```
// An alternative way to get an object is via pop-ing the
// array to remove objects. Doing this removes the object
// from the array, and returns the object. This is called
// mutating the array, because it changes the underlying
// data inside it.

const poppedFirstOrder = allOrders.pop();

// Now allOrders is empty. Mutating data can be useful for
// many things, but one way to reduce the complexity in your
// codebases is to avoid mutation. TypeScript offers a way
// to declare an array readonly instead:

// Creates a type based on the shape of a purchase order:
type PurchaseOrder = typeof purchaseOrder;

// Creates a readonly array of purchase orders
const readonlyOrders: readonly PurchaseOrder[] = [purchaseOrder];

// Yep! That's a bit more code for sure. There's four
// new things here:
//
//  type PurchaseOrder - Declares a new type to TypeScript.
//
//  typeof - Use the type inference system to set the type
//          based on the const which is passed in next.
//
//  purchaseOrder - Get the variable purchaseOrder and tell
//                  TypeScript this is the shape of all
//                  objects in the orders array.
//
//  readonly - This object does not support mutation, once
//             it is created then the contents of the array
//             will always stay the same.
//
// Now if you try to pop from the readonlyOrders, TypeScript
// will raise an error.

readonlyOrders.pop();

// You can use readonly in all sorts of places, it's a
// little bit of extra syntax here and there, but it
// provides a lot of extra safety.

// You can find out more about readonly:
//  -
https://www.typescriptlang.org/docs/handbook/interfaces.html#readonly-
properties
//  -
https://basarat.gitbooks.io/typescript/content/docs/types/readonly.htm
l

// And you can carry on learning about JavaScript and
// TypeScript in the example on functions:
// example:functions
//
// Or if you want to know more about immutability:
// example:immutability
```


ts_official_samples_js

Catalog

1 JavaScript Essentials

1.1 Hello World

```
// https://www.typescriptlang.org/play/?q=469&target=1
// Welcome to the TypeScript playground. This site is a lot
// like running a TypeScript project inside a web browser.

// The playground makes it easy for you to safely experiment
// with ideas in TypeScript by making it trivial to share
// these projects. The URL for this page is everything
// required to load the project for someone else.

const hello = "Hello";

// You can see on the right the result of the TypeScript
// compiler: this is vanilla JavaScript which can run on
// browsers, servers or anywhere really.

const world = "World";

// You can see how it makes tiny changes to the code, by
// converting a "const" to a "var". This is one of the many
// things TypeScript does to make it possible to run
// anywhere JavaScript runs.

console.log(hello + " " + world);

// Now that you have an idea of how the playground works,
// let's look at how TypeScript makes working with
// JavaScript more fun. During this section we'll be trying
// to keep as close to vanilla JavaScript as possible to
// show how you can re-use existing knowledge.
//
// Click below to continue:
//
// example:objects-and-arrays
```


1.2 Objects and Arrays

```
//
https://www.typescriptlang.org/play?strict=false&q=489#example/objects
-and-arrays
// JavaScript objects are collections of values wrapped up with named
keys.

const userAccount = {
    name: "Kieron",
    id: 0,
  };

  // You can combine these to make larger, more complex data-models.

  const pie = {
    type: "Apple",
  };

  const purchaseOrder = {
    owner: userAccount,
    item: pie,
  };

  // If you use your mouse to hover over some of these words
  // (try purchaseOrder above) you can see how TypeScript is
  // interpreting your JavaScript into labeled types.

  // Values can be accessed via the ".", so to get a username for a
purchase order:
  console.log(purchaseOrder.item.type);

  // If you hover your mouse over each part of the code
  // between the ()s, you can see TypeScript offering more
  // information about each part. Try re-writing this below:

  // Copy this in the next line, character by character:
  //
  //   purchaseOrder.item.type

  // TypeScript provides feedback to the playground
  // about what JavaScript objects are available in this
  // file and lets you avoid typos and see additional
  // information without having to look it up in another place.

  // TypeScript also offers these same features to arrays.
  // Here's an array with just our purchase order above in it.

  const allOrders = [purchaseOrder];

  // If you hover on allOrders, you can tell it's an array
  // because the hover info ends with []. You can access the
  // first order by using square brackets with an index
  // (starting from zero).

  const firstOrder = allOrders[0];
  console.log(firstOrder.item.type);
```

2 Functions with JavaScript

2.1 Generic Functions

```
// https://www.typescriptlang.org/play?q=479#example/generic-functions
// Generics provide a way to use Types as variables in other
// types. Meta.

// We'll be trying to keep this example light, you can do
// a lot with generics and it's likely you will see some very
// complicated code using generics at some point - but that
// does not mean that generics are complicated.

// Let's start with an example where we wrap an input object
// in an array. We will only care about one variable in this
// case, the type which was passed in:

function wrapInArray<Type>(input: Type): Type[] {
    return [input];
  }

  // Note: it's common to see Type referred to as T. This is
  // culturally similar to how people use i in a for loop to
  // represent index. T normally represents Type, so we'll
  // be using the full name for clarity.

  // Our function will use inference to always keep the type
  // passed in the same as the type passed out (though
  // it will be wrapped in an array).

  const stringArray = wrapInArray("hello generics");
  const numberArray = wrapInArray(123);

  // We can verify this works as expected by checking
  // if we can assign a string array to a function which
  // should be an object array:
  const notStringArray: string[] = wrapInArray({});

  // You can also skip the generic inference by adding the
  // type yourself also:
  const stringArray2 = wrapInArray<string>("");

  // wrapInArray allows any type to be used, however there
  // are cases when you need to only allow a subset of types.
  // In these cases you can say the type has to extend a
  // particular type.

  interface Drawable {
    draw: () => void;
  }

  // This function takes a set of objects which have a function
  // for drawing to the screen
  function renderToScreen<Type extends Drawable>(input: Type[]) {
    input.forEach((i) => i.draw());
  }
```


