```
} as const;

// When "as const" is applied to the object, then it becomes
// a object literal which doesn't change instead of a
// mutable object which can.

myUnchangingUser.name = "Raîssa";

// "as const" is a great tool for fixtured data, and places
// where you treat code as literals inline. "as const" also
// works with arrays:

const exampleUsers = [{ name: "Brian" }, { name: "Fahrooq" }] as
const;
```


typescript_official_samples

Catalog

01_any.ts

```
// https://www.typescriptlang.org/play/?q=111
// Any is the TypeScript escape clause. You can use any to
// either declare a section of your code to be dynamic and
// JavaScript like, or to work around limitations in the
// type system.

// A good case for any is JSON parsing:

const myObject = JSON.parse("{}");

// Any declares to TypeScript to trust your code as being
// safe because you know more about it. Even if that is
// not strictly true. For example, this code would crash:

myObject.x.y.z;

// Using an any gives you the ability to write code closer to
// original JavaScript with the trade-off of type safety.

// any is much like a 'type wildcard' which you can replace
// with any type (except never) to make one type assignable
// to the other.

declare function debug(value: any): void;

debug("a string");
debug(23);
debug({ color: "blue" });

// Each call to debug is allowed because you could replace the
// any with the type of the argument to match.

// TypeScript will take into account the position of the
// anys in different forms, for example with these tuples
// for the function argument.

declare function swap(x: [number, string]): [string, number];

declare const pair: [any, any];
swap(pair);

// The call to swap is allowed because the argument can be
// matched by replacing the first any in pair with number
// and the second `any` with string.

// If tuples are new to you, see: example:tuples

// Unknown is a sibling type to any, if any is about saying
// "I know what's best", then unknown is a way to say "I'm
// not sure what is best, so you need to tell TS the type"
// example:unknown-and-never
```


02_literals.ts

```
// https://www.typescriptlang.org/play/?q=98
// TypeScript has some fun special cases for literals in
// source code.

// In part, a lot of the support is covered in type widening
// and narrowing ( example:type-widening-and-narrowing ) and it's
// worth covering that first.

// A literal is a more concrete subtype of a collective type.
// What this means is that "Hello World" is a string, but a
// string is not "Hello World" inside the type system.

const helloWorld = "Hello World";
let hiWorld = "Hi World"; // this is a string because it is let

// This function takes all strings
declare function allowsAnyString(arg: string);
allowsAnyString(helloWorld);
allowsAnyString(hiWorld);

// This function only accepts the string literal "Hello World"
declare function allowsOnlyHello(arg: "Hello World");
allowsOnlyHello(helloWorld);
allowsOnlyHello(hiWorld);

// This lets you declare APIs which use unions to say it
// only accepts a particular literal:

declare function allowsFirstFiveNumbers(arg: 1 | 2 | 3 | 4 | 5);
allowsFirstFiveNumbers(1);
allowsFirstFiveNumbers(10);

let potentiallyAnyNumber = 3;
allowsFirstFiveNumbers(potentiallyAnyNumber);

// At first glance, this rule isn't applied to complex objects.

const myUser = {
  name: "Sabrina",
};

// See how it transforms `name: "Sabrina"` to `name: string`
// even though it is defined as a constant. This is because
// the name can still change any time:

myUser.name = "Cynthia";

// Because myUser's name property can change, TypeScript
// cannot use the literal version in the type system. There
// is a feature which will allow you to do this however.

const myUnchangingUser = {
  name: "Fatma",
```

```
// Because TypeScript supports code flow analysis, the language
// needs to be able to represent when code logically cannot
// happen. For example, this function cannot return:

const neverReturns = () => {
  // If it throws on the first line
  throw new Error("Always throws, never returns");
};

// If you hover on the type, you see it is a () => never
// which means it should never happen. These can still be
// passed around like other values:

const myValue = neverReturns();

// Having a function never return can be useful when dealing
// with the unpredictability of the JavaScript runtime and
// API consumers that might not be using types:

const validateUser = (user: User) => {
  if (user) {
    return user.name !== "NaN";
  }

  // According to the type system, this code path can never
  // happen, which matches the return type of neverReturns.

  return neverReturns();
};

// The type definitions state that a user has to be passed in
// but there are enough escape valves in JavaScript whereby
// you can't guarantee that.

// Using a function which returns never allows you to add
// additional code in places which should not be possible.
// This is useful for presenting better error messages,
// or closing resources like files or loops.

// A very popular use for never, is to ensure that a
// switch is exhaustive. E.g., that every path is covered.

// Here's an enum and an exhaustive switch, try adding
// a new option to the enum (maybe Tulip?)

enum Flower {
  Rose,
  Rhododendron,
  Violet,
```


