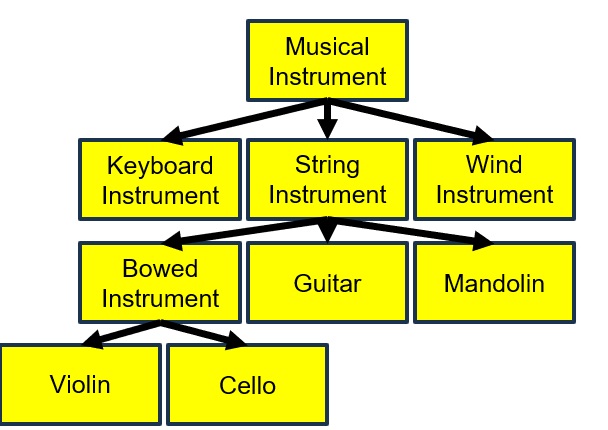
# Member Access

## Key Idea

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* relationship lower with its superclass.

Assume that the musical instrument has a name, a musical key (i.e., 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 affect 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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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 constructor 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 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.

## Next Step

Next we’ll learn about overriding class methods in Overrides

# Overrides

## Key Idea

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

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

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 use 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.

[INSERT CODE BLOCK]

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.

## Next Step

Next we’ll learn about Polymorphism.

# Polymorphism

## Key Idea

*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.

[INSERT CODE BLOCK]

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*:

[INSERT CODE BLOCK]

Note: You can install the drawing library used 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.

## Another Example

Consider the following example. We would like the console.log expression to print the value of x, but the default behavior of Object.toString() is not what we really want.

[INSERT CODE BLOCK]

To rectify this, we can override the toString() method to control how the conversion to a string takes place.

[INSERT CODE BLOCK]

Note: If you do not extend the object, you cannot use the override keyword, however this still works and provides the same behavior.

[INSERT CODE BLOCK]

## Summary

*Polymorphism* 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.

## Next Step

Next we’ll learn about Abstract Classes.

# Abstract Classes

## Key Idea

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);

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:

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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.

## Next Step

Next we’ll summarize what we have learned about polymorphism in Polymorphism Notes

# Polymorphism Notes

## Key Idea

*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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

### 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.

[INSERT CODE BLOCK]

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 object-oriented programming. In other words, while the syntax may differ slightly, all of the concepts hold true in most object-oriented 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, thanks to *inheritance*.
* We can use the idea of *polymorphism* to reference objects through their superclass, and have the correct implementation in the subclass executed 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.

## Next Step

Next we’ll learn about Exceptions and Code Quality.

# Chapter 7 - Exceptions and Code Quality

# Exceptions

## Key Idea

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 exceptions 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.

[INSERT CODE BLOCK]

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

[INSERT CODE BLOCK]

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 two 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.

[INSERT CODE BLOCK]

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 nonsensical values to be stored in red, green and blue. But we can use exceptions to prevent this.

[INSERT CODE BLOCK]

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 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.

[INSERT CODE BLOCK]

Throws (or “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.

[INSERT CODE BLOCK]

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, we can create polygons with 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 we use exception handling to make sure it is not possible to create an invalid polygon?

[INSERT CODE BLOCK]

Now, if we 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?

[INSERT CODE BLOCK]

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).

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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

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 elegantly 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.

[INSERT CODE BLOCK]

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).

[INSERT CODE BLOCK]

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.

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:

[INSERT CODE BLOCK]

If we 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; this is allowed, but it is not recommended. It is better to throw an Error object, or an object that extends Error.
* 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.

Here is what we mean by rethrowing an exception:

[INSERT CODE BLOCK]

## 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).

## Next Step

Next we’ll learn about Comments.

# Comments

## Key Idea

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

## Code Quality

Code quality 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, what is the user experience like?

## Why Care About Code Quality?

Comments help others (and yourself) use your code without having to read it. Comments inform the 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.

Good names: 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.

## Comments

At this point, you should be convinced that comments are worth your time. Now let’s 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.

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

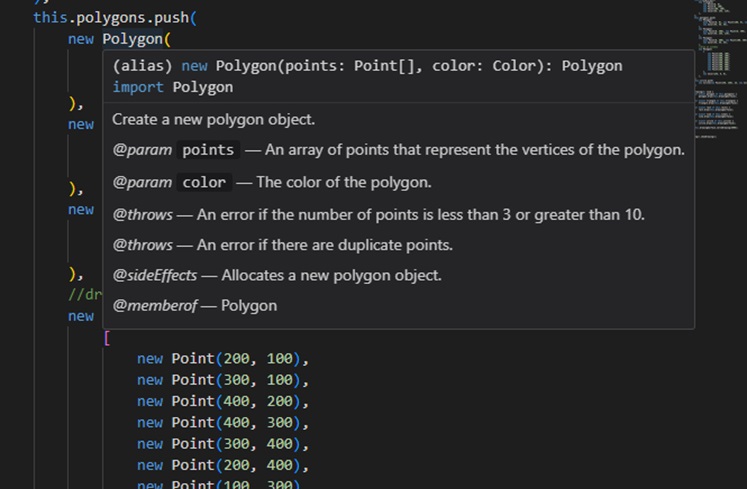
* We see the parameters and their types and description.
* What exceptions to expect
* It’s side effects
* It’s parent class
* It is a constructor
* An example of how to use it.

The clone method as well:

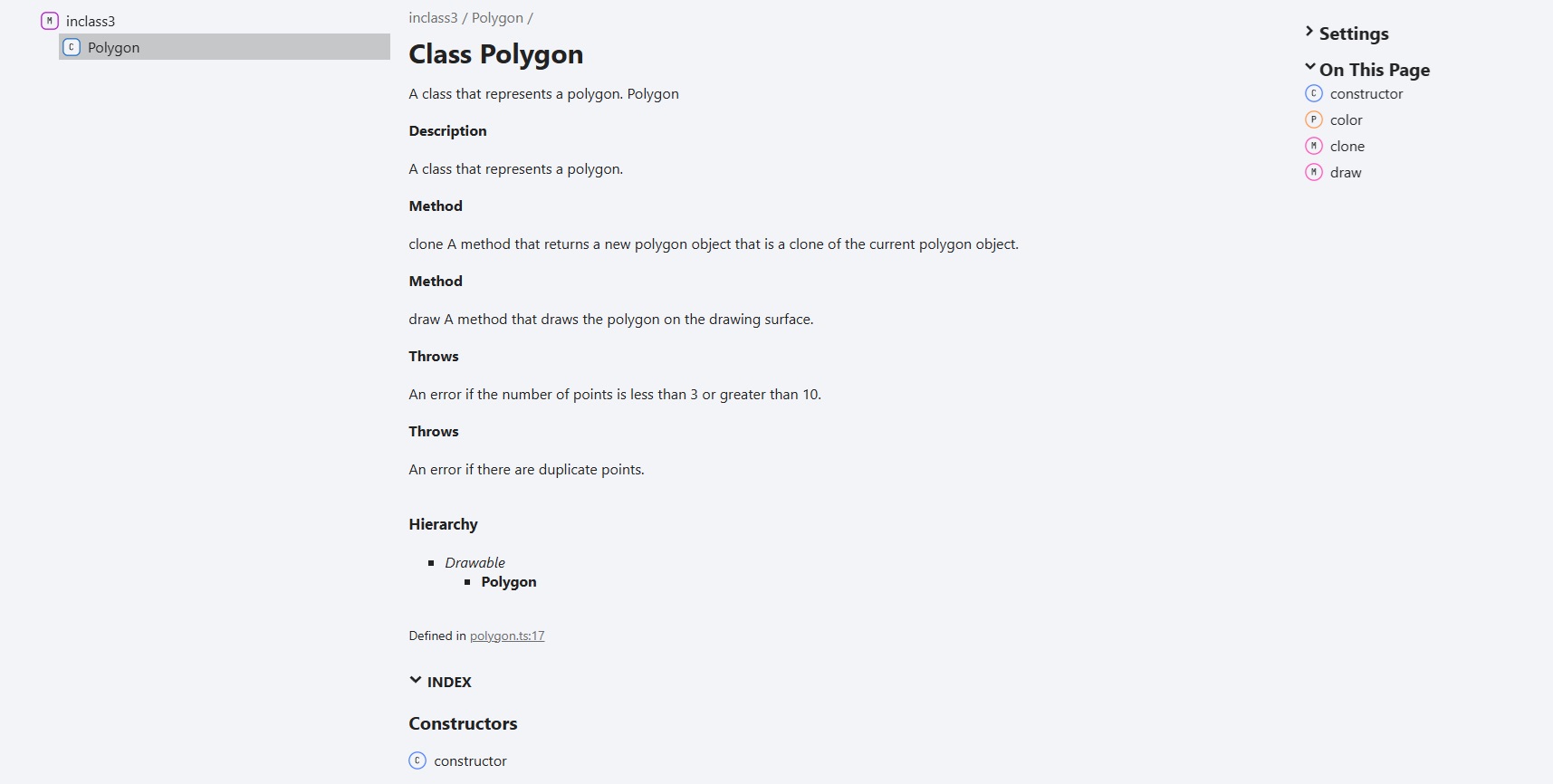
[INSERT CODE BLOCK]

* 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.

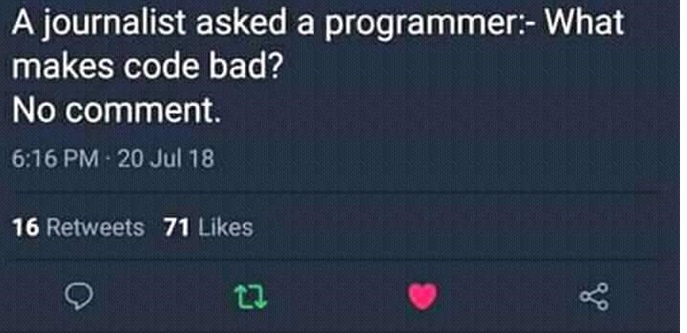
Why bother with all this formatting?



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.



Quality, well formatted comments make your code more usable, manageable, and maintainable.



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.

## Next Step

Next we’ll learn about the importance of Naming.

# Naming

## Key Idea

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:

[INSERT CODE BLOCK]

* 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:

[INSERT CODE BLOCK]

* 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:

[INSERT CODE BLOCK]

* 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:

[INSERT CODE BLOCK]

* 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.

## Next Step

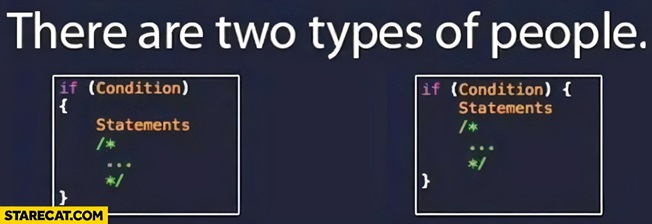
Next we’ll learn about some general practices to improve General Code Quality.

# General Code Quality

## Key Idea

Programmers should always try to create efficient, readable, and maintainable code. It’s not hard to do it right!

## Best Practices

 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 #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 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.

## Next Step

Next we’ll examine Testing.

# Chapter 8 - Testing

Testing code is critical! In this chapter, we’ll discuss how to write tests for your code, including both unit tests and integration tests. We’ll also discuss how to use Jest, a popular testing framework for JavaScript and TypeScript. This will also be an opportunity to discuss anonymous functions and how they can be used in testing.

# Testing

## Key Idea

*Software testing* is the process of validating that software is bug free and meets requirements.

## General

What we must do 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.

## In other words, 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.

[INSERT IMAGE]

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 our 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.

[INSERT CODE BLOCK]

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?

Here is another example:

[INSERT CODE BLOCK]

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 things should be tested. Make sure you test edge cases and exceptional situations to make sure you have covered all possible inputs.

## Next Step

Next we’ll learn how to write tests in Jest in Testing in Jest

# Testing in Jest

## Key Idea

Jest is a test runner and testing framework that works with JavaScript and TypeScript. We’ve been using Jest the entire time!

## Jest Syntax

There are just a few Jest functions that you need to know to get started:

* 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 methods we’ll need to learn about, but we can get by with these three functions for now.

Back to the example from the previous section, let’s look at what we want to do for each of these:

[INSERT CODE BLOCK]

We will start with a describe block for the elements class:

[INSERT CODE BLOCK]

Let’s write a test to see if we can construct an instance of the class:

[INSERT CODE BLOCK]

Does it work normally? Just create a test block that populates some items and then verify them.

[INSERT CODE BLOCK]

This will test the normal case of the function. What about the edge case where the array is empty?

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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?

[INSERT CODE BLOCK]

So what does our final test suite for this code look like?

[INSERT CODE BLOCK]

If we run our test using the Jest command line, we get

[INSERT CODE BLOCK]

Here we can see that all 4 tests passed. Now we know that the class method works as expected, in all of the cases that we could think of. We have also verified that it behaves as documented on exceptions. Once written, this test will run every time we 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:

[INSERT CODE BLOCK]

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 across the application.

## Next Step

You may have noticed the odd syntax. Before we continue, let’s cover Anonymous Functions to help us better understand the testing syntax.

# Anonymous Functions

## Key Idea

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.

[INSERT CODE BLOCK]

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.

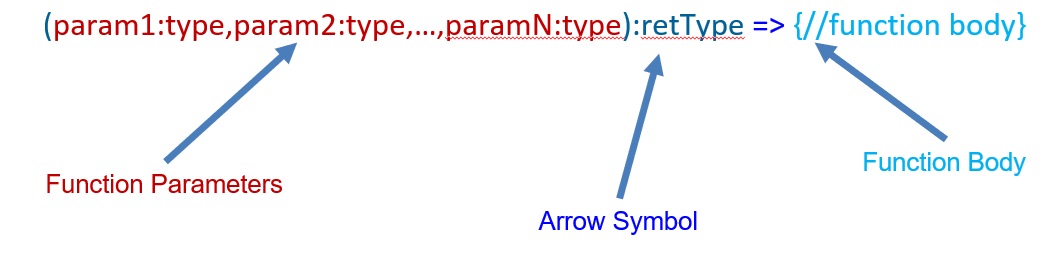
[INSERT CODE BLOCK]

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 are referred to as “first class objects.”

### Syntax

Let’s look at the overall structure of an anonymous function: 

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.

[INSERT CODE BLOCK]

There is another way to accomplish this using the typescript Array.filter method.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

But wait, there’s more.

## First Class Objects

Being *first class objects,* functions can be used in many places:

* As a parameter to methods.

[INSERT CODE BLOCK]

* As the value of a variable or class property.

[INSERT CODE BLOCK]

* And as the return value of a function.

[INSERT CODE BLOCK]

## 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.

[INSERT CODE BLOCK]

We can then call those functions just like we would if they were defined with a name.

[INSERT CODE BLOCK]

We can even declare a type to use for our functions.

[INSERT CODE BLOCK]

## Summary

*Anonymous 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.

## Next Step

Next we’ll start to learn about web programming through Webz Introduction.

# Chapter 9 - Webz Introduction

Webz is a simple web framework designed to minimize the learning curve for web application development. It was designed to use many design concepts from more complex web platforms, but to simplify the learning curve for the purposes of this text.

# Web Basics

## Key Idea

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). *Domain Name Service* (DNS) 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 *Hypertext Transport Protocol* (HTTP) or its 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

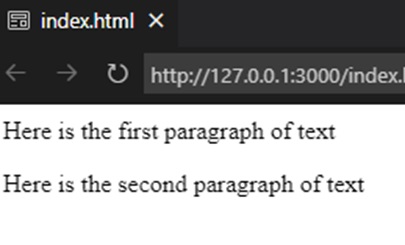
Common tags:

* <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>: A regular 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 box

### HTML Tags in action

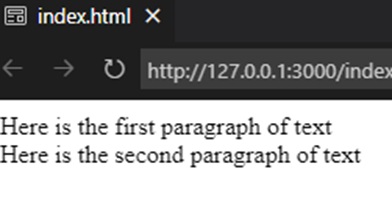
[INSERT CODE BLOCK]

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 to make it more important.



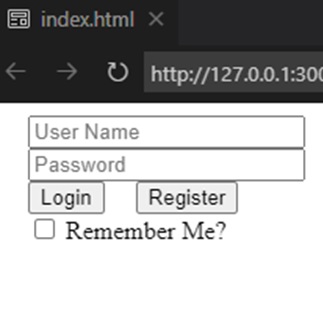
The two paragraph tags simply output the text to the browser with paragraph spacing between them. If we want less spacing between the two lines of text, we can use a line break instead of putting the text in paragraph tags.

[INSERT CODE BLOCK]



By removing the <p></p> tags, and adding a <br/>, we get line spacing instead of paragraph spacing. You can set various attributes on each tag. Here is a simple login screen:

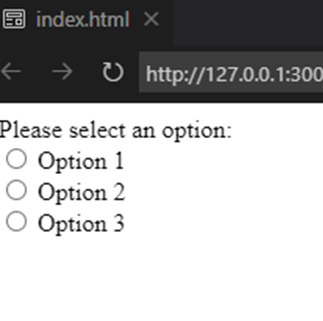
[INSERT CODE BLOCK]



Notice the placeholder attribute being set (and the type attribute).