1.3 Functions

```
//
https://www.typescriptlang.org/play?noImplicitAny=false&q=17#example/f
unctions
// There are quite a few ways to declare a function in
// JavaScript. Let's look at a function which adds two
// numbers together:

// Creates a function in global scope called addOldSchool
function addOldSchool(x, y) {
    return x + y;
  }

  // You can move the name of the function to a variable
  // name also
  const anonymousOldSchoolFunction = function (x, y) {
    return x + y;
  };

  // You can also use fat-arrow shorthand for a function
  const addFunction = (x, y) => {
    return x + y;
  };

  // We're going to focus on the last one, but everything
  // applies to all three formats.

  // TypeScript provides additional syntax which adds to a
  // function definition and offers hints on what types
  // are expected by this function.
  //
  // Up next is the most open version of the add function, it
  // says that add takes two inputs of any type: this could
  // be strings, numbers or objects which you've made.

  const add1 = (x: any, y: any) => {
    return x + y;
  };
  add1("Hello", 23);

  // This is legitimate JavaScript (strings can be added
  // like this for example) but isn't optimal for our function
  // which we know is for numbers, so we'll convert the x and
  // y to only be numbers.

  const add2 = (x: number, y: number) => {
    return x + y;
  };
  add2(16, 23);
  add2("Hello", 23);

  // Great. We get an error when anything other than a number
  // is passed in. If you hover over the word add2 above,
  // you'll see that TypeScript describes it as:
```

```
//
//   const add2: (x: number, y: number) => number
//
// Where it has inferred that when the two inputs are
// numbers the only possible return type is a number.
// This is great, you don't have to write extra syntax.
// Let's look at what it takes to do that:

const add3 = (x: number, y: number): string => {
  return x + y;
};

// This function fails because we told TypeScript that it
// should expect a string to be returned but the function
// didn't live up to that promise.

const add4 = (x: number, y: number): number => {
  return x + y;
};

// This is a very explicit version of add2 - there are
// cases when you want to use the explicit return type
// syntax to give yourself a space to work within before
// you get started. A bit like how test-driven development
// recommends starting with a failing test, but in this case
// it's with a failing shape of a function instead.

// This example is only a primer, you can learn a lot more
// about how functions work in TypeScript in the handbook and
// inside the Functional JavaScript section of the examples:
//
// https://www.typescriptlang.org/docs/handbook/functions.html
// example:function-chaining

// And to continue our tour of JavaScript essentials,
// we'll look at how code flow affects the TypeScript types:
// example:code-flow
```


1.4 Code Flow

```
//
https://www.typescriptlang.org/play?strictNullChecks=true&q=275#exampl
e/code-flow
// How code flows inside our JavaScript files can affect
// the types throughout our programs.

const users = [{ name: "Ahmed" }, { name: "Gemma" }, { name: "Jon" }];

// We're going to look to see if we can find a user named "jon".
const jon = users.find((u) => u.name === "jon");

// In the above case, 'find' could fail. In that case we
// don't have an object. This creates the type:
//
//   { name: string } | undefined
//
// If you hover your mouse over the three following uses of 'jon'
below,
// you'll see how the types change depending on where the word is
located:

if (jon) { jon; } else { jon; }

// The type '{ name: string } | undefined' uses a TypeScript
// feature called union types. A union type is a way to
// declare that an object could be one of many things.
//
// The pipe acts as the separator between different types.
// JavaScript's dynamic nature means that lots of functions
// receive and return objects of unrelated types and we need
// to be able to express which ones we might be dealing with.

// We can use this in a few ways. Let's start by looking at
// an array where the values have different types.

const identifiers = ["Hello", "World", 24, 19];

// We can use the JavaScript 'typeof x === y' syntax to
// check for the type of the first element. You can hover on
// 'randomIdentifier' below to see how it changes between
// different locations
const randomIdentifier = identifiers[0];
if (typeof randomIdentifier === "number") {
  randomIdentifier;
} else {
  randomIdentifier;
}
// This control flow analysis means that we can write vanilla
// JavaScript and TypeScript will try to understand how the
// code types will change in different locations.

// To learn more about code flow analysis:
// - example:type-guards

// To continue reading through examples you could jump to a
// few different places now:
//
// - Modern JavaScript: example:immutability
// - Type Guards: example:type-guards
// - Functional Programming with JavaScript example:function-chaining
```

```
interface BoolOrNumberOrStringFunction {
  /** Takes a bool, returns a bool */
  (input: boolean): boolean;
  /** Takes a number, returns a number */
  (input: number): number;
  /** Takes a string, returns a bool */
  (input: string): boolean;
}

// If this is your first time seeing declare, it allows you
// to tell TypeScript something exists even if it doesn't
// exist in the runtime in this file. Useful for mapping
// code with side-effects but extremely useful for demos
// where making the implementation would be a lot of code.

declare const boolOrNumberOrStringFunction:
BoolOrNumberOrStringFunction;

const boolValue = boolOrNumberOrStringFunction(true);
const numberValue = boolOrNumberOrStringFunction(12);
const boolValue2 = boolOrNumberOrStringFunction("string");

// If you hover over the above values and functions you
// can see the right documentation and return values.

// Using function overloads can get you very far, however
// there's another tool for dealing with different types of
// inputs and return values and that is generics.

// These provide a way for you to have types as placeholder
// variables in type definitions.

// example:generic-functions
// example:function-chaining
```



```
  const objectsWithDraw = [{ draw: () => {} }, { draw: () => {} }];
  renderToScreen(objectsWithDraw);

  // It will fail if draw is missing:

  renderToScreen([{}, { draw: () => {} }]);

  // Generics can start to look complicated when you have
  // multiple variables. Here is an example of a caching
  // function that lets you have different sets of input types
  // and caches.

  interface CacheHost {
    save: (a: any) => void;
  }

  function addObjectToCache<Type, Cache extends CacheHost>(obj: Type,
cache: Cache): Cache {
    cache.save(obj);
    return cache;
  }

  // This is the same as above, but with an extra parameter.
  // Note: to make this work though, we had to use an any. This
  // can be worked out by using a generic interface.

  interface CacheHostGeneric<ContentType> {
    save: (a: ContentType) => void;
  }

  // Now when the CacheHostGeneric is used, you need to tell
  // it what ContentType is.

  function addTypedObjectToCache<Type, Cache extends
CacheHostGeneric<Type>>(obj: Type, cache: Cache): Cache {
    cache.save(obj);
    return cache;
  }

  // That escalated pretty quickly in terms of syntax. However,
  // this provides more safety. These are trade-offs, that you
  // have more knowledge to make now. When providing APIs for
  // others, generics offer a flexible way to let others use
  // their own types with full code inference.

  // For more examples of generics with classes and interfaces:
  //
  // example:advanced-classes
  // example:typescript-with-react
  // https://www.typescriptlang.org/docs/handbook/generics.html
```