03_union-and-intersection-types.ts

```
// https://www.typescriptlang.org/play?q=374#example/union-and-
intersection-types
// Type unions are a way of declaring that an object
// could be more than one type.

type StringOrNumber = string | number;
type ProcessStates = "open" | "closed";
type OddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
type AMessyUnion = "hello" | 156 | { error: true };

// If the use of "open" and "closed" vs string is
// new to you, check out: example:literals

// We can mix different types into a union, and
// what we're saying is that the value is one of those types.

// TypeScript will then leave you to figure out how to
// determine which value it could be at runtime.

// Unions can sometimes be undermined by type widening,
// for example:

type WindowStates = "open" | "closed" | "minimized" | string;

// If you hover above, you can see that WindowStates
// becomes a string - not the union. This is covered in
// example:type-widening-and-narrowing

// If a union is an OR, then an intersection is an AND.
// Intersection types are when two types intersect to create
// a new type. This allows for type composition.

interface ErrorHandling {
  success: boolean;
  error?: { message: string };
}

interface ArtworksData {
  artworks: { title: string }[];
}

interface ArtistsData {
  artists: { name: string }[];
}

// These interfaces can be composed in responses which have
// both consistent error handling, and their own data.

type ArtworksResponse = ArtworksData & ErrorHandling;
type ArtistsResponse = ArtistsData & ErrorHandling;

// For example:
```

```
const handleArtistsResponse = (response: ArtistsResponse) => {
  if (response.error) {
    console.error(response.error.message);
    return;
  }

  console.log(response.artists);
};

// A mix of Intersection and Union types becomes really
// useful when you have cases where an object has to
// include one of two values:

interface CreateArtistBioBase {
  artistID: string;
  thirdParty?: boolean;
}

type CreateArtistBioRequest = CreateArtistBioBase & ({ html: string }
| { markdown: string });

// Now you can only create a request when you include
// artistID and either html or markdown

const workingRequest: CreateArtistBioRequest = {
  artistID: "banksy",
  markdown: "Banksy is an anonymous England-based graffiti artist...",
};

const badRequest: CreateArtistBioRequest = {
  artistID: "banksy",
};
```

04_unknown-and-never.ts

```
// https://www.typescriptlang.org/play?q=308#example/unknown-and-never
// Unknown

// Unknown is one of those types that once it clicks, you
// can find quite a lot of uses for it. It acts like a sibling
// to the any type. Where any allows for ambiguity - unknown
// requires specifics.

// A good example would be in wrapping a JSON parser. JSON
// data can come in many different forms and the creator
// of the json parsing function won't know the shape of the
// data - the person calling that function should.

const jsonParser = (jsonString: string) => JSON.parse(jsonString);

const myAccount = jsonParser(`{ "name": "Dorothea" }`);

myAccount.name;
myAccount.email;

// If you hover on jsonParser, you can see that it has the
// return type of any, so then does myAccount. It's possible
// to fix this with generics - but it's also possible to fix
// this with unknown.

const jsonParserUnknown = (jsonString: string): unknown =>
JSON.parse(jsonString);

const myOtherAccount = jsonParserUnknown(`{ "name": "Samuel" }`);

myOtherAccount.name;

// The object myOtherAccount cannot be used until the type has
// been declared to TypeScript. This can be used to ensure
// that API consumers think about their typing up-front:

type User = { name: string };
const myUserAccount = jsonParserUnknown(`{ "name": "Samuel" }`) as
User;
myUserAccount.name;

// Unknown is a great tool, to understand it more read these:
// https://mariusschulz.com/blog/the-unknown-type-in-typescript
// https://www.typescriptlang.org/docs/handbook/release-
notes/typescript-3-0.html#new-unknown-top-type

// Never
```