Here is a set of radio buttons:

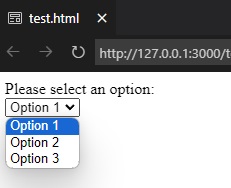
[INSERT CODE BLOCK]



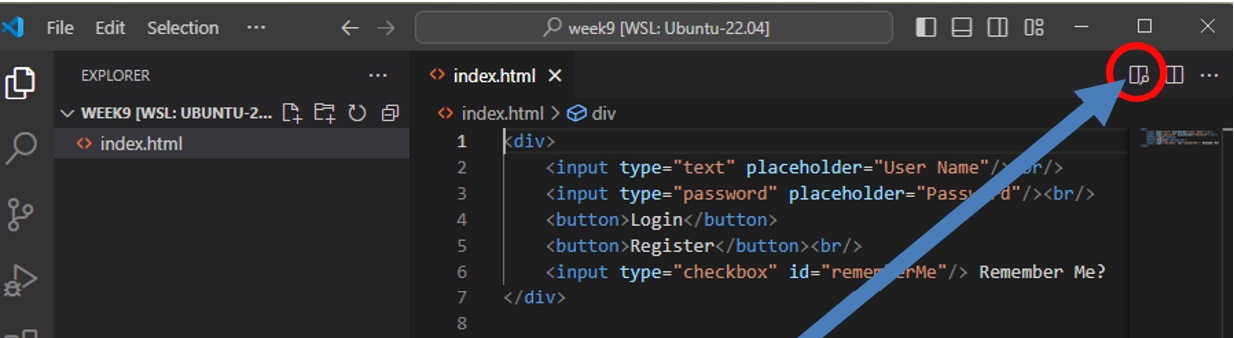
If radio buttons have the same name property, they will act as a group, where selecting one deselects the others.

Here is a dropdown box:

[INSERT CODE BLOCK]



### Web Development in VS-Code



Clicking this button will bring up a preview panel on the right which will change automatically as you edit the file.

[INSERT IMAGE]

### Styling and CSS

Styling our elements allows us to alter colors, shapes, behaviors, appearance, and placement. Often these styles are placed in a separate file with a .css extension (stands for *cascading style sheets*). 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.

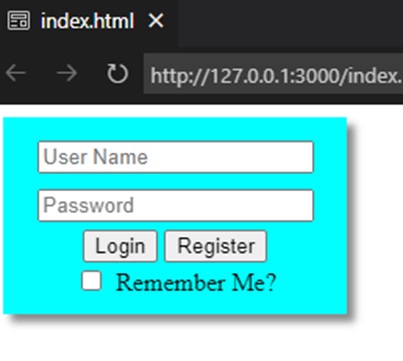
Returning to our login screen example, we have added *id* and *class* attributes to our elements to allow us to style them.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

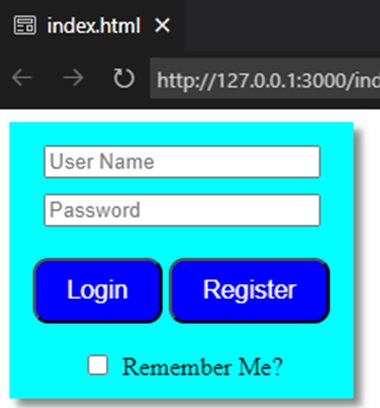


The first rule tells the loginForm div to center its content horizontally and set its background color. The second is a combination of rule selectors. This rule says to style all input tags inside the element with id #loginForm.

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 preceding the class name with a “.” so btn is referenced as .btn.

[INSERT CODE BLOCK]



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 that 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. No one can know them all, and 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 future projects.

## 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 applications instead of static web pages.

## Next Step

Next we’ll introduce Webz, a lightweight web framework for application development, in Beginning Webz.

# Beginning Webz

## Key Idea

*Webz* is a lightweight web framework designed for this book. It uses many of the same principles as more advanced web frameworks such as React and Angular, but simplifies operations to focus on TypeScript development.

# Working Example of Webz

Below, you can see a fully working example of Webz in action. Try running the code to see what it looks like in action! Notice that there are three files for the component: main.component.ts, main.component.html, and main.component.css. The HTML file contains the structure of the component, the CSS file contains the styling, and the ts file contains the logic. The ts file uses decorators to attach methods and properties of the class to the HTML by the element’s id attribute. We’ll explain all of this in more detail throughout this chapter.

[INSERT CODE BLOCK]

[INSERT CODE BLOCK]

[INSERT CODE BLOCK]

## 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 one 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.

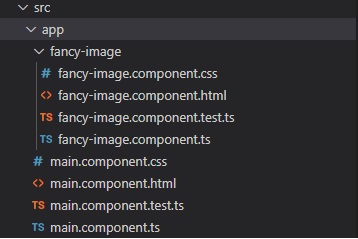
webz new first-example

This creates a fully working website with one component in it (MainComponent) and 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).



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):

[INSERT CODE BLOCK]

And the typescript class (main.component.ts):

[INSERT CODE BLOCK]

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 works with npm run start. We should see the default text for fancy-image and two 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 its size:

[INSERT CODE BLOCK]

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 imagePath property in the following code. Here in this online version of the book, we use the path ../../assets/images/pet-ada.jpg; but you would be able to use assets/images/pet-ada.jpg in your local environment (as long as you place a file named pet-ada.jpg in the assets/images folder).

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

If you run this code with npm run start you will see the image displayed.

Let’s see the new example so far:

[INSERT CODE BLOCK]

main.component.html

<div id="image-holder"></div>

<br />

<button id="prev">Previous</button>

<button id="next">Next</button>

fancy-image.component.ts

[INSERT CODE BLOCK]

fancy-image.component.html

<img id="image"></img>

Fancy-image.component.css

[INSERT CODE BLOCK]

## 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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

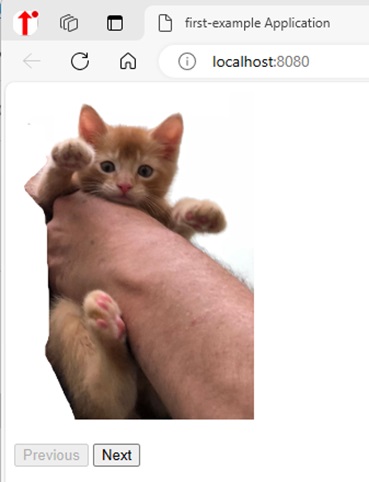
Next, we need to bind the button’s click events to a function that increments/decrements the value and properly sets the values of prevDisabled and nextDisabled. Notice that the HTML id of the buttons is prev and next. We use that in the @Click decorator to specify which button we are binding.

[INSERT CODE BLOCK]

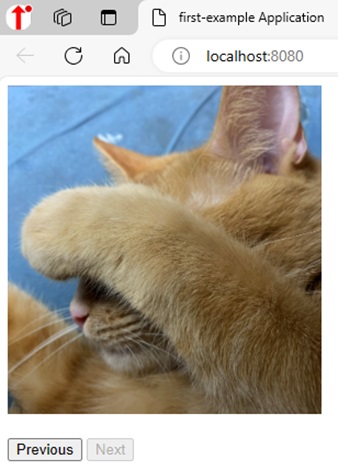
If next is pushed:

* 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 pet-1.jpg displayed and our buttons.



Clicking the next enables the previous button, disables the next button, and displays pet-2.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.

And here is the complete example here in the browser:

main.component.ts

[INSERT CODE BLOCK]

main.component.html

[INSERT CODE BLOCK]

fancy-image.component.ts

[INSERT CODE BLOCK]

fancy-image.component.html

<img id="image"></img>

fancy-image.component.css

[INSERT CODE BLOCK]

## Decorators in Webz

Here, we will list out all of the useful decorators in Webz. You can only use these on a class derived from WebzComponent. All classes created with the CLI will automatically subclass WebzComponent. The Bind Decorators are used to move data from the class to the HTML. The event decorators are used to move data from the HTML to the class.

### Bind Decorators

First, here are the decorators for binding variables to elements, which allow the element to change when a field (member variable) changes in the class:

* @BindValue(id, ?trans): Binds a member variable to the value of an element (useful for changing the text of an element)
* @BindAttribute(id, attr, ?trans): Binds a member variable to an HTML attribute of an element (useful for changing the rest of the content of an element)
* @BindStyle(id, style, ?trans): Binds a member variable to a CSS style of an element (useful for changing the look and feel of an element)
* @BindCSSClass(id, class, ?trans): Binds a member variable to a CSS class of an element (useful for quickly changing many style properties at once)

Some of these decorators have an optional transform function that can be used to transform the value before it is applied to the element. This is useful for things like adding a prefix to a value or converting a number to a string.

Some of these transformations are so common (like converting a number) that Webz provides them for you. For example, @BindValueToNumber will convert a string to a number before applying it to the element. Here’s the complete list:

* @BindValueToNumber(id, ?append): Converts the value from a number to a string before applying it to the element (useful for binding numeric fields to text elements)
* @BindCSSClassToBoolean(id, class): Applies the class to the element if the value is true, removes it if the value is false (useful for conditionally applying styles)
* @BindDisabledToBoolean(id): Disables the element if the value is true, enables it if the value is false (useful for conditionally disabling things)
* @BindVisibleToBoolean(id): Shows the element if the value is true, hides it if the value is false (useful for conditionally showing things)
* @BindStyleToNumber(id, style, ?append): Converts the value from a number to a string before applying it to the element (useful for binding numeric fields to style properties)
* @BindStyleToNumberAppendPx(id, style): Converts the value from a number to a string and appends “px” before applying it to the element (useful for binding numeric fields to style properties that require “px” units, such as padding, margin, width, height, etc.)

### Event Decorators

On the other hand, when you want the instance to react to an event on the element, you can use the decorators below to bind a method to an event. Most of the time, you will want to use one of these specialized decorators:

* @Click(id) (e: MouseEvent)=>{} for when any element (like a button or an image) is clicked on with the mouse or tapped.
* @Change(id) (e: ValueEvent)=>{} for when an element’s value changes (like a dropdown box).
* @Input(id) (e: ValueEvent)=>{} for when an editable text element’s value changes (like a textbox or input box).
* @Blur(id) (e: Event)=>{} for when an element loses focus (like clicking away from a textbox).

Then there are the generic event decorators, which can be used for any HTML event (keep in mind that there are actually many [HTML events](https://www.w3schools.com/jsref/dom_obj_event.asp)):

* @GenericEvent(id,eventType) (e:Event)=>{} for when ANY event occurs on an element. This can be used to capture other kinds of events not covered by the specialized decorators.
* @WindowEvent(eventType) (e:WindowEvent)=>{} for when an event occurs on the window (like resizing the window, closing the tab, or typing a key outside of an element).
* @Timer(milliseconds) (f:TimerCancelFunction)=>{} for setting a timer that will go off later. A future chapter will talk about timers in more detail.

## References

This is merely scratching the surface of what is possible with web development. Here are some references to help you learn more about HTML and CSS.

HTML References:

* Intro: https://developer.mozilla.org/en-US/docs/Learn/Getting\_started\_with\_the\_web/HTML\_basics
* 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 to experiment in:

* 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.

## Next Step

Next we’ll walkthrough building a Webz application, step by step, in Webz Basic Walkthrough.

# Webz Basics Walkthrough

## Key Idea

*Webz* can be used to create an interactive website.

This walkthrough will help you build and deploy your first Webz application. You will learn how to create components, bind values, and handle basic events.

## 0) Setup

* Use the GitHub classroom link provided in the original assignment on Canvas to create your own copy of the starter repo.
* Clone the repo to your local machine in an appropriate directory.
* Open the directory in VS Code, as you normally do.
* Run npm install in the VS Code terminal to install the dependencies.

npm install

* Run npm run start in the terminal to start the development server. This may take a few seconds to compile the code and start the server. If you need to stop the server, you can press Ctrl+C in the terminal.

You can click here to see what the output looks like for us when the server starts successfully.

* Although we could now open your website in chrome at the localhost url http://localhost:8080, we will use the integrated debugger in VS Code. Activate this by pressing F5 on your keyboard (or selecting the Run tab from the top menu and then clicking Start Debugging). This will open a new browser window with your application running. The debugger has a bunch of useful features, like setting breakpoints and inspecting variables—we’ll talk more about them later on.

**DEBUGGING IN VS CODE**

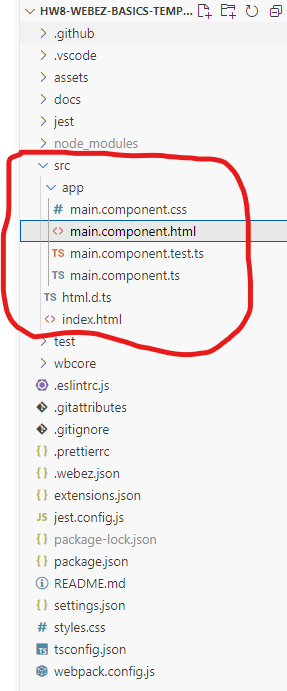
You can only activate the debugger if you have the server running. If you close the server, you will need to start it again before you can use the debugger.

* You should now be able to see your website. It won’t be very exciting at first, but you can now start editing the code and see the changes in real-time!

## 1) Basic HTML

Let’s start by adding some dynamic text to the page. We’ll need to edit the MainComponent, which came with the starter repo. This is the main component that is displayed on the page, and there’s always exactly one of them in every Webz application. Although we created the component for you, we haven’t done anything else. Let’s add some text to the page.

* Expand the src/app folder in the sidebar. You will see four files there: main.component.ts, main.component.html, main.component.css, and main.component.test.ts.



* main.component.html is the HTML file that defines the structure of the component. This is where we put the visual content of your application.
* main.component.css is the CSS file that defines the styles for the component. This is how we make things look pretty.
* main.component.ts is the TypeScript file that defines the MainComponent class. This is where the application logic will go.
* main.component.test.ts is a (mostly empty) test file for the component. We’ll talk more about testing your own components later on.
* Open main.component.html. You will see that it contains the following code:

[INSERT CODE BLOCK]

This is the default content that comes with the starter repo. The div tag is a generic container that can hold other elements (a “division” of content). In this case, we have a div with the class header, which contains a div with the class title. The class attribute is used to apply CSS styles to the element. We’ll talk more about CSS later on. For now, focus on the text inside of the div tags.

* Change the text inside the div with the class title to something else. For example, you could change it to:

<div class="title">My First Webz Application</div>

*   
  Save the file. You should see the changes reflected in the browser window automatically. If you don’t see the changes, make sure that the debug mode is still running.
* Now, let’s add a new element to the page with some more text. Add the following code after the div with the class header, but inside the outer div. The p tag is used to create a paragraph of text.

<p>Welcome from the HTML side!</p>

If you did this correctly, the HTML should look something like the HTML below. Pay close attention to the structure of your HTML, and make sure you are closing all of your tags properly. The location of tags is important, as it determines how the elements are displayed on the page.

[INSERT CODE BLOCK]

* Save the file, and confirm that the new text is displayed in the browser window.
* We’re going to modify the p tag to include an id attribute. This will make it easier to test our application later on. Add the id attribute to the p tag, like so:

<p id="example-text">Welcome from the HTML side!</p>

Make sure you use the exact id value (example-text) as shown above. When you save the file, the page will not look any differently. The id attribute is used to uniquely identify an element in the HTML document, but does not affect the appearance of the element.

* Now, let’s add some dynamic text to the page. We’re going to use TypeScript to change the text of a new p tag. Make a second, empty p tag with an id after the existing one, like so:

<p id="example-target"></p>

Click here to see the full HTML so far

**IDS MATTER!**

Make sure you are matching the id values exactly as shown above. This will become important when we run the tests later!

* Open main.component.ts. This is the TypeScript file that defines the MainComponent class. This is where the application logic will go. You will see that it contains the following code:

[INSERT CODE BLOCK]

In Webz, components are defined as classes that extend the WebzComponent class. The MainComponent class is the main component of the application, and it is the component that is displayed on the page. The constructor method is called when an instance of the class is created. In this case, we are calling the super method with the html and css variables. This is how we tell Webz to use the HTML and CSS files that we created. The html and css variables are just strings that contain the contents of the HTML and CSS files, imported from the files themselves.

* Add a new private string field to the MainComponent class called myText, with the initial value "Hello from the TypeScript side!". This field will hold the text that we want to display on the page.

private myText: string = "Hello from the TypeScript side!";

* On its own, just creating a field doesn’t do anything interesting. We need to *bind* the field to the HTML so that the text is displayed on the page. We can do this by using the BindValue decorator. Add the following code directly above the myText field:

@BindValue("example-target")

The @BindValue decorator is used to bind a field to an element in the HTML; in general, a decorator can enhance the functionality of a field or a method of a class. This decorator takes a single argument, which is the id of the element that we want to bind to. In this case, we are binding the myText field to the element with the id example-target.

* In order to use the decorator, we must also import the BindValue decorator from the @boots-edu/webz package. Modify the existing import statement at the top of the file to include the BindValue decorator:

import { WebzComponent, BindValue } from '@boots-edu/webz';

* If you save the file now, you should be able to see the changes in the live webpage. The text "Hello from the TypeScript side!" should be displayed on the page. This is because the myText field is bound to the example-target element in the HTML. When the field changes, the text on the page will automatically update.

Click here to see the full TypeScript file so far.

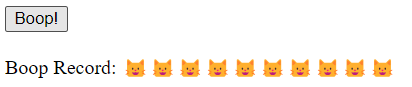
* To test if your website is working correctly, you can run the tests that are included with the starter repo. To do this, open a second terminal (click Terminal in the top menu, and then choose New Terminal) in VS Code and run the following command:

npm run test main

This will run the main.test.ts file that is included with the starter repo. If everything is working correctly, you should see a message that says All tests passed!. If you see any errors, double-check your code to make sure everything is correct.