2.2 Typing Functions

```
// https://www.typescriptlang.org/play?q=68#example/typing-functions
// TypeScript's inference can get you very far, but there
// are lots of extra ways to provide a richer way to document
// the shape of your functions.

// A good first place is to look at optional params, which
// is a way of letting others know you can skip params.

let i = 0;
const incrementIndex = (value?: number) => {
  i += value === undefined ? 1 : value;
};

// This function can be called like:

incrementIndex();
incrementIndex(0);
incrementIndex(3);

// You can type parameters as functions, which provides
// type inference when you write the functions.

const callbackWithIndex = (callback: (i: number) => void) => {
  callback(i);
};

// Embedding function interfaces can get a bit hard to read
// with all the arrows. Using a type alias will let you name
// the function param.

type NumberCallback = (i: number) => void;
const callbackWithIndex2 = (callback: NumberCallback) => {
  callback(i);
};

// These can be called like:

callbackWithIndex(index => {
  console.log(index);
});

// By hovering on index above, you can see how TypeScript
// has inferred the index to be a number correctly.

// TypeScript inference can work when passing a function
// as an instance reference too. To show this, we'll use
// a function which changed a number into string:

const numberToString = (n: number) => {
  return n.toString();
};

// This can be used in a function like map on an array
```



```
// to convert all numbers into a string, if you hover
// on stringedNumbers below you can see the expected types.
const stringedNumbers = [1, 4, 6, 10].map(i => numberToString(i));

// We can use shorthand to have the function passed directly
// and get the same results with more focused code:
const stringedNumbersTerse = [1, 4, 6, 10].map(numberToString);

// You may have functions which could accept a lot of types
// but you are only interested in a few properties. This is
// a useful case for indexed signatures in types. The
// following type declares that this function is OK to use
// any object so long as it includes the property name:

interface AnyObjectButMustHaveName {
  name: string;
  [key: string]: any;
}

const printFormattedName = (input: AnyObjectButMustHaveName) => {};

printFormattedName({ name: "joey" });
printFormattedName({ name: "joey", age: 23 });

// If you'd like to learn more about index-signatures
// we recommend:
//
//
https://www.typescriptlang.org/docs/handbook/interfaces.html#excess-
property-checks
// https://basarat.gitbooks.io/typescript/docs/types/index-
signatures.html

// You can also allow this kind of behavior everywhere
// via the tsconfig flag suppressExcessPropertyErrors -
// however, you can't know if others using your API have
// this set to off.

// Functions in JavaScript can accept different sets of params.
// There are two common patterns for describing these: union
// types for parameters/return, and function overloads.

// Using union types in your parameters makes sense if there
// are only one or two changes and documentation does not need
// to change between functions.

const boolOrNumberFunction = (input: boolean | number) => {};

boolOrNumberFunction(true);
boolOrNumberFunction(23);

// Function overloads on the other hand offer a much richer
// syntax for the parameters and return types.
```

3.2 This

```
// https://www.typescriptlang.org/play?q=54#example/this
// When calling a method of a class, you generally expect it
// to refer to the current instance of the class.

class Safe {
    contents: string;

    constructor(contents: string) {
      this.contents = contents;
    }

    printContents() {
      console.log(this.contents);
    }
  }

  const safe = new Safe("Crown Jewels");
  safe.printContents();

  // If you come from an objected oriented language where the
  // this/self variable is easily predictable, then you may
  // find you need to read up on how confusing 'this' can be:
  //
  // https://yehudakatz.com/2011/08/11/understanding-javascript-
function-invocation-and-this/
  // https://aka.ms/AA5ugm2

  // TLDR: this can change. The reference to which this refers
  // to can be different depending on how you call the function.

  // For example, if you use a reference to the func in another
  // object, and then call it through that - the this variable
  // has moved to refer to the hosting object:

  const customObjectCapturingThis = { contents:
"http://gph.is/VxeHsW", print: safe.printContents };
  customObjectCapturingThis.print(); // Prints "http://gph.is/VxeHsW"
- not "Crown Jewels"

  // This is tricky, because when dealing with callback APIs -
  // it can be very tempting to pass the function reference
  // directly. This can be worked around by creating a new
  // function at the call site.

  const objectNotCapturingThis = { contents: "N/A", print: () =>
safe.printContents() };
  objectNotCapturingThis.print();

  // There are a few ways to work around this problem. One
  // route is to force the binding of this to be the object
  // you originally intended via bind.
```


2.3 Function Chaining

```
//
https://www.typescriptlang.org/play?esModuleInterop=true&q=156#example
/function-chaining
// Function chaining APIs are a common pattern in
// JavaScript, which can make your code focused
// with less intermediary values and easier to read
// because of their nesting qualities.

// A really common API which works via chaining
// is jQuery. Here is an example of jQuery
// being used with the types from DefinitelyTyped:

import $ from "jquery";

// Here's an example use of the jQuery API:

$("#navigation").css("background", "red").height(300).fadeIn(200);

// If you add a dot on the line above, you'll see
// a long list of functions. This pattern is easy to
// reproduce in JavaScript. The key is to make sure
// you always return the same object.

// Here is an example API which creates a chaining
// API. The key is to have an outer function which
// keeps track of internal state, and an object which
// exposes the API that is always returned.

const addTwoNumbers = (start = 1) => {
  let n = start;

  const api = {
    // Implement each function in your API
    add(inc: number = 1) {
      n += inc;
      return api;
    },

    print() {
      console.log(n);
      return api;
    },
  };
  return api;
};

// Which allows the same style of API as we
// saw in jQuery:

addTwoNumbers(1).add(3).add().print().add(1);

// Here's a similar example which uses a class:
```

```
class AddNumbers {
  private n: number;

  constructor(start = 0) {
    this.n = start;
  }

  public add(inc = 1) {
    this.n = this.n + inc;
    return this;
  }

  public print() {
    console.log(this.n);
    return this;
  }
}

// Here it is in action:

new AddNumbers(2).add(3).add().print().add(1);

// This example used the TypeScript
// type inference to provide a way to
// provide tooling to JavaScript patterns.

// For more examples on this:
//
//  - example:code-flow
```


3 Working With Classes

3.1 [Classes 101]

```
// https://www.typescriptlang.org/play?q=369#example/classes-101
// A class is a special type of JavaScript object which
// is always created via a constructor. These classes
// act a lot like objects, and have an inheritance structure
// similar to languages such as Java/C#/Swift.

// Here's an example class:

class Vendor {
    name: string;

    constructor(name: string) {
      this.name = name;
    }

    greet() {
      return "Hello, welcome to " + this.name;
    }
  }

  // An instance can be created via the new keyword, and
  // you can call methods and access properties from the
  // object.

  const shop = new Vendor("Ye Olde Shop");
  console.log(shop.greet());

  // You can subclass an object. Here's a food cart which
  // has a variety as well as a name:

  class FoodTruck extends Vendor {
    cuisine: string;

    constructor(name: string, cuisine: string) {
      super(name);
      this.cuisine = cuisine;
    }

    greet() {
      return "Hi, welcome to food truck " + this.name + ". We serve " +
this.cuisine + " food.";
    }
  }

  // Because we indicated that there needs to be two arguments
  // to create a new FoodTruck, TypeScript will provide errors
  // when you only use one:

  const nameOnlyTruck = new FoodTruck("Salome's Adobo");

  // Correctly passing in two arguments will let you create a
  // new instance of the FoodTruck:

  const truck = new FoodTruck("Dave's Doritos", "junk");
  console.log(truck.greet());
```