06_built-in-utility-types.ts

```
// https://www.typescriptlang.org/play/?strictNullChecks=true&q=149

// When a particular type feels like it's useful in most
// codebases, they are added into TypeScript and become
// available for anyone which means you can consistently
// rely on their availability

// Partial<Type>

// Takes a type and converts all of its properties
// to optional ones.

interface Sticker {
    id: number;
    name: string;
    createdAt: string;
    updatedAt: string;
    submitter: undefined | string;
  }

  type StickerUpdateParam = Partial<Sticker>;

  // Readonly<Type>

  // Takes an object and makes its properties read-only.

  type StickerFromAPI = Readonly<Sticker>;

  // Record<KeysFrom, Type>

  // Creates a type which uses the list of properties from
  // KeysFrom and gives them the value of Type.

  // List which keys come from:
  type NavigationPages = "home" | "stickers" | "about" | "contact";

  // The shape of the data for which each of ^ is needed:
  interface PageInfo {
    title: string;
    url: string;
    axTitle?: string;
  }

  const navigationInfo: Record<NavigationPages, PageInfo> = {
    home: { title: "Home", url: "/" },
    about: { title: "About", url: "/about" },
    contact: { title: "Contact", url: "/contact" },
    stickers: { title: "Stickers", url: "/stickers/all" },
  };

  // Pick<Type, Keys>

  // Creates a type by picking the set of properties Keys
```



```
  Daisy,
}

const flowerLatinName = (flower: Flower) => {
  switch (flower) {
    case Flower.Rose:
      return "Rosa rubiginosa";
    case Flower.Rhododendron:
      return "Rhododendron ferrugineum";
    case Flower.Violet:
      return "Viola reichenbachiana";
    case Flower.Daisy:
      return "Bellis perennis";

    default:
      const _exhaustiveCheck: never = flower;
      return _exhaustiveCheck;
  }
};

// You will get a compiler error saying that your new
// flower type cannot be converted into never.

// Never in Unions

// A never is something which is automatically removed from
// a type union.

type NeverIsRemoved = string | never | number;

// If you look at the type for NeverIsRemoved, you see that
// it is string | number. This is because it should never
// happen at runtime because you cannot assign to it.

// This feature is used a lot in example:conditional-types
```