## 2) Boop Button Component

Now we’ll create a second component that lives in the MainComponent. This component will have a button that, when clicked, will add a bit of text to the screen. We’ll use this to demonstrate how to handle events in Webz. We will call this component the “Boop Button” since our example will be about booping cat’s noses. In the image below, every time the button is clicked, another cat head (“🐱”) will be added to the record.



* Begin by creating the new Boop Button component. This requires running a terminal command from within the src/app directory. Navigate to the terminal in VS Code and run the following commands:

cd src/app/

webz c boop-button

Click here to see the expected output for this command.

**ERROR: WEBZ IS NOT INSTALLED**

If you get an error that says webz: command not found or The term webz is not recognized, you need to install Webz globally. You can do this by running npm install -g @boots-edu/webz-cli in the terminal.

If you are on a Mac, you may need to use sudo npm install -g @boots-edu/webz-cli instead. You may need to input your password to install the package.

If you are using PowerShell for Windows, you might get an error that says running scripts is disabled on this system. At the top right of your terminal in VS Code, there should be a little + button with a dropdown next to it (looks like v). If you click the dropdown, select Command Prompt, and that may allow you to run the command instead.

The c in the command is short for component, and boop-button is the name of the new component. This will create a new folder in the src/app directory with the necessary files for the new component. That folder will have the following files: boop-button.component.ts, boop-button.component.html, boop-button.component.css, and boop-button.component.test.ts.

* The component has been created, but it is not yet being used in the MainComponent. Open main.component.ts and add the following import statement at the top of the file:

import { BoopButtonComponent } from './boop-button/boop-button.component';

This imports the BoopButtonComponent class from the boop-button.component.ts file. We will use this class to create an instance of the Boop Button component in the MainComponent.

* Define a new private field in the MainComponent class called boopButton of type BoopButtonComponent. This field will hold an instance of the Boop Button component that should be created when the MainComponent is instantiated.

private boopButton: BoopButtonComponent = new BoopButtonComponent();

* Although the MainComponent class now has a boopButton field, it is not yet being displayed on the page. To do this, we need to add the HTML for the Boop Button component to the main.component.html file. Open main.component.html and add the following line after the p tags, but before the final </div>:

<div id="boop-button"></div>

Click here to see the main.component.html file so far

* Next, we have to actually add the component instance from the private field to the content of the MainComponent. Open main.component.ts and add the following line to the constructor method, after the call to super:

this.addComponent(this.boopButton, "boop-button");

This takes the component stored in the boopButton field and adds it to the MainComponent’s content. The second argument is the id of the element in the HTML that the component should be added to. In this case, we are adding the Boop Button component to the element with the id boop-button. This allows us to place the Boop Button component in the correct location in the HTML; if we didn’t specify an id, the component would just be added to the end.

Click here to see the full main.component.ts file so far

* When you save these files, the Boop Button component should now be displayed on the page. However, there is not yet any button or functionality in the Boop Button component, just the default text that comes from creating a new component. Open the boop-button/boop-button.component.html file and replace the existing content with the following code:

[INSERT CODE BLOCK]

This HTML creates a button with the text “Boop!” and a paragraph that displays the boop record, with a span tag holding the actual boops data. A span tag is meant to hold a short, inline snippet of text (inside of a paragraph tag or div). The boops span is initially empty, but it will be updated every time the button is clicked. We have also added an id attribute to the button (booper) and the span element (boops), which we will use to bind the button click event and the boop record text. Make sure you get the names of the id attributes exactly right, as they will be used in the TypeScript code to bind the elements.

* Now, we need to add some functionality to the Boop Button component. Open the boop-button/boop-button.component.ts file and add a new private string field named boops to the BoopButtonComponent class, initially an empty string. You also need to import the BindValue decorator from the @boots-edu/webz package to *bind* the boops field to the span with the same id in the HTML.

[INSERT CODE BLOCK]

* When clicked, the button will not do anything yet, because we have not added any event handling to the Boop Button component. We need to add an event handler to the button that will update the boops field every time the button is clicked. To do this, we need to add a new method to the BoopButtonComponent class that will be called when the button is clicked. We will decorate that button with a special Click decorator (which must be imported from @boots-edu/webz. Add the following method to the BoopButtonComponent class:

[INSERT CODE BLOCK]

## CHOOSE YOUR OWN EMOJI

You are free to replace the cat head emoji with any other character or text you like; the tests are flexible.

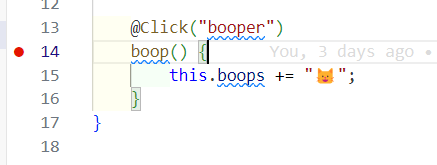
The @Click decorator is used to bind a method to an event on an element. In this case, we are binding the boop method to the Click event on the button with the id booper. This means that every time the button is clicked, the boop method will be called. The boop method appends a cat head emoji to the boops field.

Click here to see the full boop-button.component.ts file so far:

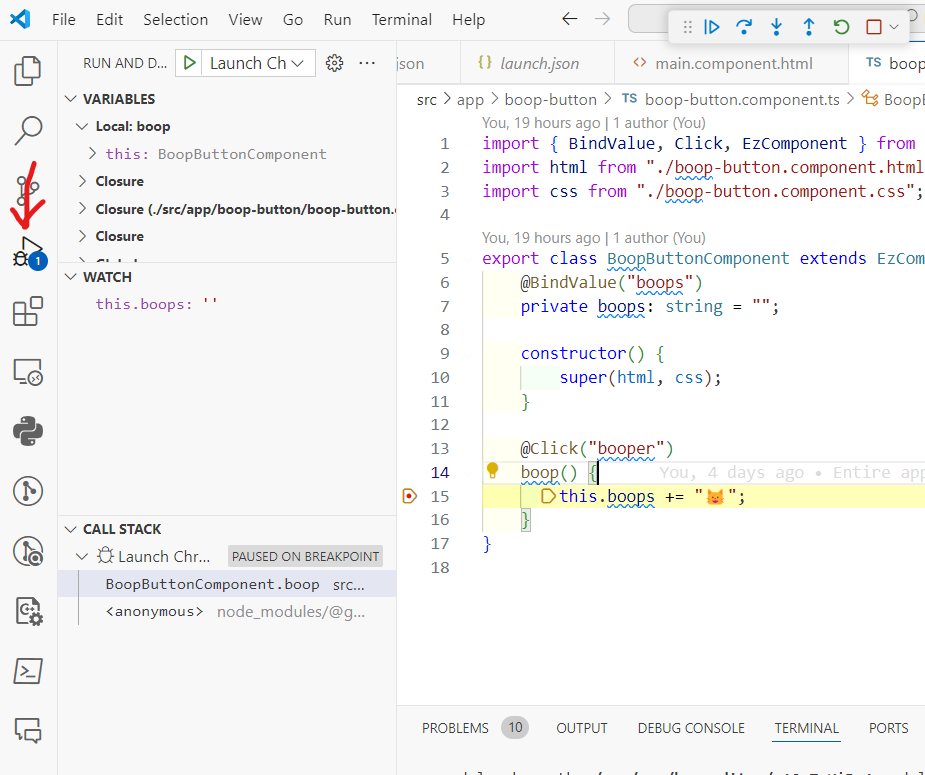
* Save the files and check the live webpage. You should see the Boop Button component with a button that says “Boop!” and a paragraph that displays the boop record. Every time you click the button, a new cat head emoji should be added to the boop record. This demonstrates how to handle events in Webz.
* To test if your website is working correctly, you can run the tests that are included with the starter repo. To do this, open a terminal in VS Code and run the following command:

npm run test boop

* This is a good time to try out the debugger in VS Code. You can set a breakpoint in the boop method in the BoopButtonComponent class to see the current values of the variables. To do this, click on the left margin of the editor window next to the line of code where you want to set the breakpoint. A red dot will appear, indicating that a breakpoint has been set. When you click the button on the live webpage, the code will pause at the breakpoint, and you can inspect the values of the variables. Set a breakpoint next to the boop method as shown below:



Click the boop button, and execution will pause inside of the boop method!



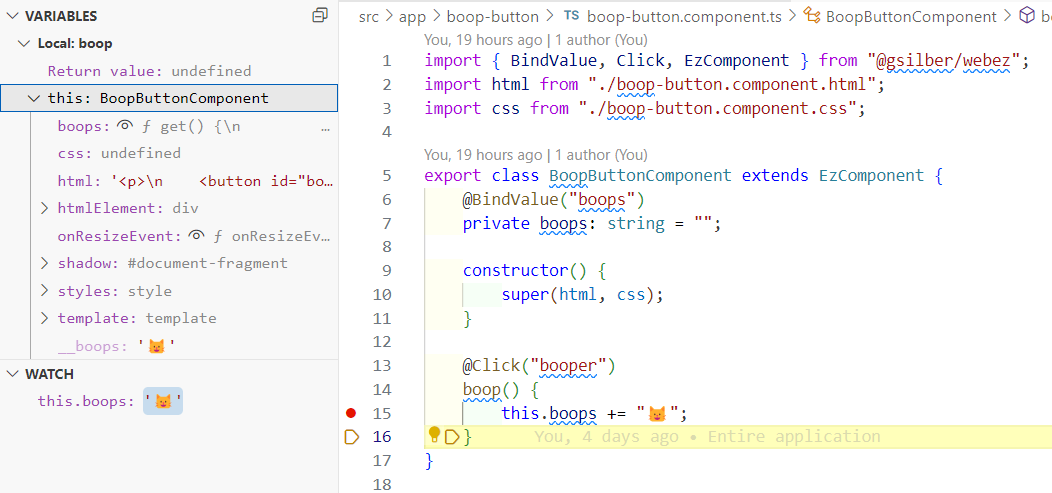
The left pane will be the debugger, accessible via the Debug tab in the sidebar (the bug on top of a triangle icon).

* The top box is the variables in the current scope (and any enclosing scopes—don’t worry about those just yet, though you’re free to poke around if you are curious).
* The middle box is the “Watch” box, where we can add specific variables and expressions to watch (we’ve added the this.boops expression to monitor).
* The bottom box is the call stack, which shows the current function calls that have led to the current point in the code.

There is a small bar in the top right of the screenshot (probably in the top center of your screen) with controls for stepping through the code. From left to right, they are:

* Continue (play button): Continue running the code until the next breakpoint.
* Step Over (arrow pointing right): Run the next line of code.
* Step Into (arrow pointing down): If the next line of code is a function call, step into the function.
* Step Out (arrow pointing up): Finish running the current function and return to the calling function.
* Restart (circular arrow): Restart the debugger from the beginning.
* Stop (square): Stop the debugger.

Click the “Step Over” button to run the next line of code.



If you expand the this object in the variables pane, you can see the current value of the boops field. The output may look a little confusing, saying something like f get() {\n - this is a quirk of how Webz-decorated fields look in the debugger. To see the value properly, you must click the little eyeball symbol next to the field name.

To make things a little easier, you can also add the this.boops expression to the watch box to monitor the value of the boops field as you step through the code. Hover over the WATCH box and click the + button to add a new watch expression. Type this.boops into the box and press Enter. You should see the current value of the boops field displayed in the watch box.

**HOVER OVER VARIABLES**

You can also hover over variables in the code to see their current values, while you are stepping through code. There are many other little features in the debugger that can be very helpful, so feel free to explore!

## 3) Simple Calculator Component

Our next component will be a simple calculator. This calculator will have two input fields for numbers, a select box for the operation, and a button to calculate the result. The result will be displayed on the page. We will use this component to demonstrate how to bind values to input fields and select boxes, and how to handle their associated events in Webz.

Simple Calculator

* Begin by creating the new simple calculator component. Once again, this requires running a terminal command from within the src/app directory. Navigate to the terminal in VS Code. Most likely, you are already in the src/app directory, but if not, run cd src/app/ and then run the following commands:

webz c simple-calculator

Just like last time, this will create a new folder in the src/app directory with the necessary files for the new component. The folder will have the following files: simple-calculator.component.ts, simple-calculator.component.html, simple-calculator.component.css, and simple-calculator.component.test.ts.

* As before, we have to import the new component into the MainComponent, create an instance of the component, and add it to the content of the MainComponent. Open main.component.ts, import the SimpleCalculatorComponent class, and add a new private field to the MainComponent class called calculator of type SimpleCalculatorComponent. Then, add the component to the content of the MainComponent in the constructor method.

Click here to see what the main.component.ts file should look like when you have done this!

* Similarly, you must also add in a new div element with the id calculator to the main.component.html file. This will be the location where the simple calculator component will be displayed on the page. Open main.component.html and add the following line after the boop-button div, but before the final </div>:

<div id="calculator"></div>

Click here to see the main.component.html file when you have done this correctly.

* Once those files are saved, we can see the simple calculator component on the live webpage, although it will still just have the default content that comes from creating a new component. Open the simple-calculator/simple-calculator.component.html file and replace the existing content with the following code:

[INSERT CODE BLOCK]

Let’s break down all the new elements in the simple calculator component:

* The first element is an input tag with the type attribute set to number and an id attribute set to first-number. This is where the user will input the first number for the calculation. The type attribute specifies the type of input field; we could have used text, password, or other types as well.
* The second element is another input tag, with similar settings, but with an id attribute set to second-number. This is where the user will input the second number for the calculation.
* The third element is a select tag with the id attribute set to operation-select. This is a dropdown box that allows the user to select the operation they want to perform. The option tags inside the select tag represent the different options in the dropdown box. Each option tag has a value attribute that specifies the actual value of the option when it is selected. The text inside the option tag is what is displayed to the user.
* The fourth element is a button tag with the id attribute set to calculate-button. This is the button that the user will click to perform the calculation. The text inside the button tag is what is displayed on the button.
* The fifth element is a span tag with the id attribute set to result. This is where the result of the calculation will be displayed.
* Next, we need to add some TypeScript functionality to the simple calculator component. Open the simple-calculator/simple-calculator.component.ts file and add a new private field to the SimpleCalculatorComponent class called firstNumber of type number, initially set to 7. You also need to import the BindValueToNumber decorator from the @boots-edu/webz package to *bind* the result field to the span with the same id in the HTML. Note that we are using BindValueToNumber instead of BindValue because the input fields are of type number!

[INSERT CODE BLOCK]

When saved, the firstNumber field should be bound to the first-number input field in the HTML. This means that the value of the firstNumber field will be displayed in the input field. However, this is a one-way binding; if the user changes the value in the input field, the firstNumber field will not be updated. We will add that functionality next.

* We need to add a new method to the SimpleCalculatorComponent class that will be called every time the value in the first-number input field changes. We will decorate this method with a special Input decorator (which must be imported from @boots-edu/webz, along with another class named ValueEvent). Add the following method to the SimpleCalculatorComponent class:

[INSERT CODE BLOCK]

**WHERE TO PLACE?**

Put this method in the SimpleCalculatorComponent class, fully after the constructor. In general, we recommend putting the fields before the constructor, and the methods after the constructor.

The @Input decorator is used to bind a method to an “Input” event on an input element. In this case, we are binding the onFirstNumberChange method to the Input event on the input field with the id first-number. This means that every time the value in the input field changes, the onFirstNumberChange method will be called. The onFirstNumberChange method takes an event parameter of type ValueEvent, which contains the new value of the input field. The + operator is used to convert the value to a number, since the value is always a string. As long as the user inputs a valid number, this will work correctly.

* Now we need to do the same thing for the second-number input field. Add a new private field to the SimpleCalculatorComponent class called secondNumber of type number, initially set to 3. Then, add a new method to the SimpleCalculatorComponent class that will be called every time the value in the second-number input field changes. As before, we will decorate this method with a special Input decorator. Add the following code to the SimpleCalculatorComponent class:

[INSERT CODE BLOCK]

* Next, we need to add a new private field to the SimpleCalculatorComponent class called operationSelect of type string, initially set to "add." This field will hold the value of the selected operation from the dropdown box. We also need to import the BindValue decorator from the @boots-edu/webz package to *bind* the operationSelect field to the select box with the same id in the HTML.

[INSERT CODE BLOCK]

* We need to add a new method to the SimpleCalculatorComponent class that will be called every time the value in the operation-select box changes. We will decorate this method with a special Change decorator (which must, as always, be imported!). Add the following method to the SimpleCalculatorComponent class:

[INSERT CODE BLOCK]

The @Change decorator is used to bind a method to a “Change” event on a select element. In this case, we are binding the onOperationChange method to the Change event on the select box with the id operation-select. The event parameter is once again a ValueEvent, which contains the new value of the select box. This value is then stored in the operationSelect field directly, without needing to convert it to a number.

* We need to add a new private field to the SimpleCalculatorComponent class called result of type number, initially set to 0. This field will hold the result of the calculation. This field will be bound to the result span in the HTML.

@BindValueToNumber("result")

private result: number = 0;

* Finally, we need to add a new method to the SimpleCalculatorComponent class that will be called every time the button is clicked. We will decorate this method with the Click decorator. Add the following method to the SimpleCalculatorComponent class:

[INSERT CODE BLOCK]

The calculate method gets the values of the firstNumber and secondNumber fields, performs the calculation based on the selected operation, and stores the result in the result field. In this case, we are only performing addition, so you will need to modify this method to handle the other operations as well. Add an if statement to check the value of the this.operationSelect field and perform the appropriate calculation based on the selected operation.

Click here to see the full simple-calculator.component.ts file when this is done correctly.