3.4 Mixins

```
// https://www.typescriptlang.org/play?q=156#example/mixins
// Mixins are a faux-multiple inheritance pattern for classes
// in JavaScript which TypeScript has support for. The pattern
// allows you to create a class which is a merge of many
// classes.

// To get started, we need a type which we'll use to extend
// other classes from. The main responsibility is to declare
// that the type being passed in is a class.

type Constructor = new (...args: any[]) => {};

// Then we can create a series of classes which extend
// the final class by wrapping it. This pattern works well
// when similar objects have different capabilities.

// This mixin adds a scale property, with getters and setters
// for changing it with an encapsulated private property:

function Scale<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    // Mixins may not declare private/protected properties
    // however, you can use ES2020 private fields
    _scale = 1;

    setScale(scale: number) {
      this._scale = scale;
    }

    get scale(): number {
      return this._scale;
    }
  };
}

// This mixin adds extra methods around alpha composition
// something which modern computers use to create depth:

function Alpha<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    alpha = 1;

    setHidden() {
      this.alpha = 0;
    }

    setVisible() {
      this.alpha = 1;
    }

    setAlpha(alpha: number) {
      this.alpha = alpha;
    }
  };
}

// A simple sprite base class which will then be extended:
```



```
  const customObjectCapturingThisAgain = { contents: "N/A", print:
safe.printContents.bind(safe) };
  customObjectCapturingThisAgain.print();

  // To work around an unexpected this context, you can also
  // change how you create functions in your class. By
  // creating a property which uses an arrow function, the
  // binding of this is done at a different time. Which makes
  // it more predictable for those less experienced with the
  // JavaScript runtime.

  class SafelyBoundSafe {
    contents: string;

    constructor(contents: string) {
      this.contents = contents;
    }

    printContents = () => {
      console.log(this.contents);
    };
  }

  // Now passing the function to another object
  // to run does not accidentally change this.

  const saferSafe = new SafelyBoundSafe("Golden Skull");
  saferSafe.printContents();

  const customObjectTryingToChangeThis = {
    contents: "http://gph.is/XLof62",
    print: saferSafe.printContents,
  };

  customObjectTryingToChangeThis.print();

  // If you have a TypeScript project, you can use the compiler
  // flag noImplicitThis to highlight cases where TypeScript
  // cannot determine what type "this" is for a function.

  // You can learn more about that in the handbook:
  //
  // https://www.typescriptlang.org/docs/handbook/utility-
types.html#thistypet
```

3.3 Generic Classes

```
// https://www.typescriptlang.org/play?q=462#example/generic-classes
// This example is mostly in TypeScript, because it is much
// easier to understand this way first. At the end we'll
// cover how to create the same class but using JSDoc instead.

// Generic Classes are a way to say that a particular type
// depends on another type. For example, here is a drawer
// which can hold any sort of object, but only one type:

class Drawer<ClothingType> {
    contents: ClothingType[] = [];

    add(object: ClothingType) {
      this.contents.push(object);
    }

    remove() {
      return this.contents.pop();
    }
  }

  // In order to use a Drawer, you will need another
  // type to work with:

  interface Sock {
    color: string;
  }

  interface TShirt {
    size: "s" | "m" | "l";
  }

  // We can create a Drawer just for socks by passing in the
  // type Sock when we create a new Drawer:
  const sockDrawer = new Drawer<Sock>();

  // Now we can add or remove socks to the drawer:
  sockDrawer.add({ color: "white" });
  const mySock = sockDrawer.remove();

  // As well as creating a drawer for TShirts:
  const tshirtDrawer = new Drawer<TShirt>();
  tshirtDrawer.add({ size: "m" });

  // If you're a bit eccentric, you could even create a drawer
  // which mixes Socks and TShirts by using a union:

  const mixedDrawer = new Drawer<Sock | TShirt>();

  // Creating a class like Drawer without the extra TypeScript
  // syntax requires using the template tag in JSDoc. In this
  // example we define the template variable, then provide
  // the properties on the class:
```



```
// To have this work in the playground, you'll need to change
// the settings to be a JavaScript file, and delete the
// TypeScript code above

/**
 * @template {{}} ClothingType
 */
class Dresser {
  constructor() {
    /** @type {ClothingType[]} */
    this.contents = [];
  }

  /** @param {ClothingType} object */
  add(object) {
    this.contents.push(object);
  }

  /** @return {ClothingType} */
  remove() {
    return this.contents.pop();
  }
}

// Then we create a new type via JSDoc:

/**
 * @typedef {Object} Coat An item of clothing
 * @property {string} color The colour for coat
 */

// Then when we create a new instance of that class
// we use @type to assign the variable as a Dresser
// which handles Coats.

/** @type {Dresser<Coat>} */
const coatDresser = new Dresser();

coatDresser.add({ color: "green" });
const coat = coatDresser.remove();
```

4.2 Immutability

```
// https://www.typescriptlang.org/play?q=405#example/immutability
// JavaScript is a language with a few ways to declare that
// some of your objects don't change. The most prominent is
// const - which says that the value won't change.

const helloWorld = "Hello World";

// You cannot change helloWorld now, TypeScript will give
// you an error about this, because you would get one at
// runtime instead.

helloWorld = "Hi world";

// Why care about immutability? A lot of this is about
// reducing complexity in your code. If you can reduce the
// number of things which can change, then there are less
// things to keep track of.

// Using const is a great first step, however this fails
// down a bit when using objects.

const myConstantObject = {
  msg: "Hello World",
};

// myConstantObject is not quite a constant though, because
// we can still make changes to parts of the object, for
// example we can change msg:

myConstantObject.msg = "Hi World";

// const means the value at that point stays the same, but
// that the object itself may change internally. This can
// be changed using Object.freeze.

const myDefinitelyConstantObject = Object.freeze({
  msg: "Hello World",
});

// When an object is frozen, then you cannot change the
// internals. TypeScript will offer errors in these cases:

myDefinitelyConstantObject.msg = "Hi World";

// This works the same for arrays too:

const myFrozenArray = Object.freeze(["Hi"]);
myFrozenArray.push("World");

// Using freeze means you can trust that the object is
// staying the same under the hood.

// TypeScript has a few extra syntax hooks to improve working
// with immutable data which you can find in the TypeScript
// section of the examples:
//
// example:literals
// example:type-widening-and-narrowing
```



```
class Sprite {
  name = "";
  x = 0;
  y = 0;

  constructor(name: string) {
    this.name = name;
  }
}

// Here we create two different types of sprites
// which have different capabilities:

const ModernDisplaySprite = Alpha(Scale(Sprite));
const EightBitSprite = Scale(Sprite);

// Creating instances of these classes shows that
// the objects have different sets of properties
// and methods due to their mixins:

const flappySprite = new ModernDisplaySprite("Bird");
flappySprite.x = 10;
flappySprite.y = 20;
flappySprite.setVisible();
flappySprite.setScale(0.8);
console.log(flappySprite.scale);

const gameBoySprite = new EightBitSprite("L block");
gameBoySprite.setScale(0.3);

// Fails because an EightBitSprite does not have
// the mixin for changing alphas:
gameBoySprite.setAlpha(0.5);

// If you want to make more guarantees over the classes
// which you wrap, you can use a constructor with generics.

type GConstructor<T = {}> = new (...args: any[]) => T;

// Now you can declare that this mixin can only be
// applied when the base class is a certain shape.

type Moveable = GConstructor<{ setXYAcceleration: (x: number, y:
number) => void }>;

// We can then create a mixin which relies on the function
// present in the parameter to the GConstructor above.

function Jumpable<TBase extends Moveable>(Base: TBase) {
  return class extends Base {
    jump() {
      // This mixin knows about setXYAcceleration now
      this.setXYAcceleration(0, 20);
    }
  };
}

// We cannot create this sprite until there is a class
// in the mixin hierarchy which adds setXYAcceleration:
const UserSprite = new Jumpable(ModernDisplaySprite);
```