05_tuples.ts

```
// https://www.typescriptlang.org/play?q=164#example/tuples
// Typically an array contains zero to many objects of a
// single type. TypeScript has special analysis around
// arrays which contain multiple types, and where the order
// in which they are indexed is important.

// These are called tuples. Think of them as a way to
// connect some data, but with less syntax than keyed objects.

// You can create a tuple using JavaScript's array syntax:

const failingResponse = ["Not Found", 404];

// but you will need to declare its type as a tuple.

const passingResponse: [string, number] = ["{}", 200];

// If you hover over the two variable names you can see the
// difference between an array ( (string | number)[] ) and
// the tuple ( [string, number] ).

// As an array, the order is not important so an item at
// any index could be either a string or a number. In the
// tuple the order and length are guaranteed.

if (passingResponse[1] === 200) {
  const localInfo = JSON.parse(passingResponse[0]);
  console.log(localInfo);
}

// This means TypeScript will provide the correct types at
// the right index, and even raise an error if you try to
// access an object at an un-declared index.

passingResponse[2];

// A tuple can feel like a good pattern for short bits of
// connected data or for fixtures.

type StaffAccount = [number, string, string, string?];

const staff: StaffAccount[] = [
  [0, "Adankwo", "adankwo.e@"],
  [1, "Kanokwan", "kanokwan.s@"],
  [2, "Aneurin", "aneurin.s@", "Supervisor"],
];

// When you have a set of known types at the beginning of a
// tuple and then an unknown length, you can use the spread
// operator to indicate that it can have any length and the
// extra indexes will be of a particular type:

type PayStubs = [StaffAccount, ...number[]];
```



```
const payStubs: PayStubs[] = [
  [staff[0], 250],
  [staff[1], 250, 260],
  [staff[0], 300, 300, 300],
];

const monthOnePayments = payStubs[0][1] + payStubs[1][1] +
payStubs[2][1];
const monthTwoPayments = payStubs[1][2] + payStubs[2][2];
const monthThreePayments = payStubs[2][2];

// You can use tuples to describe functions which take
// an undefined number of parameters with types:

declare function calculatePayForEmployee(id: number, ...args:
[...number[]]): number;

calculatePayForEmployee(staff[0][0], payStubs[0][1]);
calculatePayForEmployee(staff[1][0], payStubs[1][1], payStubs[1][2]);

//
// https://www.typescriptlang.org/docs/handbook/release-
notes/typescript-3-0.html#tuples-in-rest-parameters-and-spread-
expressions
// https://auth0.com/blog/typescript-3-exploring-tuples-the-unknown-
type/
```

```
const userID = getID();
console.log("User Logged in: ", userID.toUpperCase());

// Only in strict mode the above will fail ^

// There are ways to tell TypeScript you know more, such as
// a type assertion or via a non-null assertion operator (!)

const definitelyString1 = getID() as string;
const definitelyString2 = getID()!;

// Or you safely can check for the existence via an if:

if (userID) {
  console.log(userID);
}

// Optional Properties

// Void

// Void is the return type of a function which does not
// return a value.

const voidFunction = () => { };
const resultOfVoidFunction = voidFunction();

// This is usually an accident, and TypeScript keeps the void
// type around to let you get compiler errors - even though at
// runtime it would be an undefined.
```



```
  // from Type. Essentially an allow-list for extracting type
  // information from a type.

  type StickerSortPreview = Pick<Sticker, "name" | "updatedAt">;

  // Omit<Type, Keys>

  // Creates a type by removing the set of properties Keys
  // from Type. Essentially a block-list for extracting type
  // information from a type.

  type StickerTimeMetadata = Omit<Sticker, "name">;

  // Exclude<Type, RemoveUnion>

  // Creates a type where any property in Type's properties
  // which don't overlap with RemoveUnion.

  type HomeNavigationPages = Exclude<NavigationPages, "home">;

  // Extract<Type, MatchUnion>

  // Creates a type where any property in Type's properties
  // are included if they overlap with MatchUnion.

  type DynamicPages = Extract<NavigationPages, "home" | "stickers">;

  // NonNullable<Type>

  // Creates a type by excluding null and undefined from a set
  // of properties. Useful when you have a validation check.

  type StickerLookupResult = Sticker | undefined | null;
  type ValidatedResult = NonNullable<StickerLookupResult>;

  // ReturnType<Type>

  // Extracts the return value from a Type.

  declare function getStickerByID(id: number):
Promise<StickerLookupResult>;
  type StickerResponse = ReturnType<typeof getStickerByID>;

  // InstanceType<Type>

  // Creates a type which is an instance of a class, or object
  // with a constructor function.

  class StickerCollection {
    stickers: Sticker[];
  }

  type CollectionItem = InstanceType<typeof StickerCollection>;
```

```
// Required<Type>

// Creates a type which converts all optional properties
// to required ones.

type AccessiblePageInfo = Required<PageInfo>;

// ThisType<Type>

// Unlike other types, ThisType does not return a new
// type but instead manipulates the definition of this
// inside a function. You can only use ThisType when you
// have noImplicitThis turned on in your TSConfig.

// https://www.typescriptlang.org/docs/handbook/utility-types.html
```


07_nullable-types.ts

```
// https://www.typescriptlang.org/play/?strictNullChecks=false&q=408
// JavaScript has two ways to declare values which don't
// exist, and TypeScript adds extra syntax which allows even
// more ways to declare something as optional or nullable.

// First up, the difference between the two JavaScript
// primitives: undefined and null

// Undefined is when something cannot be found or set

const emptyObj = {};
const anUndefinedProperty: undefined = emptyObj["anything"];

// Null is meant to be used when there is a conscious lack
// of a value.

const searchResults = {
  video: { name: "LEGO Movie" },
  text: null,
  audio: { name: "LEGO Movie Soundtrack" },
};

// Why not use undefined? Mainly, because now you can verify
// that text was correctly included. If text returned as
// undefined then the result is the same as though it was
// not there.

// This might feel a bit superficial, but when converted into
// a JSON string, if text was an undefined, it would not be
// included in the string equivalent.

// Strict Null Types

// Before TypeScript 2.0 undefined and null were effectively
// ignored in the type system. This let TypeScript provide a
// coding environment closer to un-typed JavaScript.

// Version 2.0 added a compiler flag called "strictNullChecks"
// and this flag required people to treat undefined and null
// as types which needs to be handled via code-flow analysis
// ( see more at example:code-flow )

// For an example of the difference in turning on strict null
// checks to TypeScript, hover over "Potential String" below:

type PotentialString = string | undefined | null;

// The PotentialString discards the undefined and null. If
// you open the "TS Config" menu, enable strictNullChecks, and come
// back, you'll see that hovering on PotentialString now shows
// the full union.

declare function getID(): PotentialString;
```

```
  // would have returned. You can verify this by
  // hovering over response below.

  if (response.version === 0) {
    console.log(response.msg);
  } else if (response.version === 1) {
    console.log(response.status, response.message);
  }
};

// You're better off using a switch statement instead of
// if statements because you can make assurances that all
// parts of the union are checked. There is a good pattern
// for this using the never type in the handbook:

// https://www.typescriptlang.org/docs/handbook/advanced-
types.html#discriminated-unions
```