* Save the files and check the live webpage. You should see the simple calculator component with two input fields for numbers, a dropdown box for the operation, a button to calculate the result, and a span to display the result. You can input numbers into the input fields, select an operation from the dropdown box, and click the button to perform the calculation. The result should be displayed in the span.
* Run the tests for the simple calculator component to make sure everything is working correctly. Open a terminal in VS Code and run the following command:

npm run test calculator

**SAVE, COMMIT, AND PUSH**

If you haven’t saved, committed, and pushed recently, you should probably do so now. This will ensure that your changes are saved and backed up on GitHub.

## 4) Box Editor Component

Our final component will be a box editor component. This component will allow the user to create a box around an image with a specified padding, margin, and background color. The user will be able to input these values into input fields and select boxes, and the box will be displayed on the page. We will use this component to demonstrate how to bind values to style properties, and show a little bit about the CSS “box model.”



* Begin by creating the new box editor component. Once again, this requires running a terminal command from within the src/app directory. Navigate to the terminal in VS Code. Most likely, you are already in the src/app directory, but if not, run cd src/app/ and then run the following commands:

webz c box-editor

Just like last time, this will create a new folder in the src/app directory with the necessary files for the new component.

* As before, we have to import the new component into the MainComponent, create an instance of the component, and add it to the content of the MainComponent. Open main.component.ts, import the BoxEditorComponent class, and add a new private field to the MainComponent class called box of type BoxEditorComponent. Then, add the component to the content of the MainComponent in the constructor method to a target id "box".
* Edit the main.component.html file to include a new div element with the id box, just like we did for the other components.
* With the component added to the content of the MainComponent, we can see the box editor component on the live webpage. However, it will still just have the default content that comes from creating a new component. Open the box-editor/box-editor.component.html file and replace the existing content with the following code:

[INSERT CODE BLOCK]

There’s a lot of new stuff in this HTML, so let’s break it down:

* The first element is a div tag with the class frame. This is the outer frame of the box that will be created around the box editor, and we are going to make it explicitly visible by adding some CSS styling to it. To do so, we have to attach the class frame to the div tag. More on that soon.
* Inside the inner div tag, we have three label tags, each with a for attribute that corresponds to the id of an input field or select box. This is a best practice for accessibility, as it allows screen readers to associate the label with the input field. It also makes it easier to use in general, since clicking on the label will focus the input field.
* The first label tag is for the padding input field, which is an input tag with the type attribute set to number and an id attribute set to padding-input. This is where the user will input the padding value for the box. The padding is the space around the image inside the box.
* The second label tag is for the margin input field, which is similar to the padding input field, but with an id attribute set to margin-input. This is where the user will input the margin value for the box. The margin is the space around the box itself, to keep it away from other HTML elements.
* The third label tag is for the background color select box, which is a select tag with the id attribute set to background-select. This is a dropdown box that allows the user to select the background color of the box. The option tags inside the select tag represent the different options in the dropdown box. Each option tag has a value attribute that specifies the actual value of the option when it is selected. The text inside the option tag is what is displayed to the user.
* The br tags are used to create line breaks between the input fields and select box, to make the form easier to read.
* Finally, the img tag is used to display an image inside the box. The src attribute is set to \_\_\_, which is a placeholder for the actual image URL. The id attribute is set to image, and the class attribute is set to frame. The class attribute is used to attach the frame class to the image, which will make it visible inside the box.
* We need to choose an image to display inside the box. You can use any image you like, although we recommend one that is not too large. We used a picture of our dog Ada. Save the image to the assets directory and replace the \_\_\_ in the img tag with the name of the image file. For example, if the image is named ada.jpg, the img tag should look like this:

<img src="assets/ada.jpg" id="image" class="frame" />

**THE SRC MATTERS**

The src attribute of the img tag should point to the correct location of the image file. If the image is not displayed, double-check the path to the image file. Make sure you put the file in the assets directory and that the path in the src attribute is correct, and does not have unnecessary slashes (/). Also double check that you have the right file extension (.jpg, .png, etc.).

* Before we add the TypeScript functionality to the box editor component, we need to add some CSS styling to make the box frame visible. Open the box-editor/box-editor.component.css file and add the following CSS code:

[INSERT CODE BLOCK]

This is the first CSS we’ve written, so let’s break it down:

* The #image selector is used to style the image inside the box. By using a # symbol, we can explicitly refer to a specific id on the page. Then, inside of the curly braces, we can set rules by writing key: value; pairs. In this case, we are setting the width and height of the image to 100px. You can adjust these values to make the image larger or smaller, depending on your preference.
* The .frame selector is used to style the outer frame of the box. By using a . symbol, we can refer to any occurrence of a class on the page (as specified by the class attribute). In this case, we are setting the border of the frame to 1px solid black. This will create a thin black border around the box. You can adjust the 1px value to make the border thicker or thinner, and you can change the black value to any other color you like.

CSS is very powerful and allows you to style the page in a lot of different ways. That is all the CSS class styling we will do, but next we will add the TypeScript functionality that will affect the styling of the box dynamically.

* Open the box-editor/box-editor.component.ts file and add a new private field to the BoxEditorComponent class called padding of type number, initially set to 0. You also need to import the BindValueToNumber decorator from the @boots-edu/webz package to *bind* the padding field to the padding-input input field in the HTML. You then also need to create a new method (onPaddingChange) to handle the Input event on the padding-input input field. Refer to the code you used in the simple calculator component to help you with this.
* Repeat this for the margin input field. Add a new private field to the BoxEditorComponent class called margin of type number, initially set to 0. Bind the margin field to the margin-input input field in the HTML. Create a new method (onMarginChange) to handle the Input event on the margin-input input field.
* Just changing the padding and margin values won’t be enough to see the changes on the page. We need to bind the padding and margin values to the actual CSS properties of the box frame. To do this, we need to use a special BindStyleToNumberAppendPx decorator that will bind the padding and margin values to the padding and margin CSS properties of the image (these style attributes are measured in pixels, so the decorator appends "px" to the numbers automatically). Add the following code to the BoxEditorComponent fields:

[INSERT CODE BLOCK]

Now you should be able to change the padding and margin values in the input fields, and see the changes reflected in the box frame on the page. Try making the padding and margin values larger or smaller to see how it affects the box.

* Next, we need to add a new private field to the BoxEditorComponent class called background of type string, initially set to "red." You also need to import the BindValue decorator from the @boots-edu/webz package to *bind* the background field to the background-select select box in the HTML. Create a new method (onBackgroundChange) to handle the Change event on the background-select select box
* .
* To make the background color change whenever we change the background field, we need to bind the background field to the background-color CSS property of the image. To do this, we need to use a special BindStyle decorator that will bind the background field to the backgroundColor CSS property of the image. Add the following code to the BoxEditorComponent fields:

@BindStyle("image", "backgroundColor")

@BindValue("background-select")

background: string = "red";

**THE** BACKGROUNDCOLOR **PROPERTY**

The CSS property is actually called background-color, but Webz refers to it as backgroundColor to match the TypeScript naming style. This is a common convention in JavaScript and TypeScript, where hyphens from the CSS/HTML side are replaced with camelCase.

Click here to see the full box-editor.component.ts file when this is done correctly.

* Save the files and check the live webpage. You should see the box editor component with input fields for padding and margin, a select box for the background color, and an image inside a box frame. You can input values into the input fields, select a background color from the dropdown box, and see the changes reflected in the box frame on the page.

You should also now run the tests for the box editor component to make sure everything is working correctly:

npm run test box

We have only begun to scratch the surface of what we can do with Webz:

* There are many other CSS properties we can manipulate, like borders, shadows, and animations.
* We can bind to other types of events, including timers.
* We can create more complex components with multiple elements and interactions.
* We can use other HTML elements and CSS properties to create more complex layouts and designs.

## 5) Deploy Your Site

For now, this is a good place to stop. You have learned how to create components, bind values to elements, handle events, and style elements with Webz. You have also learned how to use TypeScript and CSS to create dynamic and interactive web pages. You can now build on this knowledge to create more complex and interesting web applications. But before we finish, let’s deploy your site!

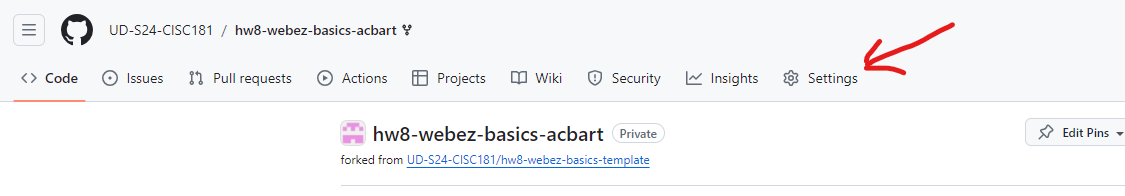
* In order to let you build your site locally (despite the tests originally failing), we modified one of the build files a little bit. To deploy your site, you need to revert this change. Open the tsconfig.json file in the top-level of your project folder, and change line 13 to become:

 "include": ["./src/\*\*/\*", "./wbcore/\*\*/\*", "./Jest/\*\*/\*", "./test/\*\*/\*"],

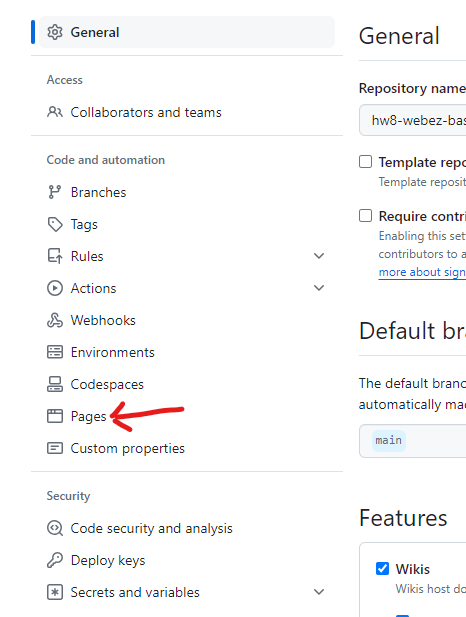
**EDITING BUILD FILES**

We won’t normally ask you to edit your build files; this is a special case just to make it easier to get started on the assignment.

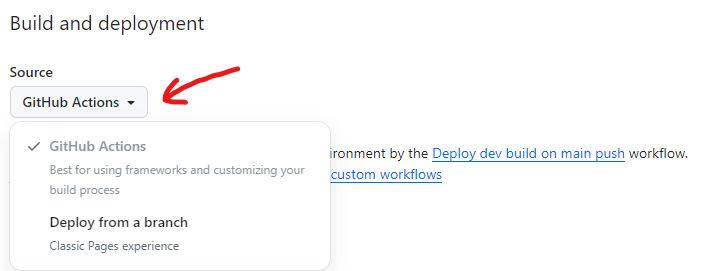
* Make sure you save all the files, commit your changes, and push them to GitHub.
* Next, you need to enable GitHub Pages for your repository. Go to the repository on GitHub, click on the “Settings” tab.



* Scroll down to the “GitHub Pages” section.



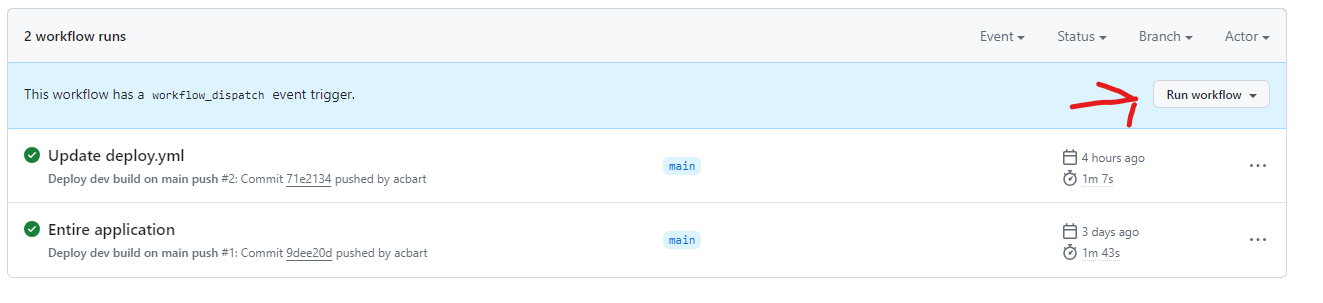
* In the source dropdown, select “GitHub Actions.”



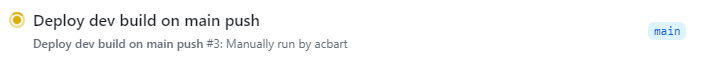
* Go to the “Actions” tab and you should see a “workflow” running. This workflow will build and deploy your site to GitHub Pages. Once the workflow is complete, you should see a link to your site at the top of the page.



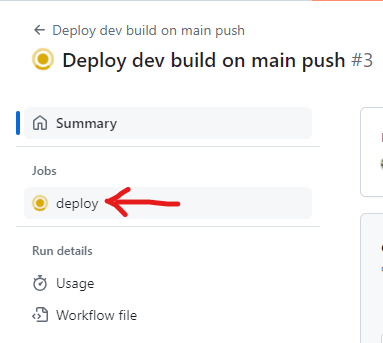
If the workflow doesn’t seem to be running, click “Deploy dev build on main push” and then click “Run workflow.” This will manually trigger the workflow to run, although you may have to reload the page to see it.



You can check the progress of a workflow by clicking on it:

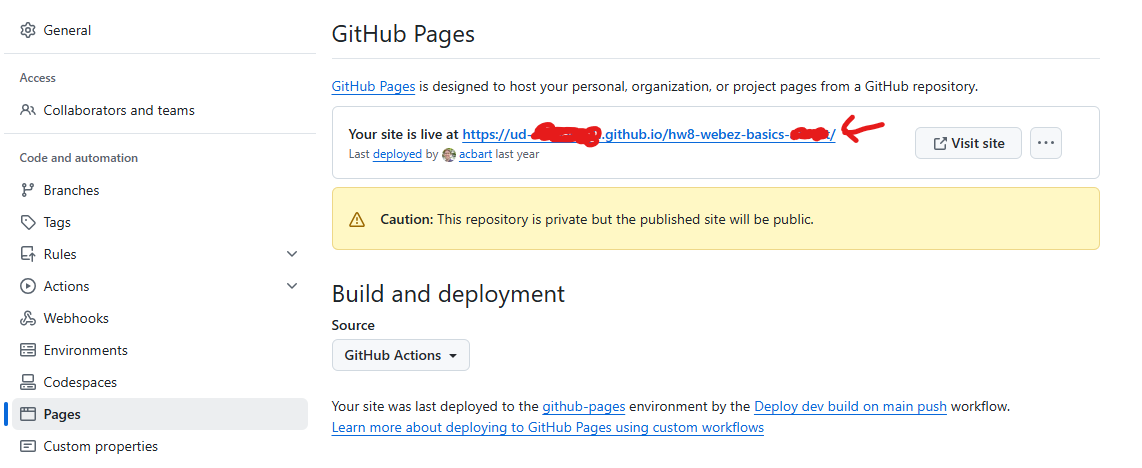


Click on the “deploy” button on the left sidebar to see the details of the deployment.



Assuming nothing goes wrong during deployment, you should see checkmarks next to all the steps.

You can find the URL of your live site by going back to the “Settings” tab and scrolling down to the “GitHub Pages” section. The URL should be displayed there.



## 6) Submission

Once you have completed the tutorial and deployed your site, you can submit on GradeScope. If you have any questions or issues, please don’t hesitate to ask for help!

In addition to passing our tests, you will also be graded on the successful deployment of your site. If the site is not deployed, you will not receive credit for the assignment. The TAs and instructors will review your site, your tests, and your code to ensure that you have completed the assignment correctly.

## Next Step

Next we’ll learn more advanced features of Webz and how to use them in Advanced Webz.

# Chapter 10 - Advanced Webz

# Dynamic Components

## Key Idea

We can create and attach components dynamically in order to create complex applications.

## 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.

[INSERT CODE BLOCK]

[INSERT CODE 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.

[INSERT CODE BLOCK]

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 (count) 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.

[INSERT CODE BLOCK]

[INSERT IMAGE]

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.

[INSERT IMAGE]

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.

## Working Example

Here we pull all of the code for our dynamic component so that you can work with it and see it in action.

main.component.ts

[INSERT CODE BLOCK]

main.component.html

[INSERT CODE BLOCK]

main.component.css

[INSERT CODE BLOCK]

LineItem.component.html

<p>LineItem Component</p>

LineItem.component.ts

[INSERT CODE BLOCK]

LineComment.component.html

<p>LineComment Component</p>

LineComment.component.ts

[INSERT CODE BLOCK]

As you can see, we can dynamically add (or remove) components from our site as needed based on user input.

## 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.

## Next Step

Next we’ll learn about component communication in Webz Events.

# Webz Events

## Key Idea

We can pass events between components so that our components can communicate.

## Component Hierarchy

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 hierarchies, we can have each parent notify its child down the hierarchy 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 information 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 the 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 the 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 a notifier that does not emit data, just a notification.
* The second line creates a notifier that emits a number.
* The third line creates a notifier that emits a string.
* The fourth line creates a notifier that emits an instance of a custom class
* The fifth line creates a 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. The type of parameter specifies the type expected for the notify method. Using the wrong type will be an error.

this.event.notify();

this.event2.notify(1);

this.event3.notify("hello");

this.event4.notify(new SomeClass());

this.event5.notify(["hello", "notifier", "world"]);

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(...).