4 Modern JavaScript

4.1 Async Await

```
// https://www.typescriptlang.org/play?q=388#example/async-await
// Modern JavaScript added a way to handle callbacks in an
// elegant way by adding a Promise based API which has special
// syntax that lets you treat asynchronous code as though it
// acts synchronously.

// Like all language features, this is a trade-off in
// complexity: making a function async means your return
// values are wrapped in Promises. What used to return a
// string, now returns a Promise<string>.

const func = () => ":wave:";
const asyncFunc = async () => ":wave:";

const myString = func();
const myPromiseString = asyncFunc();

myString.length;

// myPromiseString is a Promise, not the string:

myPromiseString.length;

// You can use the await keyword to convert a promise
// into its value. Today, these only work inside an async
// function.

const myWrapperFunction = async () => {
  const myString = func();
  const myResolvedPromiseString = await asyncFunc();

  // Via the await keyword, now myResolvedPromiseString
  // is a string
  myString.length;
  myResolvedPromiseString.length;
};

// Code which is running via an await can throw errors,
// and it's important to catch those errors somewhere.

const myThrowingFunction = async () => {
  throw new Error("Do not call this");
};

// We can wrap calling an async function in a try catch to
// handle cases where the function acts unexpectedly.

const asyncFunctionCatching = async () => {
  const myReturnValue = "Hello world";
  try {
    await myThrowingFunction();
  } catch (error) {
    console.error("myThrowingFunction failed", error);
  }
  return myReturnValue;
```



```
};

// Due to the ergonomics of this API being either returning
// a single value, or throwing, you should consider offering
// information about the result inside the returned value and
// use throw only when something truly exceptional has
// occurred.

const exampleSquareRootFunction = async (input: any) => {
  if (isNaN(input)) {
    throw new Error("Only numbers are accepted");
  }

  if (input < 0) {
    return { success: false, message: "Cannot square root negative
number" };
  } else {
    return { success: true, value: Math.sqrt(input) };
  }
};

// Then the function consumers can check in the response and
// figure out what to do with your return value. While this
// is a trivial example, once you have started working with
// networking code these APIs become worth the extra syntax.

const checkSquareRoot = async (value: number) => {
  const response = await exampleSquareRootFunction(value);
  if (response.success) {
    response.value;
  }
};

// Async/Await took code which looked like this:

// getResponse(url, (response) => {
//   getResponse(response.url, (secondResponse) => {
//     const responseData = secondResponse.data
//     getResponse(responseData.url, (thirdResponse) => {
//       ...
//     })
//   })
// })

// And let it become linear like:

// const response = await getResponse(url)
// const secondResponse = await getResponse(response.url)
// const responseData = secondResponse.data
// const thirdResponse = await getResponse(responseData.url)
// ...

// Which can make the code sit closer to left edge, and
// be read with a consistent rhythm.
```

```
// Then use it by referencing the typedef's name:

/** @type { User } */
const user = {};

// There's the TypeScript compatible inline type shorthand,
// which you can use for both type and typedef:

/** @type {{ owner: User, name: string }} */
const resource;

/** @typedef {{owner: User, name: string}} Resource */

/** @type {Resource} */
const otherResource;

// Declaring a typed function:

/**
 * Adds two numbers together
 * @param {number} a The first number
 * @param {number} b The second number
 * @returns {number}
 */
function addTwoNumbers(a, b) {
  return a + b;
}

// You can use most of TypeScript's type tools, like unions:

/** @type {(string | boolean)} */
let stringOrBoolean = "";
stringOrBoolean = false;

// Extending globals in JSDoc is a more involved process
// which you can see in the VS Code docs:
//
// https://code.visualstudio.com/docs/nodejs/working-with-
javascript#_global-variables-and-type-checking

// Adding JSDoc comments to your functions is a win-win
// situation; you get better tooling and so do all your
// API consumers.
```


4.3 Import Export

```
// https://www.typescriptlang.org/play?q=426#example/import-export
// JavaScript added import/export to the language back in 2016
// and TypeScript has complete support for this style of
// linking between files and to external modules. TypeScript
// expands on this syntax by also allowing types to be passed
// with code.

// Let's look at importing code from a module.

import { danger, message, warn, DangerDSLType } from "danger";

// This takes a set of named imports from a node module
// called danger. While there are more than four imports,
// these are the only ones that we have chosen to import.

// Specifically naming which imports you are importing
// gives tools the ability to remove unused code in your
// apps, and helps you understand what is being used in
// a particular file.

// In this case: danger, message and warn are JavaScript
// imports - where as DangerDSLType is an interface type.

// TypeScript lets engineers document their code using
// JSDoc, and docs are imported also. For example if
// you hover on the different parts below, you see
// explanations of what they are.

danger.git.modified_files;

// If you want to know how to provide these documentation
// annotations read example:jsdoc-support

// Another way to import code is by using the default export
// of a module. An example of this is the debug module, which
// exposes a function that creates a logging function.

import debug from "debug";
const log = debug("playground");
log("Started running code");

// Because of the nature of default exports having no true
// name, they can be tricky when applied with static analysis
// tools like the refactoring support in TypeScript but they
// have their uses.

// Because there is a long history in importing/exporting code
// in JavaScript, there is a confusing part of default exports:
// Some exports have documentation that implies you can write
// an import like this:

import req from "request";

// However that fails, and then you find a stack overflow
// which recommends the import as:
```

```
import * as req from "request";

// And this works. Why? We'll get back to that at the end of
// our section on exporting.

// In order to import, you must be able to export. The modern
// way to write exports is using the export keyword.

/** The current stickers left on the roll */
export const numberOfStickers = 11;

// This could be imported into another file by:
//
// import { numberOfStickers } from "./path/to/file"

// You can have as many of those in a file as you like. Then
// a default export is close to the same thing.

/** Generates a sticker for you */
const stickerGenerator = () => { };
export default stickerGenerator;

// This could be imported into another file by:
//
// import getStickers from "./path/to/file"
//
// The naming is up to the module consumer.

// These aren't the only types of imports, just the most common
// in modern code. Covering all of the ways code can cross
// module boundaries is a very long topic in the handbook:
//
// https://www.typescriptlang.org/docs/handbook/modules.html

// However, to try cover that last question. If you look at
// the JavaScript code for this example - you'll see this:

// var stickerGenerator = function () { };
// exports.default = stickerGenerator;

// This sets the default property on the exports object
// to be stickerGenerator. There is code out there which
// sets exports to be a function, instead of an object.
//
// TypeScript opted to stick with the ECMAScript specification
// about how to handle those cases, which is to raise an
// error. However, there is a compiler setting which will
// automatically handle those cases for you which is
// esModuleInterop.
//
// If you turn that on for this example, you will see that
// error go away.
```