08_conditional-types.ts

```
// https://www.typescriptlang.org/play/?q=245
// Conditional Types provide a way to do simple logic in the
// TypeScript type system. This is definitely an advanced
// feature, and it's quite feasible that you won't need to
// use this in your normal day to day code.

// A conditional type looks like:
//
//   A extends B ? C : D
//
// Where the condition is whether a type extends an
// expression, and if so what type should be returned.

// Let's go through some examples, for brevity we're
// going to use single letters for generics. This is optional
// but restricting ourselves to 60 characters makes it
// hard to fit on screen.

type Cat = { meows: true };
type Dog = { barks: true };
type Cheetah = { meows: true; fast: true };
type Wolf = { barks: true; howls: true };

// We can create a conditional type which lets extract
// types which only conform to something which barks.

type ExtractDogish<A> = A extends { barks: true } ? A : never;

// Then we can create types which ExtractDogish wraps:

// A cat doesn't bark, so it will return never
type NeverCat = ExtractDogish<Cat>;
// A wolf will bark, so it returns the wolf shape
type Wolfish = ExtractDogish<Wolf>;

// This becomes useful when you want to work with a
// union of many types and reduce the number of potential
// options in a union:

type Animals = Cat | Dog | Cheetah | Wolf;

// When you apply ExtractDogish to a union type, it is the
// same as running the conditional against each member of
// the type:

type Dogish = ExtractDogish<Animals>;

// = ExtractDogish<Cat> | ExtractDogish<Dog> |
//   ExtractDogish<Cheetah> | ExtractDogish<Wolf>
//
// = never | Dog | never | Wolf
//
// = Dog | Wolf (see example:unknown-and-never)

// This is called a distributive conditional type because
// the type distributes over each member of the union.
```

```
// Deferred Conditional Types

// Conditional types can be used to tighten your APIs which
// can return different types depending on the inputs.

// For example this function which could return either a
// string or number depending on the boolean passed in.

declare function getID<T extends boolean>(fancy: T): T extends true ?
string : number;

// Then depending on how much the type-system knows about
// the boolean, you will get different return types:

let stringReturnValue = getID(true);
let numberReturnValue = getID(false);
let stringOrNumber = getID(Math.random() < 0.5);

// In this case above TypeScript can know the return value
// instantly. However, you can use conditional types in functions
// where the type isn't known yet. This is called a deferred
// conditional type.

// Same as our Dogish above, but as a function instead
declare function isCatish<T>(x: T): T extends { meows: true } ? T :
undefined;

// There is an extra useful tool within conditional types, which
// is being able to specifically tell TypeScript that it should
// infer the type when deferring. That is the 'infer' keyword.

// infer is typically used to create meta-types which inspect
// the existing types in your code, think of it as creating
// a new variable inside the type.

type GetReturnValue<T> = T extends (...args: any[]) => infer R ? R :
T;

// Roughly:
//
//  - this is a conditional generic type called GetReturnValue
//    which takes a type in its first parameter
//
//  - the conditional checks if the type is a function, and
//    if so create a new type called R based on the return
//    value for that function
//
//  - If the check passes, the type value is the inferred
//    return value, otherwise it is the original type
//

type getIDReturn = GetReturnValue<typeof getID>;

// This fails the check for being a function, and would
// just return the type passed into it.
type getCat = GetReturnValue<Cat>;
```