[INSERT CODE BLOCK]

* this.event2.notify(4) called in the child would print 5 from the parent.
* this.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 section. First we need to define it:

[INSERT CODE BLOCK]

[INSERT CODE BLOCK]

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

* The public property commentChange is of type Notifier and emits a string.
* The method now calls notify with the new value.

And that’s it, our child class now emits its subscriptions whenever the user types and this will happen for each line comment we create. We could 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. The following code would appear in MainComponent.

[INSERT CODE BLOCK]

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 logging the comment. Let’s create an array and counter to store the comments and update them when they change.

[INSERT CODE BLOCK]

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.

### How Does Notifier Work?

Notifier is a simple class that holds a list of functions that are called when the notify method is called. The subscribe method adds a function to the list. The notify method calls each function in the list with the data passed to notify. The data is of the type specified in the type of parameter of the notifier.

[INSERT CODE BLOCK]

## Working Example

Here we pull all of the code for our event notifier together so that you can work with it and see it in action.

main.component.ts

[INSERT CODE BLOCK]

main.component.html

[INSERT CODE BLOCK]

main.component.css

[INSERT CODE BLOCK]

LineItem.component.html

[INSERT CODE BLOCK]

LineItem.component.ts

[INSERT CODE BLOCK]

LineComment.component.html

<div class="line-comment">

Comment:

<input id="comment" type="text" />

</div>

LineComment.component.css

[INSERT CODE BLOCK]

LineComment.component.ts

[INSERT CODE BLOCK]

## 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.

## Next Step

Next we’ll learn about Webz dialogs and popups.

# Webz Dialogs

## Key Idea

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. Webz provides two methods for doing this:

* 1) Popup: Creates a popup window with a title, some text, and buttons that returns the text on the button through a notifier.

2) Dialog: Creates a popup window whose content is determined by a component. These can be created with the cli (webz dialog myDlg).

## 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).

[INSERT CODE BLOCK]

We can call this method to show a dialog box:

[INSERT CODE BLOCK]

Popup returns a notifier that emits the text of the button pressed. We can subscribe to the returned value to be notified when the popup closes.

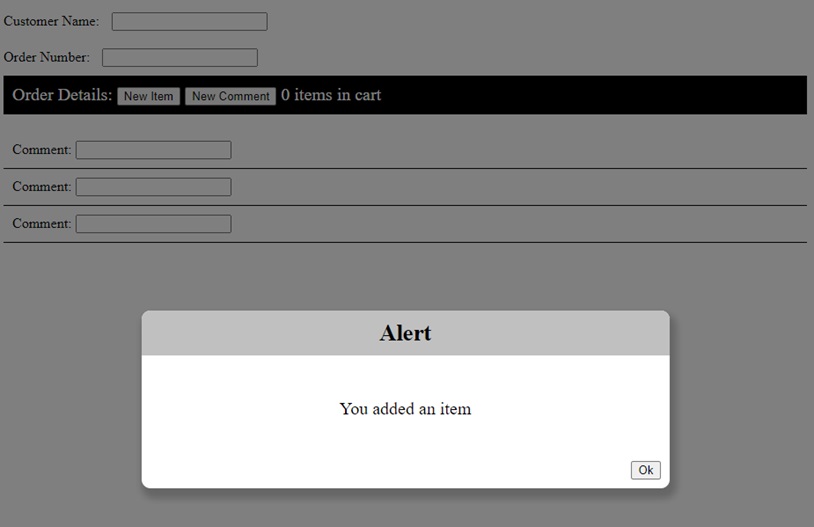
Let’s examine this in detail:

* 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) is the text inside the popup.
* title (optional) is the title for your popup, displayed at the top.
* buttons (optional) is 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) is an optional CSS class string to style the buttons. This allows you to optionally attach a CSS class to the button for styling.
* Returns is a notifier that emits the label of the pressed button.

Back to our point of sale example, we can use a popup to notify the user that a comment was added.

[INSERT CODE BLOCK]

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 we needed to know that the popup was closed.



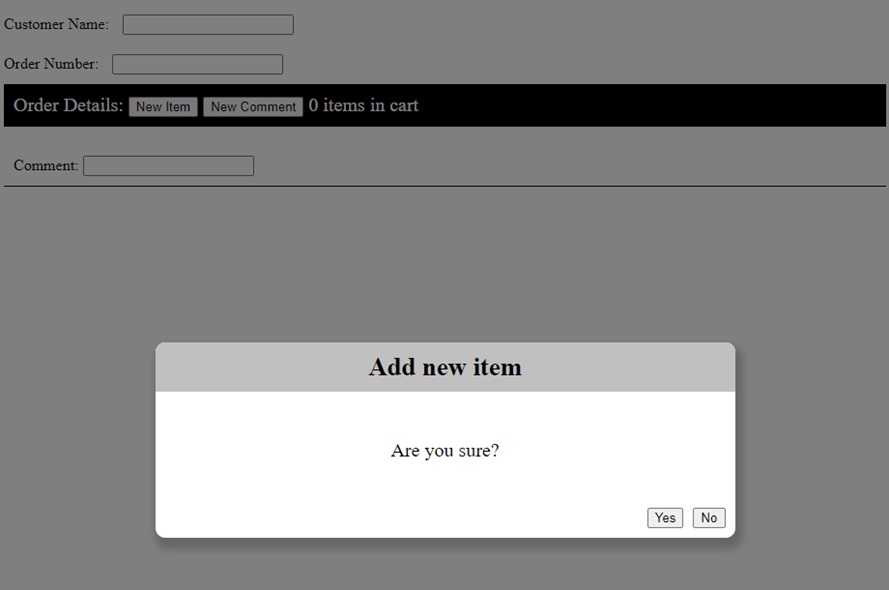
Notice how the popup greys out the underlying website. You cannot click buttons or enter text while the popup is open. Once it is closed, the gray background goes away and the rest of the page will again accept input. With this screen, the only thing I can do is click ok.

Let’s look at a more complex example:

[INSERT CODE BLOCK]

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 Notifier 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 popup is closed.



## 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).

[INSERT CODE BLOCK]

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 its 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…”

[INSERT CODE BLOCK]

[INSERT CODE BLOCK]

Note: We are not showing the typescript for PlsWaitDialog as it is unmodified.

In the parent, we create a property for our dialog and add it to the component.

[INSERT CODE BLOCK]

When we want the dialog we can simply display it while some time consuming task is occurring, then hide it after.

[INSERT CODE BLOCK]



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.

## Working Example

Here is a working example of the please wait dialog in action. Note that we never close the dialog as we don’t have anything time consuming to do. In the next sections we will learn about Timers which allow us to execute code after a delay.

## 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.

## Next Step

Next we’ll learn about Webz Timers.

# Webz Timers

## Key Idea

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.

[INSERT CODE BLOCK]

And style it so that it has a fixed width and will appear inline after the words “Please Wait.”

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

## 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.

## Next Step

Next we’ll build an image editor in Webz by following the Webz Advanced Tutorial.

# Webz Advanced Tutorial

## Key Idea

Notifier is a powerful class for facilitating inter-component data transfer in Webz.

This tutorial will walk you through building a more complex Webz application with multiple components and dynamic content. This will involve creating a new component, binding data between components, and handling events. By the end of this tutorial, you will have a better understanding of how to build more complex applications with Webz.

# The Image Editor

Our goal is to create a simple image editor to edit pixel art. We will create a component that displays an image and allows the user to edit the image by changing the color of individual pixels. We will also create a color picker component that allows the user to select a color to use for editing the image.



## 0) Setup

* Use the GitHub classroom link provided in the original assignment on Canvas to create your own copy of the starter repo.
* Clone the repo to your local machine in an appropriate directory.
* Open the directory in VS Code, as you normally do.
* Run npm install in the VS Code terminal to install the dependencies.

npm install

* Run npm run start in the terminal to start the development server. This may take a few seconds to compile the code and start the server. If you need to stop the server, you can press Ctrl+C in the terminal.

You can click here to see what the output looks like for us when the server starts successfully.

* Although we could now open your website in chrome at the localhost url http://localhost:8080, we will use the integrated debugger in VS Code. Activate this by pressing F5 on your keyboard (or selecting the Run tab from the top menu and then clicking Start Debugging). This will open a new browser window with your application running. The debugger has a bunch of useful features, like setting breakpoints and inspecting variables - we’ll talk more about them later on.

**DEBUGGING IN VS CODE**

You can only activate the debugger if you have the server running. If you close the server, you will need to start it again before you can use the debugger.

* You should now be able to see your website. Now we can start making the actual application.

## 1) Colors

Our goal is to eventually create a simple image editor, but that all begins with a representation of colors. We will create a Color class that represents a color as an RGB value. This will not be a WebzComponent, but a simple TypeScript class. We’ll eventually create other components that use this class.

* Create a new file in the src/app directory called color.ts. Create and export a class called Color with three private properties: red, green, and blue. These properties should be numbers that represent the red, green, and blue values of the color. The constructor should take three parameters, one for each property, in this order.
* Define a method in the class named toString() that consumes nothing and returns a string. This method should return a string that represents the color in the format rgb(red, green, blue). For example, if the color has red=255, green=0, and blue=0, the toString() method should return the string rgb(255,0,0).
* Define a method in the class named asNumbers() that consumes nothing and returns an array of three numbers. This method should return a newly created array with the red, green, and blue values of the color in that order.

### Palettes

Colors are usually described as a combination of red, green, and blue values. Each value can range from 0 to 255, where 0 means no color and 255 means the maximum amount of color. For example, rgb(255, 0, 0) is a bright red color, while rgb(0, 255, 0) is a bright green color.

However, writing out three numbers every time we want to represent a color can be cumbersome. Our application will use an additional color representation that is more user-friendly: a palette. A palette is an index of colors (usually an array of colors) that can be used to represent colors in a more user-friendly way. Specifically, we will use a palette of 9 colors (0-8) to represent colors in our image editor:

|  |  |  |  |
| --- | --- | --- | --- |
| **Palette Index** | **Color Triple** | **Name** | **Preview** |
| 0 | [0, 0, 0] | Black |  |
| 1 | [255, 255, 255] | White |  |
| 2 | [255, 0, 0] | Red |  |
| 3 | [0, 255, 0] | Green |  |
| 4 | [0, 0, 255] | Blue |  |
| 5 | [255, 255, 0] | Yellow |  |
| 6 | [255, 0, 255] | Magenta |  |
| 7 | [0, 255, 255] | Cyan |  |
| 8 | [128, 128, 128] | Gray |  |

This way, an image can be written as a 2D array of palette indices, where each index represents a color in the palette. For example, the following 2D grid represents a 5x5 image of a smiley face (remember that 0 is black and 5 is yellow):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 5 | 5 | 5 | 5 |
| 5 | 0 | 5 | 0 | 5 |
| 5 | 5 | 5 | 5 | 5 |
| 5 | 0 | 5 | 0 | 5 |
| 5 | 0 | 0 | 0 | 5 |

This would translate to the following 2D number array in TypeScript:

[INSERT CODE BLOCK]

Which is much more compact than the Color[][] representation would be.

* To make it possible to support palettes, we will need a PALETTE array that holds the colors of the palette. This array should be a constant array of number[] triples (arrays of length 3), where each index corresponds to the index of the color in the palette. For example, PALETTE[0] should be black ([0, 0, 0]), PALETTE[1] should be white ([255, 255, 255]), and so on. You will need to export the PALETTE array so that the test can access it.

[INSERT CODE BLOCK]

* Create and export a function named makeColor that consumes a number and returns a Color. This function should take a number index and return a new Color object with the red, green, and blue values from the PALETTE array at the given index. For example, makeColor(0) should return a new Color object with the red, green, and blue values from PALETTE[0]. Additionally, if the index is out of bounds, the function should throw an error with an appropriate message (e.g., "Invalid color index").
* Finally, create and export a function named convertPalette that consumes a 2D number array (representing a palette-indexed image) and returns a 2D Color array. This function convert each of the palette-indexed numbers to a Color object using the makeColor function. For example, convertPalette(smileyFace) should return a 2D array of Color objects that represent the smiley face image.

**2D ARRAYS**

Notice that both the PALETTE and the smileyFace arrays are 2D arrays (an array of arrays). However, they are different types of arrays. The PALETTE array is an array of number[] triples, while the smileyFace array is an array of number[] arrays (with the outer array representing rows and the inner arrays representing individual columns within a row). This is because the smileyFace array represents a 2D grid of palette indices, while the PALETTE array represents a list of colors. Don’t get these confused as you work with them!

If everything has been done correctly so far, you should be able to run the Color tests and see them pass. You can run the tests by running the following command in the terminal:

npm run test color

If the tests fail, then you can run them in interactive mode. This will make the tests run whenever you save a file, and you can see the output in the terminal. To run the tests in interactive mode, run the following command:

npm run watch color

## 2) Pixels

Now that we have a way to represent colors, we can create a PixelComponent class that represents a single pixel in an image. A pixel is a color at a specific location in an image. We will create a Pixel class that has a color property (a Color object), size (a number), and read-only x and y properties (numbers that represent the location of the pixel in the image).

These pixels are going to appear in multiple places in our editor. We’ll make their size adjustable so that they can be used in different contexts (the preview area, the color picker, and the image editor itself). Since they’re going to be used in multiple places, we’ll make them a WebzComponent so that we can reuse them easily.

* Run the following command in the src/app/ directory to create a new component called pixel:

webz component pixel

A new directory called pixel will be created in the src/app/ directory. This directory will contain the TypeScript, HTML, and CSS files for the Pixel component: pixel.component.ts, pixel.component.test.ts, pixel.component.html, and pixel.component.css.

This class is going to be used by many other classes, but we don’t actually place it on the screen until we have a grid or toolbar to place it in. However, during development, you may find it easier to add a PixelComponent to the Main component so that you can see it in the browser. You can do this by adding the following code to the Main component:

[INSERT CODE BLOCK]

* To further test your PixelComponent, we created some tests that are specifically for the PixelComponent. Since you just created the PixelComponent, these tests are not yet in the pixel.component.test.ts file. You’ll need to add them yourself. Copy all of the following code and paste it into the pixel.component.test.ts file, making sure to replace any existing code in that file.

[INSERT CODE BLOCK]

If you read over the tests above, you will see that we have two tests. The first test checks if the pixel can change color, and the second test checks if the pixel can change size. You can run these tests by running the following command in the terminal:

npm run watch pixel

The tests will fail until we implement the PixelComponent.

* In the pixel.component.html file, create a div element with the id pixel. This element will represent the pixel on the screen.
* In the pixel.component.ts file, add four private fields to the class:
* color (a Color object that represents the color of the pixel): Choose an appropriate default value (e.g., white, black, purple).
* size (a number that represents the size of the pixel in actual screen pixels): Choose an appropriate default value (e.g., 10, 20, 50).
* x (a number that represents the x-coordinate of the pixel in the image): Initialize this member variable through a constructor parameter.
* y (a number that represents the y-coordinate of the pixel in the image): Initialize this member variable through a constructor parameter.
* Define the methods getX, getY, and getColor that consume nothing and return the x, y, and color properties of the pixel. Do not overthink these methods; they should be simple one-liners that return the appropriate property.
* Define the methods setColor (which consumes a Color and returns nothing but updates the color property of the pixel) and setSize (which consumes a number and returns nothing but updates the size property of the pixel). Again, do not overthink these methods; they should be simple one-liners that update the appropriate property. These are public methods for other parts of the application to safely change the color and size of the pixel. Notice that we do NOT have a setX or setY method; the location of the pixel should be immutable once it is created.
* We need the color of the PixelComponent class to update the background color of the div element in the HTML. To do this, we need to use the BindStyle decorator on the color member variable. This decorator will bind the color property to the background-color style of the div element in the HTML. Add the following code to the pixel.component.ts file:

[INSERT CODE BLOCK]

The first parameter to the BindStyle decorator is the id of the div element in the HTML. The second parameter is the style property that we want to bind to (in this case, backgroundColor). The third parameter is an anonymous function that takes the color property of the PixelComponent class and returns a string. This function should convert the Color object to a string that represents the color in the format rgb(red, green, blue).

* We also need to update the size of the div element in the HTML to match the size of the pixel. To do this, we need to use the BindStyleToNumberAppendPx decorator on the size member variable. This decorator will bind the size property to the width and height styles of the div element in the HTML. This decorator automatically converts the number to a string and appends px to the end. You will need to attach the decorator to the size member variable TWICE, once for the width style and once for the height style. Both times, you will be binding to the pixel id in the HTML.

The tests should now pass. If they do not, you may need to debug your code to find the issue. If you are having trouble, don’t hesitate to ask for help! Don’t be afraid to experiment with your testPixel in the Main component to see how it behaves (but remember to remove it before you finish).

## 3) Toolbar

With our PixelComponent ready, we can now create a Toolbar component that will contain a set of PixelComponent representing the colors in the palette. The Toolbar component will allow the user to select a color from the palette to use in the image editor, changing a currently active color (also represented on the screen by a PixelComponent).

* Run the following command in the src/app/ directory to create a new component called toolbar:

webz component toolbar

A new directory called toolbar will be created in the src/app/ directory. This directory will contain the TypeScript, HTML, and CSS files for the Toolbar component: toolbar.component.ts, toolbar.component.test.ts, toolbar.component.html, and toolbar.component.css.