4.4 JSDoc Support

```
//
https://www.typescriptlang.org/play?useJavaScript=trueq=34#example/jsd
oc-support
// TypeScript has very rich JSDoc support, for a lot of cases
// you can even skip making your files .ts and just use JSDoc
// annotations to create a rich development environment.
//
// A JSDoc comment is a multi-line comment which starts with
// two stars instead of one.

/* This is a normal comment */
/** This is a JSDoc comment */

// JSDoc comments become attached to the closest JavaScript
// code below it.

const myVariable = "Hi";

// If you hover over myVariable, you can see that it has the
// text from inside the JSDoc comment attached.

// JSDoc comments are a way to provide type information to
// TypeScript and your editors. Let's start with an easy one
// setting a variable's type to a built-in type.

// For all of these examples, you can hover over the name,
// and on the next line try write [example]. to see the
// auto-complete options.

/** @type {number} */
var myNumber;

// You can see all of the supported tags in the handbook:
//
// https://www.typescriptlang.org/docs/handbook/type-checking-
javascript-files.html#supported-jsdoc

// However, we'll try go through some of the more common examples
// here. You can also copy & paste any examples from the handbook
// into here.

// Importing the types for JavaScript configuration files:

/** @type { import("webpack").Config } */
const config = {};

// Creating a complex type to re-use in many places:

/**
 * @typedef {Object} User - a User account
 * @property {string} displayName - the name used to show the user
 * @property {number} id - a unique id
 */
```

```
    setState: (prevState: State, props: Props) => Props;
    callback?: () => void;
    render(): FauxactClassComponent<any> | null;
  }

  // Because this class can have both Props and State - it has
  // two generic arguments which are used throughout the class.

  // The React library comes with its own type definitions
  // like these but are much more comprehensive. Let's bring
  // those into our playground and explore a few components.

  import * as React from "react";

  // Your props are your public API, so it's worth taking the
  // time to use JSDoc to explain how it works:

  export interface Props {
    /** The user's name */
    name: string;
    /** Should the name be rendered in bold */
    priority?: boolean;
  }

  const PrintName: React.FC<Props> = props => {
    return (
      <div>
        <p style={{ fontWeight: props.priority ? "bold" :
"normal" }}>{props.name}</p>
      </div>
    );
  };

  // You can play with the new component's usage below:

  const ShowUser: React.FC<Props> = props => {
    return <PrintName name="Ned" />;
  };

  // TypeScript supports providing intellisense inside
  // the {} in an attribute

  let username = "Cersei";
  const ShowStoredUser: React.FC<Props> = props => {
    return <PrintName name={username} priority />;
  };

  // TypeScript works with modern React code too, here you can
  // see that count and setCount have correctly been inferred
  // to use numbers based on the initial value passed into
  // useState.

  import { useState, useEffect } from "react";

  const CounterExample = () => {
    const [count, setCount] = useState(0);

    useEffect(() => {
      document.title = `You clicked ${count} times`;
    });
```


5 External APIs

5.1 TypeScript with Web

```
//
https://www.typescriptlang.org/play?useJavaScript=trueq=35#example/typ
escript-with-web
// The DOM (Document Object Model) is the underlying API for
// working with a webpage, and TypeScript has great support
// for that API.

// Let's create a popover to show when you press "Run" in
// the toolbar above.

const popover = document.createElement("div");
popover.id = "example-popover";

// Note that popover is correctly typed to be a HTMLDivElement
// specifically because we passed in "div".

// To make it possible to re-run this code, we'll first
// add a function to remove the popover if it was already there.

const removePopover = () => {
  const existingPopover = document.getElementById(popover.id);
  if (existingPopover && existingPopover.parentElement) {
    existingPopover.parentElement.removeChild(existingPopover);
  }
};

// Then call it right away.

removePopover();

// We can set the inline styles on the element via the
// .style property on a HTMLElement - this is fully typed.

popover.style.backgroundColor = "#0078D4";
popover.style.color = "white";
popover.style.border = "1px solid black";
popover.style.position = "fixed";
popover.style.bottom = "10px";
popover.style.left = "20px";
popover.style.width = "200px";
popover.style.height = "100px";
popover.style.padding = "10px";

// Including more obscure, or deprecated CSS attributes.
popover.style.webkitBorderRadius = "4px";

// To add content to the popover, we'll need to add
// a paragraph element and use it to add some text.

const message = document.createElement("p");
message.textContent = "Here is an example popover";
```

```
// And we'll also add a close button.

const closeButton = document.createElement("a");
closeButton.textContent = "X";
closeButton.style.position = "absolute";
closeButton.style.top = "3px";
closeButton.style.right = "8px";
closeButton.style.color = "white";
closeButton.style.cursor = "pointer";

closeButton.onclick = () => {
  removePopover();
};

// Then add all of these elements on to the page.
popover.appendChild(message);
popover.appendChild(closeButton);
document.body.appendChild(popover);

// If you hit "Run" above, then a popup should appear
// in the bottom left, which you can close by clicking
// on the x in the top right of the popup.

// This example shows how you can work with the DOM API
// in JavaScript - but using TypeScript to provide great
// tooling support.

// There is an extended example for TypeScript tooling with
// WebGL available here: example:typescript-with-webgl
```