09_discriminate-types.ts

```
// https://www.typescriptlang.org/play/?q=242
// A discriminated type union is where you use code flow
// analysis to reduce a set of potential objects down to one
// specific object.
//
// This pattern works really well for sets of similar
// objects with a different string or number constant
// for example: a list of named events, or versioned
// sets of objects.

type TimingEvent = { name: "start"; userStarted: boolean } | { name:
"closed"; duration: number };

// When event comes into this function, it could be any
// of the two potential types.

const handleEvent = (event: TimingEvent) => {
  // By using a switch against event.name TypeScript's code
  // flow analysis can determine that an object can only
  // be represented by one type in the union.

  switch (event.name) {
    case "start":
      // This means you can safely access userStarted
      // because it's the only type inside TimingEvent
      // where name is "start"
      const initiatedByUser = event.userStarted;
      break;

    case "closed":
      const timespan = event.duration;
      break;
  }
};

// This pattern is the same with numbers which we can use
// as the discriminator.

// In this example, we have a discriminate union and an
// additional error state to handle.

type APIResponses = { version: 0; msg: string } | { version: 1;
message: string; status: number } | { error: string };

const handleResponse = (response: APIResponses) => {
  // Handle the error case, and then return
  if ("error" in response) {
    console.error(response.error);
    return;
  }

  // TypeScript now knows that APIResponse cannot be
  // the error type. If it were the error, the function
```

```
listenForEvent("keyboard", (event: KeyboardInputEvent) => { });
listenForEvent("mouse", (event: MouseInputEvent) => { });

// This can go all the way back to the smallest common type:

listenForEvent("mouse", (event: {}) => { });

// But no further:

listenForEvent("mouse", (event: string) => { });

// This covers the real-world pattern of event listener
// in JavaScript, at the expense of having being sound.

// TypeScript can raise an error when this happens via
// `strictFunctionTypes`. Or, you could work around this
// particular case with function overloads,
// see: example:typing-functions

// Void special casing

// Parameter Discarding

// To learn about special cases with function parameters
// see example:structural-typing

// Rest Parameters

// Rest parameters are assumed to all be optional, this means
// TypeScript will not have a way to enforce the number of
// parameters available to a callback.

function getRandomNumbers(count: number, callback: (...args: number[])
=> void) { }

getRandomNumbers(2, (first, second) => console.log([first, second]));
getRandomNumbers(400, (first) => console.log(first));

// Void Functions Can Match to a Function With a Return Value

// A function which returns a void function, can accept a
// function which takes any other type.

const getPI = () => 3.14;

function runFunction(func: () => void) {
  func();
}

runFunction(getPI);

// For more information on the places where soundness of the
// type system is compromised, see:

// https://github.com/Microsoft/TypeScript/wiki/FAQ#type-system-
behavior
// https://github.com/Microsoft/TypeScript/issues/9825
// https://www.typescriptlang.org/docs/handbook/type-
compatibility.html
```


10_indexed-types.ts

```
// https://www.typescriptlang.org/play/?q=271
// There are times when you find yourself duplicating types.
// A common example is nested resources in an auto-generated
// API response.

interface ArtworkSearchResponse {
    artists: {
      name: string;
      artworks: {
        name: string;
        deathdate: string | null;
        bio: string;
      }[];
    }[];
  }

  // If this interface were hand-crafted, it's pretty easy to
  // imagine pulling out the artworks into an interface like:

  interface Artwork {
    name: string;
    deathdate: string | null;
    bio: string;
  }

  // However, in this case we don't control the API, and if
  // we hand-created the interface then it's possible that
  // the artworks part of ArtworkSearchResponse and
  // Artwork could get out of sync when the response changes.

  // The fix for this is indexed types, which replicate how
  // JavaScript allows accessing properties via strings.

  type InferredArtwork =
ArtworkSearchResponse["artists"][0]["artworks"][0];

  // The InferredArtwork is generated by looking through the
  // type's properties and giving a new name to the subset which
  // you have indexed.
```

11_mapped-types.ts

```
// https://www.typescriptlang.org/play/?q=479
// Mapped types are a way to create new types based
// on another type. Effectively a transformational type.

// Common cases for using a mapped type is dealing with
// partial subsets of an existing type. For example
// an API may return an Artist:

interface Artist {
    id: number;
    name: string;
    bio: string;
  }

  // However, if you were to send an update to the API which
  // only changes a subset of the Artist then you would
  // typically have to create an additional type:

  interface ArtistForEdit {
    id: number;
    name?: string;
    bio?: string;
  }

  // It's very likely that this would get out of sync with
  // the Artist above. Mapped types let you create a change
  // in an existing type.

  type MyPartialType<Type> = {
    // For every existing property inside the type of Type
    // convert it to be a ?: version
    [Property in keyof Type]?: Type[Property];
  };

  // Now we can use the mapped type instead to create
  // our edit interface:
  type MappedArtistForEdit = MyPartialType<Artist>;

  // This is close to perfect, but it does allow id to be null
  // which should never happen. So, let's make one quick
  // improvement by using an intersection type (see:
  // example:union-and-intersection-types )

  type MyPartialTypeForEdit<Type> = {
    [Property in keyof Type]?: Type[Property];
  } & { id: number };

  // This takes the partial result of the mapped type, and
  // merges it with an object which has id: number set.
  // Effectively forcing id to be in the type.

  type CorrectMappedArtistForEdit = MyPartialTypeForEdit<Artist>;

  // This is a pretty simple example of how mapped types
  // work, but covers most of the basics. If you'd like to
  // dive in with more depth, check out the handbook:
  //
  // https://www.typescriptlang.org/docs/handbook/2/mapped-types.html
```