* In the toolbar.component.html file, you can use the following HTML to create a div element with the id swatches (styled to be in a row). This element will contain the Pixel components representing the colors in the palette. You can also create a div element with the id active to display the currently active color. We put a horizontal rule (<hr>) at the bottom of the toolbar to separate it from the rest of the page, but you can remove that if you like. We could also have styled the swatches in the CSS file instead of the HTMLfile, or left the swatches in a vertical column.

[INSERT CODE BLOCK]

* In the main.component.html file, create a new div element with the id toolbar. This element will represent the palette toolbar on the screen. Then, in the main.component.ts file, create an instance of a ToolbarComponent and add it to the MainComponent using the addComponent method, with the target id "toolbar". You will need to import the ToolbarComponent class at the top of the file. At this point, the toolbar should appear on the screen, but it will not contain any colors. Assign the instance of the ToolbarComponent to a field in the MainComponent class named toolbar so that you can access it later.
* In the toolbar.component.ts file, add a public constant field named DEFAULT\_COLOR (a Color) that represents the default color of the active pixel. You can choose any color you like for the default color, but we recommend black (0, 0, 0) and not white (255, 255, 255) so that it is visible on the screen.
* In the toolbar.component.ts file, add a private field to the class called active (a PixelComponent object that represents the currently active color). Initialize this field to a new PixelComponent object with the x and y properties set to 0 (the top-left corner of the image). In the ToolbarComponent constructor, use the addComponent method to add the active pixel to the component. You will need to use the setSize method to set the size of the active pixel to 30 (or another appropriate size), and the setColor method to set the color of the active pixel to the DEFAULT\_COLOR. At this point, the active color should appear on the screen!
* Since the active field is private, other classes will not be able to ask the ToolbarComponent for the active color. To allow other classes to access the active color, define a public method in the ToolbarComponent class named getActiveColor that consumes nothing and returns a Color. This method should return the color of the active pixel.
* The ToolbarComponent class will contain a set of PixelComponent objects representing the colors in the palette. We’ll need to create one pixel for each color in the palette, set them to the appropriate size and color, and then add them to the ToolbarComponent. To do this, we will need to use the PALETTE array that we created earlier. In the ToolbarComponent constructor, iterate over the PALETTE array and create a new PixelComponent object for each color in the palette (you can use the makeColor function too, if you want). Set the size of each pixel to an appropriate value (e.g., 20), set the color of each pixel to the color from the PALETTE array, and add each pixel to the ToolbarComponent using the addComponent method. You will also need to create a private swatches field in the ToolbarComponent class to store these newly created PixelComponent objects. make sure you adding the component to the screen (addComponent) AND to the swatches array (push).

### Clickable Pixels

The Toolbar component contains a set of PixelComponent objects representing the colors in the palette. These pixels should be clickable, allowing the user to select a color from the palette to use in the image editor. When a pixel is clicked, the active pixel should change to the color of the clicked pixel. We’re going to need to use the Click decorator to make this happen.

However, the active pixel should not be clickable (what would it even do?). Since we want to have a version of Pixels that can be clicked and a version that cannot (but is otherwise the same), a simple way to do this is to create a new class that extends PixelComponent and adds the Click decorator. We’ll call this class ClickablePixelComponent.

* Do NOT run the webz component command to create the ClickablePixelComponent. Instead, create a new file in the src/app/pixel/ directory called clickable-pixel.component.ts. In this file, create a new class called ClickablePixelComponent that extends PixelComponent. This class should extend the PixelComponent class:

[INSERT CODE BLOCK]

Our new ClickablePixelComponent class is now a subclass of the PixelComponent class. This means that it has all the properties and methods of the PixelComponent class, but it can also have additional properties and methods that are unique to the ClickablePixelComponent class. The main difference between the PixelComponent and the ClickablePixelComponent is that the ClickablePixelComponent will have the Click decorator on it so that it supports clicking.

* In the clickable-pixel.component.ts file, define a new function named onClick that consumes nothing and returns nothing. This function should be empty for now. Next, attach the Click decorator with the id pixel to the onClick function. This will make the onClick function run whenever the div element with the id pixel is clicked.

[INSERT CODE BLOCK]

But what on earth should go into the onClick function? We want the active pixel in the ToolbarComponent to change to the color of the clicked pixel. However, the ClickablePixelComponent class does not have access to the active pixel in the ToolbarComponent. When using Composition (which is the type of relationship between the ToolbarComponent and the ClickablePixelComponent), the ClickablePixelComponent should not have direct access to the active pixel in the ToolbarComponent. Toolbars can know about their pixels, but pixels should not know whether they are in a toolbar (since they could be in other places too).

To solve this problem, we will use a Notifier. As previously described in the [Events](https://boots-edu.github.io/textbook/text/10-webz-advanced/events.html) chapter, the Notifier class is a special class that can be composed in a class to notify other classes of changes. In this case, we will create a Notifier in the ClickablePixelComponent class that will notify the ToolbarComponent when the pixel is clicked. The ToolbarComponent will then update the active pixel to the color of the clicked pixel.

A Notifier has two halves:

* An inner class will have a Notifier instance and will call its notify method when something happens.
* An outer class will have a reference to the inner class, and can subscribe a function to the Notifier instance to be called when notify is called.
* In this case, we’ll have our ClickablePixelComponent be the inner class and the ToolbarComponent be the outer class. The ClickablePixelComponent will call notify when it is clicked, and the ToolbarComponent will subscribe to the ClickablePixelComponent to update the active color when notify is called. The Notifier itself will be stored in a field named clickEvent in the ClickablePixelComponent class. Here is the new definition of the ClickablePixelComponent class:

[INSERT CODE BLOCK]

* All that is left is to update the ToolbarComponent to subscribe to the ClickablePixelComponent when it is created. In the ToolbarComponent constructor, where you originally created the PixelComponent inside of a for loop, you should now instead create a ClickablePixelComponent and subscribe to its clickEvent. When the clickEvent is called, the ToolbarComponent should update the active pixel to the color of the clicked pixel. You will need to import the ClickablePixelComponent class at the top of the file. Here is an example of how you might do this:

[INSERT CODE BLOCK]

We have to subscribe to the clickEvent right after we create the ClickablePixelComponent, since that is when we have the reference to the ClickablePixelComponent that we need to subscribe. The subscribe method takes a function that consumes a ClickablePixelComponent and returns nothing. This function should update the active pixel to the color of the clicked pixel. The subscribe method will be called whenever the clickEvent.notify method is called in the ClickablePixelComponent.

If you are having trouble with the syntax of anonymous functions, you should consider reviewing the [Anonymous Functions](https://boots-edu.github.io/textbook/text/8-testing/anonymous.html) chapter. We will use anonymous functions with Notifier a lot when building interactive web applications.

**INHERITANCE != NOTIFIERS**

We used inheritance here to create a new class that is almost the same as the PixelComponent class, but with the added ability to be clicked. This is a common use of inheritance. However, inheritance is not required to create a Notifier. Inheritance is used to create new classes that are similar to existing classes, while a Notifier is a special class that can be composed in a class to notify other classes of changes. We could have put all the code into one PixelComponent class and just let the inner class do nothing. Make sure you understand that we do not have to create a subclass just to use Notifiers!

You should now be able to click on the pixels in the toolbar to change the active color. If you are having trouble, don’t hesitate to ask for help! Make sure you are passing all the tests for the Palette, and that you understand how Notifier works, before you move on.

## 4) Preview

The preview will display the image that the user is currently editing. The preview will contain a 2D grid of PixelComponent objects representing the pixels in the image. The user will eventually be able to click on the pixels in the editor to change the pixels in a preview area. For now, we’ll just make the preview component display a grid of pixels. This will share functionality with the editor component, so we’ll have a general GridComponent that can be used in both places.

* Run the following command in the src/app/ directory to create a new component called grid:

webz component grid

* In the grid.component.html file, you can use the following HTML with CSS styling:

[INSERT CODE BLOCK]

This HTML creates a div element with the id pixels that will contain the PixelComponent objects representing the pixels in the image. The div element is styled to display the pixels in a grid with a border and a background color. The flex-wrap: wrap property allows the pixels to wrap to the next line when they reach the edge of the container. The justify-content: space-between and align-content: space-between properties space the pixels evenly in the container. As long as we make the grid the right size, the pixels should be evenly spaced and fill the container, with an optional (purple) gap in between. You are free to change the background-color and the border to suit your design.

* In the main.component.html file, create a new div element with the id preview. This element will represent the preview area on the screen. Then, in the main.component.ts file, create an instance of a GridComponent and add it to the MainComponent using the addComponent method, with the target id "preview". You will need to import the GridComponent class at the top of the file. At this point, the preview area should appear on the screen, but it will not contain any pixels. Assign the instance of the GridComponent to a field in the MainComponent class named preview so that you can access it later.
* In the grid.component.ts file, add the following fields:
* gap (number): Represents the size of the gap between the pixels in the grid in pixels. This should be assigned via the first parameter of the constructor.
* zoom (number): Represents the zoom level of the grid (how big the pixels are). This should be assigned via the second parameter of the constructor.
* size (number): Represents the total size of the grid in pixels, initially zero. This will be recalculated and updated whenever the pixels field is updated.

You’ll need to update the constructor call in the MainComponent to pass in the gap and zoom values when creating the GridComponent (we recommend 0 and 20 for the preview, but you might start off with 1 and 32 so you can make sure you’ve got the gap size correct).

* Use the BindStyleToNumberAppendPx to bind the size field to the pixels div element’s width and height styles. This will ensure that the grid is the correct size on the screen. You will need to attach the decorator to the size member variable TWICE, once for the width style and once for the height style. Both times, you will be binding to the pixels id in the HTML.
* Now we need to create the PixelComponent objects that will represent the pixels in the image. In the GridComponent class, define a private field named pixels (a 2D array of PixelComponent objects) that represents the pixels in the image. Initialize this field to an empty 2D array. We’re going to need A) a way to load an entire initial image into the grid, and B) a way to update individual pixels in the grid. We’ll start with A.
* In order to setup the grid’s pixels, we’re going to need to create a pixel. Because there are so many steps involved in creating a pixel, let’s create a helper function to do this. In the grid.component.ts file, define a private method named addPixel that does all of the following:
* Consumes a number x and a number y (the location of the pixel in the grid), and a color (the Color of the pixel).
* Constructs a new PixelComponent object with the x and y properties set to the given x and y values.
* Sets the size of the pixel to the zoom property of the GridComponent.
* Sets the color of the pixel to the given color.
* Adds the pixel to the pixels field at the appropriate location in the 2D array.
* Adds the pixel to the pixels div element in the HTML using the addComponent method.
* Returns the newly created PixelComponent object.

**ROW-COLUMN FORMAT OF 2D ARRAYS**

A 2D array is an array of arrays. The outer array represents the rows of the grid, and the inner arrays represent the columns within a row. When you add a pixel to the pixels field, you will need to add it to the appropriate row and column in the 2D array. The x and y values represent the column and row of the pixel in the grid, respectively. A tricky part is that when iterating through the pixels array, you are first iterating through the y values (the rows) and then the x values (the columns). Then, when you are indexing into the pixels array, you will need to index into the y value *first* and then the x value. It is tempting to write expressions like this.pixels[x][y], but this will be transposed from what you expect. Make sure you are indexing into the pixels array correctly!

* Now you can add a public method to the GridComponent named loadImage that consumes a 2D array of Color objects and returns nothing. This method should iterate through the 2D array of Color objects and call the addPixel method for each color in the array. This will create a PixelComponent object for each color in the image and add it to the pixels field of the GridComponent. During the iteration, you must also create a new array for each row of the grid, push this array to the appropriate row of the pixels 2D array, and add the PixelComponent objects to that row array. Finally, after the iteration, you should update the size field of the GridComponent to be the size of the grid in pixels, using the following formula (replacing the terms with the appropriate expressions or variables):

size = number\_of\_rows \* (zoom + gap) - gap

Having trouble with `loadImage`?

* In your MainComponent class, define a new public constant member variable named DEFAULT\_IMAGE that is a 2D array of Color objects. This array should represent a simple image (e.g., a smiley face) that you can use to test the GridComponent. The image must be square and at least 5 pixels wide and tall. You can use the convertPalette function to convert a 2D array of palette indices to a 2D array of Color objects. For example, the smiley face would look like:

[INSERT CODE BLOCK]

* Call the new loadImage method inside of the MainComponent constructor, passing in your DEFAULT\_IMAGE constant, on your Preview’s GridComponent instance. This will load the default image into the grid when the page is loaded.

If you run your tests (npm run watch preview) at this point, you will fail one labeled The loadImage method correctly clears old pixels. You can see why if you try calling loadImage more than once. Instead of clearing out the old image, the new image is just added below the old one. You will need to add a new method to the GridComponent class to clear out the old image before loading a new one. This method is named clearPixels, takes no arguments, and should be called just before you start adding new pixels to the grid in loadImage. The clearPixels function is partially implemented for you below:

[INSERT CODE BLOCK]

You will need to replace the TODO comments with the appropriate code to remove the PixelComponent objects from the screen and clear out the pixels array.

* While we are here, let’s also provide a method named getImage that consumes nothing and returns a 2D array of Color objects. This method should iterate through the pixels field of the GridComponent and create a new 2D array of Color objects that represents the image in the grid. You can use the getColor method of the PixelComponent class to get the color of each pixel in the grid. This will be helpful for testing purposes (and would be essential if we were going to add functionality for saving the image).
* We’ve only got one more method for the GridComponent to implement. We need a way to update individual pixels in the grid. Add a public method to the GridComponent named setColorAt that consumes a number x, a number y, and a Color color, and returns nothing. This method should update the color of the pixel at the given x and y location in the grid to the given color. You will need to use the setColor method of the PixelComponent class to update the color of the pixel.

We can now see the image, although we cannot yet interact with it. You should be able to pass all the tests when you run npm run watch preview.

## 5) The Editor

We’re getting closer to the final product, but we are still missing our actual editor. Like the preview area, the editor will also have a GridComponent. But unlike the preview area, the editor will need to support clicking on the pixels to change their color. We could follow the same pattern that we did for pixels and extend the GridComponent class to create a ClickableGridComponent class. However, this time we’ll add the new functionality directly into the GridComponent class.

**WHEN TO EXTEND**

Knowing when it is worth it to extend a class is challenging. In this case, we are adding a new feature to the GridComponent class that is not present in the Preview area. This new feature is specific to the Editor area, so it would make sense to extend the class to avoid having unnecessary functionality in the Preview area. If we thought that we might later need clickable preview areas, we would be better off adding the functionality to the GridComponent class directly. In this case, we’re not extending the class because we want to make it clear that the notifier is not tied to inheritance. But making these kinds of decisions is a big part of software design!

* In the grid.component.ts file, define a new field named onPixelClick that is a Notifier<ClickablePixelComponent>. This Notifier will be used to forward the click events from the pixel objects in the grid to the EditorComponent.
* Modify the addPixel method to create an instance of a ClickablePixelComponent instead of a PixelComponent. Then, in the same method, subscribe to the clickEvent of the newly constructed ClickablePixelComponent with an anonymous function that calls the notify method of the onPixelClick Notifier with the ClickablePixelComponent object as an argument. Now, whenever someone clicks on a pixel, that will trigger the this subscription, which will in turn notify any subscribers of the onPixelClick Notifier.

**POLYMORPHIC PIXELS**

Although you needed to change the type of pixel you were creating to be a ClickablePixelComponent, you did not need to change the type of the pixels field itself. Because ClickablePixelComponent is a subclass of PixelComponent, you can store ClickablePixelComponent objects in a PixelComponent array. This is an example of polymorphism, where a subclass can be used in place of a superclass. This works in this situation because we are not relying on any additional functionality of the ClickablePixelComponent class, outside of the addPixel method (where the compiler still knows that the pixel is a ClickablePixelComponent).

* Return to the main.component.ts file and add a new field named editor to the MainComponent class. This field should be a GridComponent object that represents the editor area. Add the editor to the MainComponent using the addComponent method, with the target id "editor". You will need to import the GridComponent class at the top of the file. When constructing the GridComponent, we recommend using a gap of 1 and a size of 32 for the editor.
* Duplicate the loadImage call you previously used for the preview area, but this time call it on the editor field of the MainComponent class. This will load the default image into the editor when the page is loaded. Both the preview and the editor should now display the same image.
* Finally, we need to “wire” up the editor and preview area to handle their click events. Basically, when a pixel in the editor is clicked, we want to change the color of that pixel AND the corresponding pixel in the preview area, using the current active color from the toolbar. This requires information from three different components spread across the application, which means we must rely on the Notifier. Observe the class composition diagram below that shows the composition relationships between the components (note that we have not included the WebzComponent class, which is a parent class of all the components except Color, and we have also not shown the inheritance relationship between ClickablePixelComponent and PixelComponent):

[INSERT CODE BLOCK]

The only place where we can have the two grids and toolbar all talk to each other is their earliest common ancestor, which is the MainComponent. The MainComponent will need to subscribe to the onPixelClick Notifier of the editor GridComponent. When the onPixelClick Notifier is called, the MainComponent should update the color of the clicked pixel in the editor and the corresponding pixel in the preview area using their setColorAt methods. The MainComponent will need to get the x and y position of the clicked pixel (which is available as the parameter of the subscription function) and the active color from the toolbar ToolbarComponent. This is really only four lines of code (although Prettier may split it up into more lines):

Try to write the code yourself before looking at the solution!

Once you have the data flowing between these components, you should be able to pass all the tests when you run npm run watch editor. Congratulations! You have created a working image editor!