5.2 TypeScript with React

```
//
https://www.typescriptlang.org/play?jsx=2&esModuleInterop=true&q=143#e
xample/typescript-with-react
// React is a popular library for creating user interfaces.
// It provides a JavaScript abstraction for creating view
// components using a JavaScript language extension called
// JSX.

// TypeScript supports JSX, and provides a rich set of
// type tools to richly model how components connect.

// To understand how TypeScript works with React components
// you may want a primer on generics:
//
// - example:generic-functions
// - example:generic-classes

// First we'll look at how generic interfaces are used to map
// React components. This is a faux-React functional component:

type FauxactFunctionComponent<Props extends {}> = (
    props: Props,
    context?: any
  ) => FauxactFunctionComponent<any> | null | JSX.Element;

  // Roughly:
  //
  // FauxactFunctionComponent is a generic function which relies on
  // another type, Props. Props has to be an object (to make sure
  // you don't pass a primitive) and the Props type will be
  // re-used as the first argument in the function.

  // To use it, you need a props type:

  interface DateProps {
    iso8601Date: string;
    message: string;
  }

  // We can then create a DateComponent which uses the
  // DateProps interface, and renders the date.

  const DateComponent: FauxactFunctionComponent<DateProps> = props =>
(
    <time dateTime={props.iso8601Date}>{props.message}</time>
  );

  // This creates a function which is generic with a Props
  // variable which has to be an object. The component function
  // returns either another component function or null.

  // The other component API is a class-based one. Here's a
  // simplified version of that API:

  interface FauxactClassComponent<Props extends {}, State = {}> {
    props: Props;
    state: State;
```

```
attribute float a_particleDistance;
attribute float a_particleAngle;
attribute float a_particleY;
uniform float u_time; // Global state

varying vec2 v_position;
varying vec3 v_color;
varying float v_overlight;

void main() {
  float angle = a_startAngle + a_angularVelocity * u_time;
  float vertPosition = 1.1 - mod(u_time * .25 + a_particleY, 2.2);
  float viewAngle = a_particleAngle + mod(u_time * .25, 6.28);

  mat4 vMatrix = mat4(
    1.3, 0.0, 0.0, 0.0,
    0.0, 1.3, 0.0, 0.0,
    0.0, 0.0, 1.0, 1.0,
    0.0, 0.0, 0.0, 1.0
  );

  mat4 shiftMatrix = mat4(
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    a_particleDistance * sin(viewAngle), vertPosition,
a_particleDistance * cos(viewAngle), 1.0
  );

  mat4 pMatrix = mat4(
    cos(a_rotationAxisAngle), sin(a_rotationAxisAngle), 0.0, 0.0,
    -sin(a_rotationAxisAngle), cos(a_rotationAxisAngle), 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0
  ) * mat4(
    1.0, 0.0, 0.0, 0.0,
    0.0, cos(angle), sin(angle), 0.0,
    0.0, -sin(angle), cos(angle), 0.0,
    0.0, 0.0, 0.0, 1.0
  );

  gl_Position = vMatrix * shiftMatrix * pMatrix * vec4(a_position *
0.03, 0.0, 1.0);
  vec4 normal = vec4(0.0, 0.0, 1.0, 0.0);
  vec4 transformedNormal = normalize(pMatrix * normal);

  float dotNormal = abs(dot(normal.xyz, transformedNormal.xyz));
  float regularLighting = dotNormal / 2.0 + 0.5;
  float glanceLighting = smoothstep(0.92, 0.98, dotNormal);
  v_color = vec3(
    mix((0.5 - transformedNormal.z / 2.0) * regularLighting, 1.0,
glanceLighting),
    mix(0.5 * regularLighting, 1.0, glanceLighting),
    mix((0.5 + transformedNormal.z / 2.0) * regularLighting, 1.0,
glanceLighting)
  );

  v_position = a_position;
  v_overlight = 0.9 + glanceLighting * 0.1;
}
```



```
    return (
      <div>
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>Click me</button>
      </div>
    );
  };

  // React and TypeScript is a really, really big topic
  // but the fundamentals are pretty small: TypeScript
  // supports JSX, and the rest is handled by the React
  // typings from Definitely Typed.

  // You can learn more about using React with TypeScript
  // from these sites:
  //
  // https://github.com/typescript-cheatsheets/react-typescript-
cheatsheet
  // https://egghead.io/courses/use-typescript-to-develop-react-
applications
  // https://levelup.gitconnected.com/ultimate-react-component-
patterns-with-typescript-2-8-82990c516935
```

5.3 TypeScript with Deno

```
// https://www.typescriptlang.org/play/?q=100#example/typescript-with-
deno
// Deno is a work-in-progress JavaScript and TypeScript
// runtime based on v8 with a focus on security.

// https://deno.land

// Deno has a sandbox-based permissions system which reduces the
// access JavaScript has to the file-system or the network and uses
// http based imports which are downloaded and cached locally.

// Here is an example of using deno for scripting:

import compose from "https://deno.land/x/denofun/lib/compose.ts";

function greet(name: string) {
  return `Hello, ${name}!`;
}

function makeLoud(x: string) {
  return x.toUpperCase();
}

const greetLoudly = compose(makeLoud, greet);

// Echos "HELLO, WORLD!."
greetLoudly("world");

import concat from "https://deno.land/x/denofun/lib/concat.ts";

// Returns "helloworld"
concat("hello", "world");
```

5.4 TypeScript with Node

```
//
https://www.typescriptlang.org/play?useJavaScript=trueq=501#example/ty
pescript-with-node
// Node.js is a very popular JavaScript runtime built on v8,
// the JavaScript engine which powers Chrome. You can use it
// to build servers, front-end clients and anything in-between.

// https://nodejs.org/

// Node.js comes with a set of core libraries which extend the
// JavaScript runtime. They range from path handling:

import { join } from "path";
const myPath = join("~", "downloads", "todo_list.json");

// To file manipulation:

import { readFileSync } from "fs";
const todoListText = readFileSync(myPath, "utf8");

// You can incrementally add types to your JavaScript projects
// using JSDoc-style type. We'll make one for our TODO list item
// based on the JSON structure:

/**
 * @typedef {Object} TODO a TODO item
 * @property {string} title The display name for the TODO item
 * @property {string} body The description of the TODO item
 * @property {boolean} done Whether the TODO item is completed
 */

// Now assign that to the return value of JSON.parse
// to learn more about this, see: example:jsdoc-support

/** @type {TODO[]} a list of TODOs */
const todoList = JSON.parse(todoListText);

// And process handling:
import { spawnSync } from "child_process";
todoList
  .filter(todo => !todo.done)
  .forEach(todo => {
    // Use the ghi client to create an issue for every todo
    // list item which hasn't been completed yet.

    // Note that you get correct auto-complete and
    // docs in JS when you highlight 'todo.title' below.
    spawnSync(`ghi open --message "${todo.title}\n${todo.body}"`);
  });

// TypeScript has up-to-date type definitions for all of the
// built in modules via DefinitelyTyped - which means you
// can write node programs with strong type coverage.
```


5.5 TypeScript with WebGL

```
//
https://www.typescriptlang.org/play?useJavaScript=trueq=418#example/ty
pescript-with-webgl
// This example creates an HTML canvas which uses WebGL to
// render spinning confetti using JavaScript. We're going
// to walk through the code to understand how it works, and
// see how TypeScript's tooling provides useful insight.

// This example builds off: example:working-with-the-dom

// First up, we need to create an HTML canvas element, which
// we do via the DOM API and set some inline style attributes:

const canvas = document.createElement("canvas");
canvas.id = "spinning-canvas";
canvas.style.backgroundColor = "#0078D4";
canvas.style.position = "fixed";
canvas.style.bottom = "10px";
canvas.style.right = "20px";
canvas.style.width = "500px";
canvas.style.height = "400px";
canvas.style.zIndex = "100";

// Next, to make it easy to make changes, we remove any older
// versions of the canvas when hitting "Run" - now you can
// make changes and see them reflected when you press "Run"
// or (cmd + enter):

const existingCanvas = document.getElementById(canvas.id);
if (existingCanvas && existingCanvas.parentElement) {
  existingCanvas.parentElement.removeChild(existingCanvas);
}

// Tell the canvas element that we will use WebGL to draw
// inside the element (and not the default raster engine):

const gl = canvas.getContext("webgl");

// Next we need to create vertex shaders - these roughly are
// small programs that apply maths to a set of incoming
// array of vertices (numbers).

// You can see the large set of attributes at the top of the shader,
// these are passed into the compiled shader further down the example.

// There's a great overview on how they work here:
// https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html

const vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(
  vertexShader,
  `
precision lowp float;

attribute vec2 a_position; // Flat square on XY plane
attribute float a_startAngle;
attribute float a_angularVelocity;
attribute float a_rotationAxisAngle;
```

6 Helping with JavaScript

6.1 Quick Fixes

```
// https://www.typescriptlang.org/play?q=428#example/quick-fixes
// TypeScript provides quick-fix recommendations for
// common accidents. Prompts show up in your editor based
// on these recommendations.

// For example TypeScript can provide quick-fixes
// for typos in your types:

const eulersNumber = 2.7182818284;
eulersNumber.toStrang();
//          ^______^ - select this to see the light bulb

class ExampleClass {
  method() {
    this.notDeclared = 10;
  }
}
```

6.2 Errors

```
// https://www.typescriptlang.org/play?q=434#example/errors
// By default TypeScript doesn't provide error messaging
// inside JavaScript. Instead the tooling is focused on
// providing rich support for editors.

// Turning on errors however, is pretty easy. In a
// typical JS file, all that's required to turn on TypeScript
// error messages is adding the following comment:

// @ts-check

let myString = "123";
myString = {};

// This may start to add a lot of red squiggles inside your
// JS file. While still working inside JavaScript, you have
// a few tools to fix these errors.

// For some of the trickier errors, which you don't feel
// code changes should happen, you can use JSDoc annotations
// to tell TypeScript what the types should be:

/** @type {string | {}} */
let myStringOrObject = "123";
myStringOrObject = {};

// Which you can read more on here: example:jsdoc-support

// You could declare the failure unimportant, by telling
// TypeScript to ignore the next error:

let myIgnoredError = "123";
// @ts-ignore
myStringOrObject = {};

// You can use type inference via the flow of code to make
// changes to your JavaScript: example:code-flow
```



```
`
);
gl.compileShader(vertexShader);

// This example also uses fragment shaders - a fragment
// shader is another small program that runs through every
// pixel in the canvas and sets its color.

// In this case, if you play around with the numbers you can see how
// this affects the lighting in the scene, as well as the border
// radius on the confetti:

const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(
  fragmentShader,
  `
precision lowp float;
varying vec2 v_position;
varying vec3 v_color;
varying float v_overlight;

void main() {
  gl_FragColor = vec4(v_color, 1.0 - smoothstep(0.8, v_overlight,
length(v_position)));
}
`
);
gl.compileShader(fragmentShader);

// Takes the compiled shaders and adds them to the canvas'
// WebGL context so that can be used:

const shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);

gl.bindBuffer(gl.ARRAY_BUFFER, gl.createBuffer());

// We need to get/set the input variables into the shader in a
// memory-safe way, so the order and the length of their
// values needs to be stored.

const attrs = [
  { name: "a_position", length: 2, offset: 0 }, // e.g. x and y
represent 2 spaces in memory
  { name: "a_startAngle", length: 1, offset: 2 }, // but angle is just
1 value
  { name: "a_angularVelocity", length: 1, offset: 3 },
  { name: "a_rotationAxisAngle", length: 1, offset: 4 },
  { name: "a_particleDistance", length: 1, offset: 5 },
  { name: "a_particleAngle", length: 1, offset: 6 },
  { name: "a_particleY", length: 1, offset: 7 },
];

const STRIDE = Object.keys(attrs).length + 1;

// Loop through our known attributes and create pointers in memory for
the JS side
```

```
// to be able to fill into the shader.

// To understand this API a little bit: WebGL is based on OpenGL
// which is a state-machine styled API. You pass in commands in a
// particular order to render things to the screen.

// So, the intended usage is often not passing objects to every WebGL
// API call, but instead passing one thing to one function, then
passing
// another to the next. So, here we prime WebGL to create an array of
// vertex pointers:

for (var i = 0; i < attrs.length; i++) {
  const name = attrs[i].name;
  const length = attrs[i].length;
  const offset = attrs[i].offset;
  const attribLocation = gl.getAttribLocation(shaderProgram, name);
  gl.vertexAttribPointer(attribLocation, length, gl.FLOAT, false,
STRIDE * 4, offset * 4);
  gl.enableVertexAttribArray(attribLocation);
}

// Then on this line they are bound to an array in memory:

gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, gl.createBuffer());

// Set up some constants for rendering:

const NUM_PARTICLES = 200;
const NUM_VERTICES = 4;

// Try reducing this one and hitting "Run" again,
// it represents how many points should exist on
// each confetti and having an odd number sends
// it way out of whack.

const NUM_INDICES = 6;

// Create the arrays of inputs for the vertex shaders
const vertices = new Float32Array(NUM_PARTICLES * STRIDE *
NUM_VERTICES);
const indices = new Uint16Array(NUM_PARTICLES * NUM_INDICES);

for (let i = 0; i < NUM_PARTICLES; i++) {
  const axisAngle = Math.random() * Math.PI * 2;
  const startAngle = Math.random() * Math.PI * 2;
  const groupPtr = i * STRIDE * NUM_VERTICES;

  const particleDistance = Math.sqrt(Math.random());
  const particleAngle = Math.random() * Math.PI * 2;
  const particleY = Math.random() * 2.2;
  const angularVelocity = Math.random() * 2 + 1;

  for (let j = 0; j < 4; j++) {
    const vertexPtr = groupPtr + j * STRIDE;
    vertices[vertexPtr + 2] = startAngle; // Start angle
    vertices[vertexPtr + 3] = angularVelocity; // Angular velocity
    vertices[vertexPtr + 4] = axisAngle; // Angle diff
    vertices[vertexPtr + 5] = particleDistance; // Distance of the
particle from the (0,0,0)
```



```
    vertices[vertexPtr + 6] = particleAngle; // Angle around Y axis
    vertices[vertexPtr + 7] = particleY; // Angle around Y axis
  }

  // Coordinates
  vertices[groupPtr] = vertices[groupPtr + STRIDE * 2] = -1;
  vertices[groupPtr + STRIDE] = vertices[groupPtr + STRIDE * 3] = +1;
  vertices[groupPtr + 1] = vertices[groupPtr + STRIDE + 1] = -1;
  vertices[groupPtr + STRIDE * 2 + 1] = vertices[groupPtr + STRIDE * 3
+ 1] = +1;

  const indicesPtr = i * NUM_INDICES;
  const vertexPtr = i * NUM_VERTICES;
  indices[indicesPtr] = vertexPtr;
  indices[indicesPtr + 4] = indices[indicesPtr + 1] = vertexPtr + 1;
  indices[indicesPtr + 3] = indices[indicesPtr + 2] = vertexPtr + 2;
  indices[indicesPtr + 5] = vertexPtr + 3;
}

// Pass in the data to the WebGL context
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);

const timeUniformLocation = gl.getUniformLocation(shaderProgram,
"u_time");
const startTime = (window.performance || Date).now();

// Start the background colour as black
gl.clearColor(0, 0, 0, 1);

// Allow alpha channels on in the vertex shader
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE);

// Set the WebGL context to be the full size of the canvas
gl.viewport(0, 0, canvas.width, canvas.height);

// Create a run-loop to draw all of the confetti
(function frame() {
  gl.uniform1f(timeUniformLocation, ((window.performance ||
Date).now() - startTime) / 1000);

  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.drawElements(gl.TRIANGLES, NUM_INDICES * NUM_PARTICLES,
gl.UNSIGNED_SHORT, 0);
  requestAnimationFrame(frame);
})();

// Add the new canvas element into the bottom left
// of the playground
document.body.appendChild(canvas);

// Credit: based on this JSFiddle by Subzey
// https://jsfiddle.net/subzey/52sowezj/
```