## 6) Deploy Your Site

Before we finish, let’s deploy your site!

In order to let you build your site locally (despite the tests originally failing), we modified one of the build files a little bit. To deploy your site, you need to revert this change. Open the tsconfig.json file in the top-level of your project folder, and change line 13 to become:

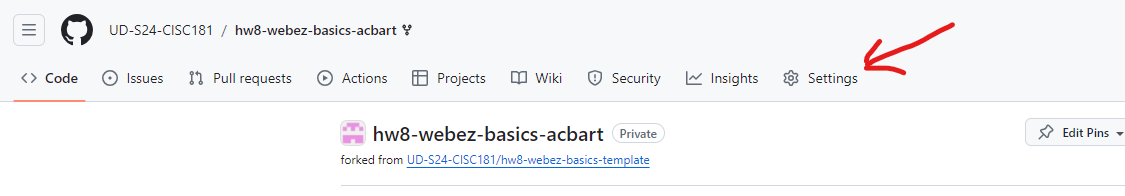
 "include": ["./src/\*\*/\*", "./wbcore/\*\*/\*", "./Jest/\*\*/\*", "./test/\*\*/\*"],

**EDITING BUILD FILES**

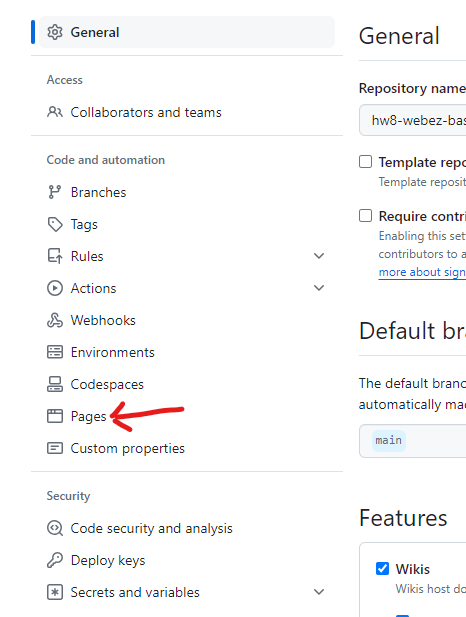
Once again, we won’t normally ask you to edit your build files; this is a special case just to make it easier to get started on the assignment.

Make sure you save all the files, commit your changes, and push them to GitHub.

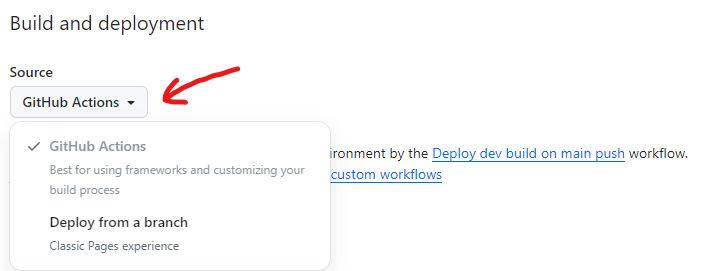
Next, you need to enable GitHub Pages for your repository. Go to the repository on GitHub, click on the “Settings” tab.



Scroll down to the “GitHub Pages” section.



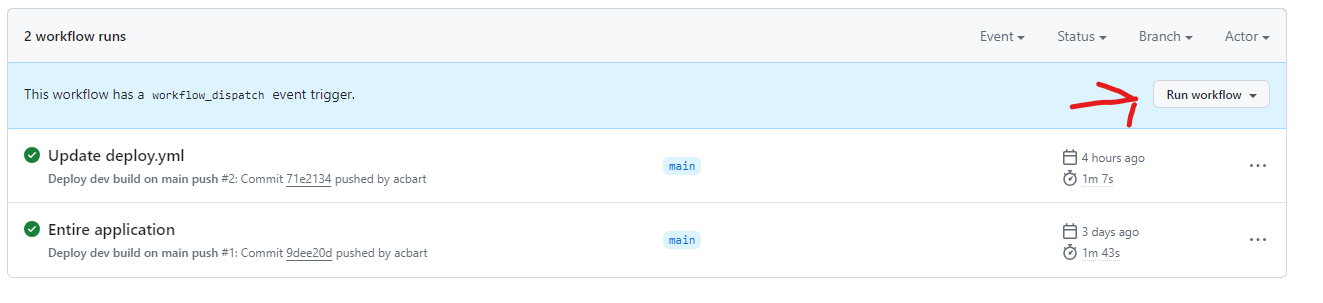
In the source dropdown menu, select “GitHub Actions.”



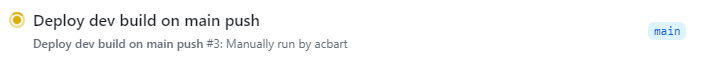
Go to the actions tab and you should see a “workflow” running. This workflow will build and deploy your site to GitHub pages. Once the workflow is complete, you should see a link to your site at the top of the page.



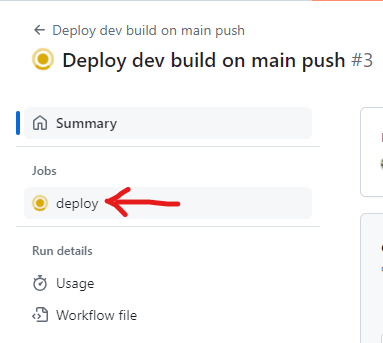
If the workflow doesn’t seem to be running, click “Deploy dev build on main push” and then click “Run workflow.” This will manually trigger the workflow to run, although you may have to reload the page to see it.



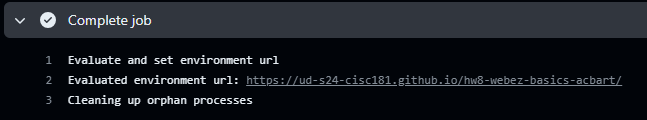
You can check the progress of a workflow by clicking on it:



Click on the “deploy” button on the left sidebar to see the details of the deployment.



Assuming nothing goes wrong during deployment, the final step can be expanded to get the URL of your live site. Click on the URL to visit your site!



If that URL is not visible, then you can also find the URL by going back to the “Settings” tab and scrolling down to the “GitHub Pages” section. The URL should be displayed there.

## 7) Submission

Once you have completed the tutorial and deployed your site, you can submit on GradeScope. If you have any questions or issues, please don’t hesitate to ask for help!

In addition to passing our tests, you will also be graded on the successful deployment of your site. If the site is not deployed, you will not receive credit for the assignment. The TAs and instructors will review your site, your tests, and your code to ensure that you have completed the assignment correctly.

## Next Step

Next we’ll learn more features of TypeScript and how to use them in Advanced TypeScript.

# Chapter 11 - Advanced TypeScript

# TypeScript Generics

## Key Idea

*Generics* allow for creation of reusable code 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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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 *generic* by adding a *type parameter* and using it as the type of the result parameter.

[INSERT CODE BLOCK]

Here we have added a *type parameter* (T), and we use that parameter to set the 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).

[INSERT CODE BLOCK]

We are not limited to a single type parameter. If we need more than one, we can specify multiple type parameters.

[INSERT CODE BLOCK]

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…

[INSERT CODE BLOCK]

…we 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.

[INSERT CODE BLOCK]

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 parameter’s* extends clause.

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

If a parameter is provided, the default is ignored. If no parameter is provided, then the type 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 look at the source code for it.

[INSERT CODE BLOCK]

* 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 technique to classes and methods so that our code works on various types of data.

## Next Step

Next we’ll learn about TypeScript Interfaces.

# TypeScript Interfaces

## Key Idea

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.

[INSERT IMAGE]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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 will not compile.

## Multiple Interfaces

A class cannot extend more than one class in typescript, but it can implement many interfaces.

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

## Using Interfaces

*Interfaces* have many uses, primarily:

* Describe the shape of data to guarantee that the data is in the right form.

[INSERT CODE BLOCK]

* 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, disparate objects can be used as if they are the same object so long as they implement a given interface.

## Next Step

Next we’ll learn about Union Types.

# Union Types

## Key Idea

*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.

[INSERT CODE BLOCK]

We might want it to take a number and add that many spaces to the front.

[INSERT CODE BLOCK]

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:

[INSERT CODE BLOCK]

We can apply this to other types as well—classes, interfaces, etc.

## Union types with classes

Consider these classes:

[INSERT CODE BLOCK]

We can union 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.

[INSERT CODE BLOCK]

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.

## Next Step

Next we’ll learn about Higher Order Methods.

# Chapter 12 - Higher Order Methods

# Higher Order String Methods

## Key Idea

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.

[INSERT CODE BLOCK]

We also saw this in the Webz library with the Notifier class’s subscribe method:

[INSERT CODE BLOCK]

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 they 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 straightforward 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:

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

If we want to call a member function instead, we can simply call it in the body of the anonymous function.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

These do not mutate the array in any way.

### The find and findIndex methods

The *find* method executes 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.

[INSERT CODE BLOCK]

*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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

*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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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 function 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);

console.log(sum);

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);

console.log(product);

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

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.

[INSERT CODE BLOCK]

We can even use it to combine map and filter in a single step.

[INSERT CODE BLOCK]

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 vals = [1, 2, 3, 4, 5];

let sum = vals.reduce((acc, val) => acc + val, 0);

console.log(sum);

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:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Acc | 0 |
| Val | 1 |
| Returns | 1 |

On the second call, the return value of the first call becomes the value of the accumulator.

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Acc | 1 |
| Val | 4 |
| Returns | 5 |

On the third call, the return value of the second call becomes the value of the accumulator.

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Acc | 5 |
| Val | 11 |
| Returns | 16 |

On the fourth call, the return value of the third call becomes the value of the accumulator.

|  |  |
| --- | --- |
| **Variable** | **Value** |
| 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 vals = [1, 2, 3, 4, 5];

let sum = vals.reduceRight((acc, val) => acc + val, 0);

console.log(sum);

Obviously, for the examples so far, this makes no difference (sum and product are commutative), but there are cases where it would.

Consider the following:

[INSERT CODE BLOCK]

This returns the first even number. If we use reduceRight instead of reduce, it would return the last even number in the list.

[INSERT CODE BLOCK]

This does not mutate the array in any way.

### The sort method

With no arguments, *sort* returns the elements in the array in ascending or alphabetical order.

let vals = [3, 2, 1, 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];

*//make a clone since sort is destructive*

let vals2 = Array.from(vals);

const ascending = vals.sort((a, b) => a - b);

const descending = vals2.sort((a, b) => b - a);

console.log(ascending);

console.log(descending);

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.

## Next Step

Next we’ll learn see how we could re-implement these higher-order methods ourselves in Example Implementation of Higher Order Methods.

# Example Implementation of Higher Order Methods

## Key Idea

A *higher order function* is a function that takes as an argument and/or returns a function.

## Overview

This page provides an example implementation of *higher order methods* in TypeScript. We will re-implement all of the higher-order array methods in our own custom List class.

Note that you do not have to implement these methods yourself - they are built into the JavaScript array class. This is just an explanation of *how* they work. Examples of using them are at the bottom of the page.

For more information about the built-in array methods, see the MDN documentation.

[INSERT CODE BLOCK]

## forEach Method

[INSERT CODE BLOCK]

## filter Method

[INSERT CODE BLOCK]

## some Method

[INSERT CODE BLOCK]

## every Method

[INSERT CODE BLOCK]

## map Method

[INSERT CODE BLOCK]

## reduce Method

[INSERT CODE BLOCK]

## rightReduce Method

[INSERT CODE BLOCK]

## find Method

[INSERT CODE BLOCK]

## findIndex Method

[INSERT CODE BLOCK]

## findLast Method

[INSERT CODE BLOCK]

## findLastIndex Method

[INSERT CODE BLOCK]

## sort Method

[INSERT CODE BLOCK]

## Usage Examples

The following is an example of how you could use our new List class:

[INSERT CODE BLOCK]

### More Examples

[INSERT CODE BLOCK]

## Summary

There is nothing special about these built-in methods, except that they are conveniently provided for us. We can re-implement them with the existing control structures we already learned.

## Next Step

Next we’ll learn about Recursion.

# Chapter 13 - Recursion

# Description and Definition of Recursion

## Key Idea

\*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.

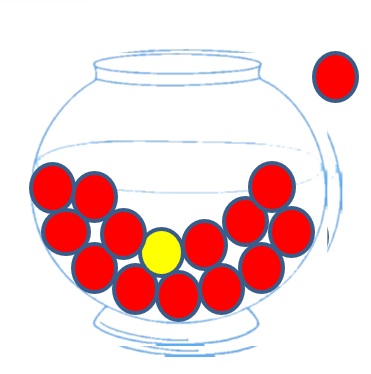
Consider the problem of teaching all of you a concept. We can restate this problem as

* 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 known 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*.

[INSERT CODE BLOCK]

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).

[INSERT CODE BLOCK]

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 problem of computing factorial.

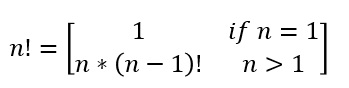
Factorial is defined as: *n! = n \* (n-1) \* (n-2) \* (n-3) \* … \* 1*

**5! = 5 \* 4 \* 3 \* 2 \* 1

We can restate this in terms of an easier instance of factorial: *n! = n \* (n - 1)!*

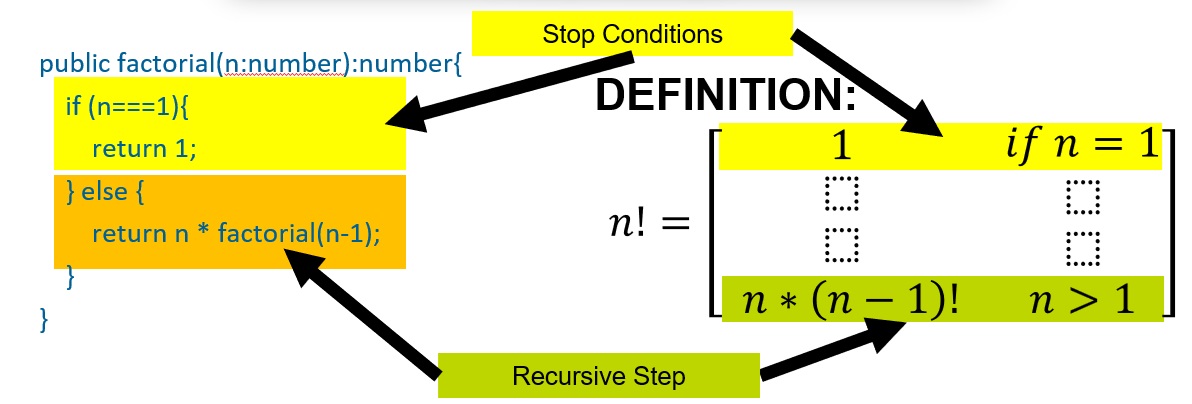
**5! = 5 \* 4!

Since we know that 1! is equal to 1, we can rewrite the definition as:



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:



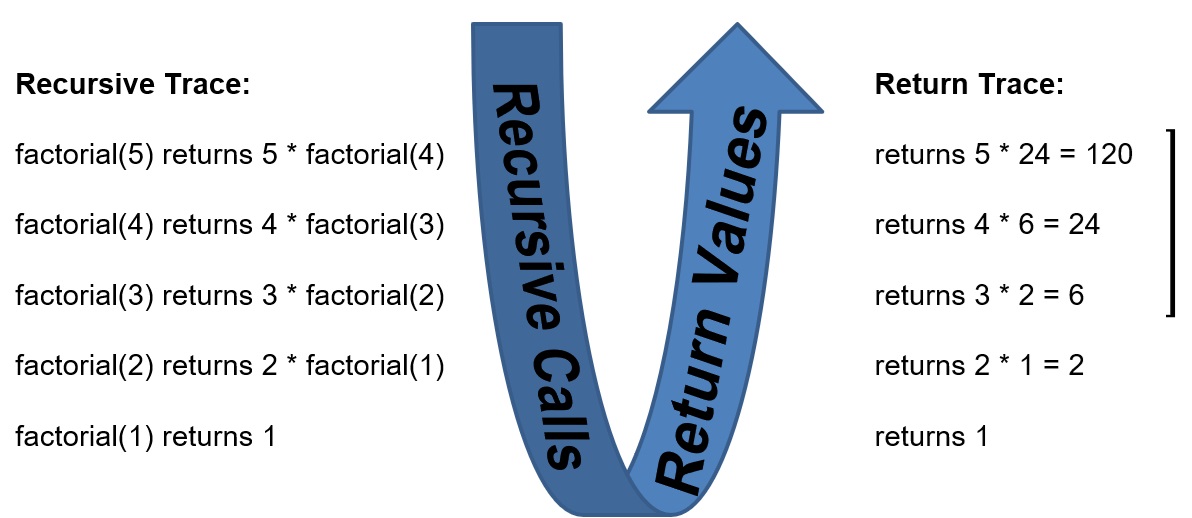
Let’s try it:

[INSERT CODE BLOCK]

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 contains 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* is:

* 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:

[INSERT CODE BLOCK]

Our recursive step simply calls itself on the correct half of the array:

[INSERT CODE BLOCK]

And the full search function:

[INSERT CODE BLOCK]

## 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.

## Next Step

Next we’ll learn about Trees.

# Trees

## Key Idea

A *tree* in computer science is a data structure that represents data in a parent/child relationship.

## Motivating Recursion

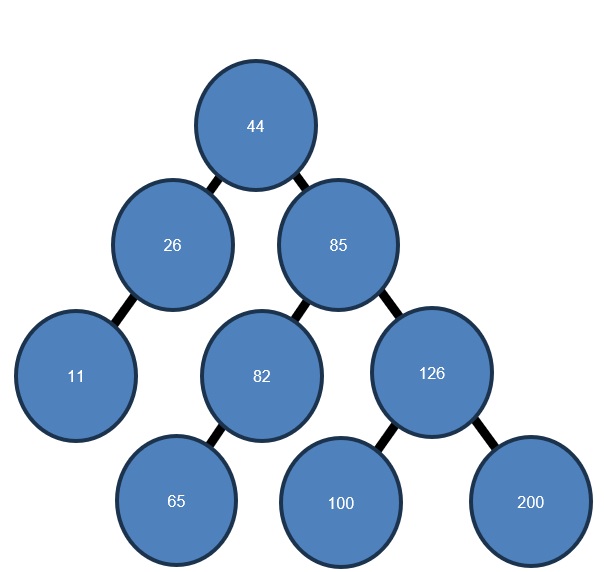
All of these would have been very easy to implement using a loop instead. One place where recursion is particularly useful is *trees*. *Trees* are a basic data structure that we can use to represent data in a parent child relationship.

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 subtree 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 subtree must be less than the value stored in the node
* Every number in the right subtree 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 would start by comparing 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 subtree.

|  |
| --- |
|  |
| *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 that 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).

[INSERT CODE BLOCK]

### 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.

[INSERT CODE BLOCK]

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 subtree, so we call treeSearch recursively to search that subtree.

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 subtree, so we call treeSearch recursively to search that subtree.

[INSERT CODE BLOCK]

### 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 subtree we would insert into is not empty, then we just insert into that (smaller) subtree.

[INSERT CODE BLOCK]

*Trees* are a common data structure in computer science and recursion is a much more natural way to deal with them.

### Complete Tree Example

[INSERT CODE BLOCK]

This example implements the *binary search tree* in the previous example, then searches it for 100.

If we inserted them in a different order, we would have gotten a different tree.

Thought question: What happens if we insert them in sorted order?

Answer:

[INSERT IMAGE]

## 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.

[INSERT CODE BLOCK]

For the search method, it is a little less straightforward. 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).

[INSERT CODE BLOCK]

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 program:

[INSERT CODE BLOCK]

## Summary

*Trees* are an important data structure in computer science. They allow us to store data in a structured way that represents 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.

# Appendix

This is the appendix. It contains additional information that is not part of the main text. It is here for reference and additional learning.

# Pulling from Upstream

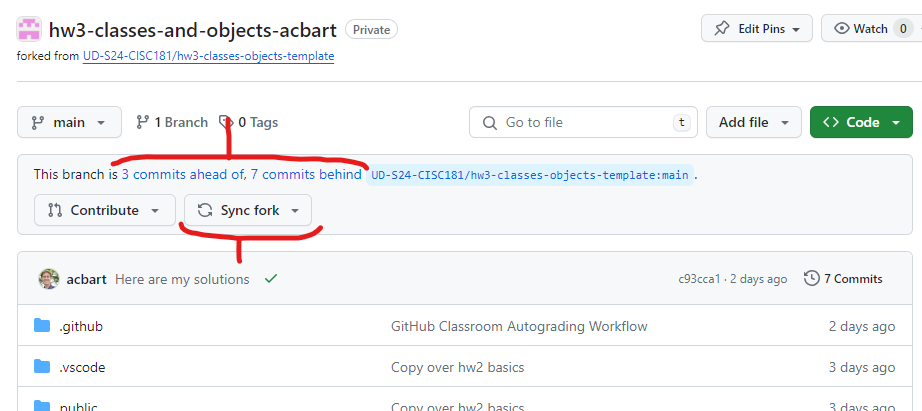
## Key Idea

When working with a forked repository, it is important to keep your fork up to date with the original repository. This is done by pulling changes from the original repository into your fork. The steps can be a little tricky the first time you do them.

Don’t panic, just follow this guide!

## Step 1: Sync Your Remote

Visit your remote repository for the assignment on the GitHub web interface. This should have your name in the repository title, like “UD-S24-CISC181/hw3-classes-and-objects-acbart.” If there are upstream changes to sync into your repository, you will see some message like the following:



If you have pushed your own changes, it will say that you have “N commits ahead of.” This is not a problem, and just means that you have done some work.

If there are upstream changes to pull, it will say that you have “N commits behind.”

You might have both, but you only have to do something if you “commits behind,” which should give you access to a “Sync fork” button.

Click the “Sync fork” button to reveal further options. If that button is not available, then congratulations! You do not have anything to pull in and you are up-to-date. You can stop reading this guide.

If the changes are easy to merge then it will offer you the ability to “Update branch” and you can go to [Step 3](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html#step-3-pulling-the-local-changes). If it offers “Open pull request,” then things are a little bit trickier and you need to go to [Step 2](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html#step-2-pulling-remotes-locally).

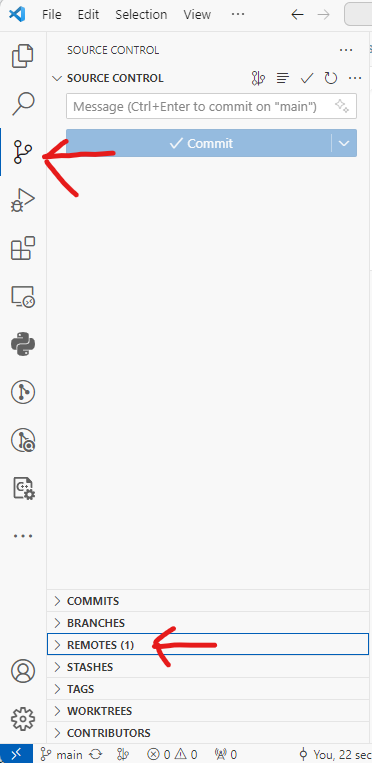
|  |  |
| --- | --- |
| **Merge Conflicts** | **No Merge Conflicts** |
| If you get the option to “Open pull request” because “this branch has conflicts that must be resolved.” That means you changed the files more than Git can handle automatically, and you will need to go to [Step 2](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html#step-2-pulling-remotes-locally). | Choose to “Update branch” and that will update your remote repository. Then go to [Step 3](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html#step-3-pulling-the-local-changes). |
| \ |  |

## Step 2: Pulling Remotes Locally

Remember, if you just used “Update branch,” then you can skip over this step and go directly to [Step 3](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html#step-3-pulling-the-local-changes).

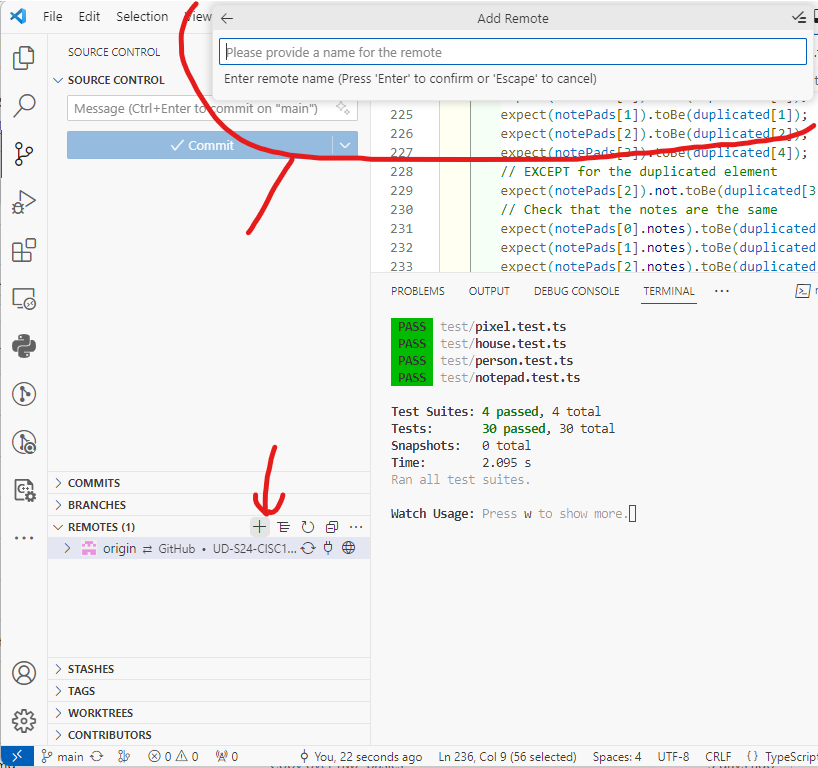
Open up your repository in VS Code.

In the source control panel on the left, expand the “Remotes (1)” option.



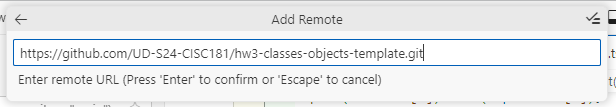
When you hover over the word “Remotes (1),” you will see a “+” button.

Click the “+” button to add a new remote, and a window will popup.



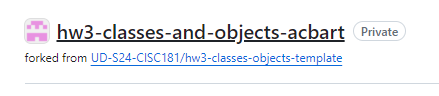
For the name of the new remote, enter “upstream” without quotes, and type enter.

The box will now ask you for the “remote URL.”

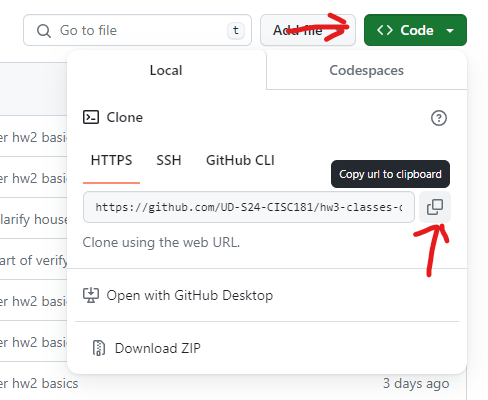


You need the URL of the original template repository (not your copy of the repository, but the original one you forked). This is where we’ve pushed the fixes to the assignment. You can access this repository by first visiting your GitHub repository in the browser.

Click the link after the text “forked from” (e.g., “UD-S24-CISC181/hw3-classes-objects-template”), which should have the word “template” in it. This will be near the top of the page, below the main header bar.

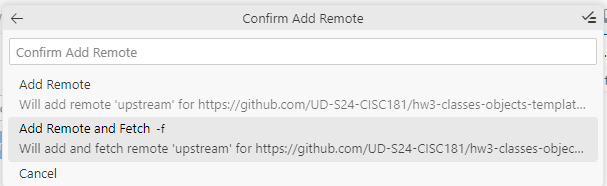


On this new page (the original template repository), you can click the green “<> Code” button to expand a frame with a link to the repository’s web URL, including a copy button. Click the button to copy the web link.

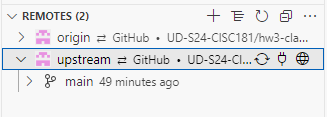


Back in VS Code, you can paste this URL into the box and type “Enter.”

Click the “Add Remote and Fetch -f” option.



Now you have a second remote!

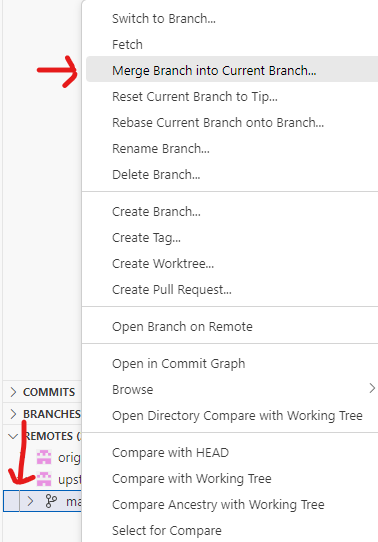


We’re almost done with this step—you just need to merge the upstream changes into your current changes.

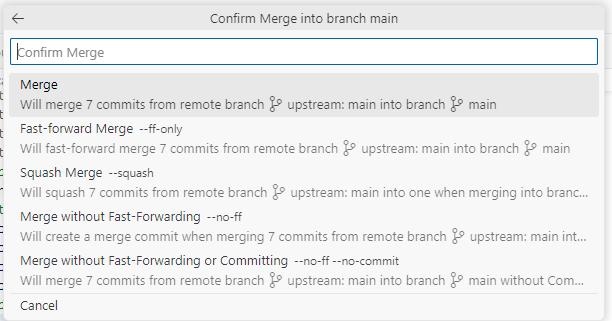
Before you merge, save all your open files.

Then, stage and commit any changed files.

Now, right click the “main” option underneath the “upstream” to expand a menu, and choose “Merge Branch into Current Branch.”



Choose the “Merge” option.



If there are merge conflicts, then go to [Step 4](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html#step-4-resolving-local-merge-conflicts). Otherwise, you are probably done and ready to continue working with the latest changes.

## Step 3: Pulling the Local Changes

Hopefully you have now updated the repository on GitHub. Now, you just need to pull the latest changes.

First, save all your open files.

Then, stage and commit any changed files.

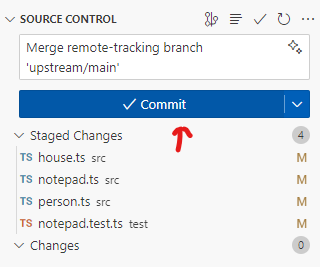
Third, click the “Sync” button to pull the new remote changes (and hopefully push your changes so far).

This last step can go wrong if your local changes have merge conflicts with the remote. For example, if you modified the same line of code that we modified, then you will get a merge conflict. Proceed to [Step 4](https://boots-edu.github.io/textbook/text/14-appendix/upstream.html" \l "step-4-resolving-local-merge-conflicts).

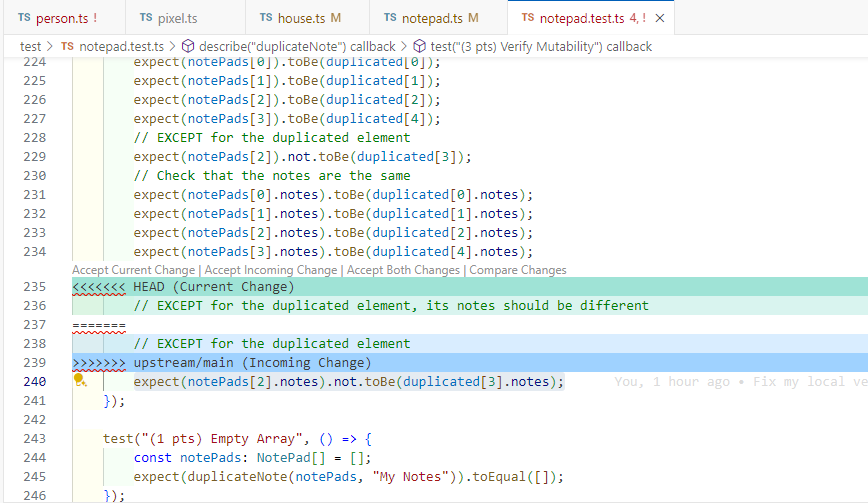
If nothing goes wrong, then you should get the new version of the files and you are done.

## Step 4: Resolving Local Merge Conflicts

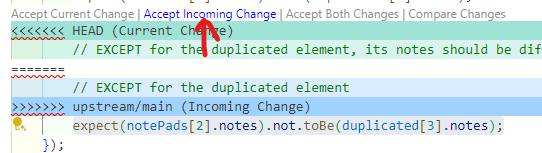
If the source control panel has files listed under “Merge Changes” with red exclamation marks, then you have merge conflicts to resolve.



Click on a file to be merged, and it will open on the right side.

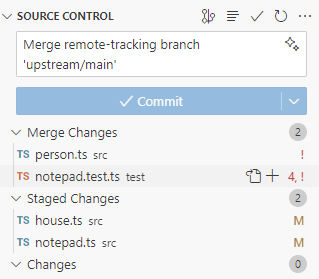


Basically, you just need to edit the file to look like the “final version” you want to have. Git will inject a bunch of <<<<, ====, and >>>> characters into your code. These indicate the modifications that Git wants to make, in order to resolve the conflict. Those need to be removed, eventually. Often, the simplest solution will be to click the “Accept Incoming Change”:



You will need to go through all of the files with red exclamation marks and resolve any conflicts. Think of it as surgery for your code. If you’re in doubt, don’t be afraid to ask for help.

When each file is resolved, then you can stage that file (using the “+” button as you usually do) and make a new commit with the existing message. If the system prompts you “are you sure you want to stage the file with merge conflicts”, then be absolutely sure that you have resolved all conflicts and that you saved the file. You should not stage a file with merge conflicts that have not been resolved!



With everything staged and committed, you can now “Sync Changes” when you are ready to push to GitHub. Most likely, you still have more to finish. But at least now, you are up to date!

Still confused about merge conflicts? We recommend reading over the “Merge Conflicts” and “3-way merge editor” part of the following document for more information about resolving merge conflicts: <https://code.visualstudio.com/docs/sourcecontrol/overview#_merge-conflicts>.

And of course, do not be afraid to seek help in office hours. The instructors are always eager to help you with issues.

## Summary

When you fork a repository, you create a copy of the repository in your own account. This allows you to make changes to the repository without affecting the original repository. However, the original repository may continue to be updated with new features, bug fixes, and other changes. To keep your fork up to date with the original repository, you need to pull changes from the original repository into your fork.