12_soundness.ts

```
// https://www.typescriptlang.org/play/?strictFunctionTypes=false&q=56
// Without a background in type theory, you're unlikely
// to be familiar with the idea of a type system being "sound".

// Soundness is the idea that the compiler can make guarantees
// about the type a value has at runtime, and not just
// during compilation. This is normal for most programming
// languages that are built with types from day one.

// Building a type system which models a language which has
// existed for a few decades however becomes about making
// decisions with trade-offs on three qualities: Simplicity,
// Usability and Soundness.

// With TypeScript's goal of being able to support all JavaScript
// code, the language tends towards simplicity and usability
// when presented with ways to add types to JavaScript.

// Let's look at a few cases where TypeScript is provably
// not sound, to understand what those trade-offs would look
// like otherwise.

// Type Assertions

const usersAge = ("23" as any) as number;

// TypeScript will let you use type assertions to override
// the inference to something which is quite wrong. Using
// type assertions is a way of telling TypeScript you know
// best, and TypeScript will try to let you get on with it.

// Languages which are sound would occasionally use runtime checks
// to ensure that the data matches what your types say - but
// TypeScript aims to have no type-aware runtime impact on
// your transpiled code.

// Function Parameter Bi-variance

// Params for a function support redefining the parameter
// to be a subtype of the original declaration.

interface InputEvent {
  timestamp: number;
}
interface MouseInputEvent extends InputEvent {
  x: number;
  y: number;
}
interface KeyboardInputEvent extends InputEvent {
  keyCode: number;
}

function listenForEvent(eventType: "keyboard" | "mouse", handler:
(event: InputEvent) => void) { }

// You can re-declare the parameter type to be a subtype of
// the declaration. Above, handler expected a type InputEvent
// but in the below usage examples - TypeScript accepts
// a type which has additional properties.
```

```
  }

  // You can see a full list of possible typeof values
  // here:
https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/
typeof

  // Using JavaScript operators can only get you so far. When
  // you want to check your own object types you can use
  // type predicate functions.

  // A type predicate function is a function where the return
  // type offers information to the code flow analysis when
  // the function returns true.

  // Using the possible order, we can use two type guards
  // to declare which type the possibleOrder is:

  function isAnInternetOrder(order: PossibleOrders): order is
InternetOrder {
    return order && "email" in order;
  }

  function isATelephoneOrder(order: PossibleOrders): order is
TelephoneOrder {
    return order && "callerNumber" in order;
  }

  // Now we can use these functions in if statements to narrow
  // down the type which possibleOrder is inside the if:

  if (isAnInternetOrder(possibleOrder)) {
    console.log("Order received via email:", possibleOrder.email);
  }

  if (isATelephoneOrder(possibleOrder)) {
    console.log("Order received via phone:",
possibleOrder.callerNumber);
  }

  // You can read more on code flow analysis here:
  //
  //  - example:code-flow
  //  - example:type-guards
  //  - example:discriminate-types
```


13_structural-typing.ts

```
// https://www.typescriptlang.org/play/?q=499
// TypeScript is a Structural Type System. A structural type
// system means that when comparing types, TypeScript only
// takes into account the members on the type.

// This is in contrast to nominal type systems, where you
// could create two types but could not assign them to each
// other. See example:nominal-typing

// For example, these two interfaces are completely
// transferrable in a structural type system:

interface Ball {
    diameter: number;
  }
  interface Sphere {
    diameter: number;
  }

  let ball: Ball = { diameter: 10 };
  let sphere: Sphere = { diameter: 20 };

  sphere = ball;
  ball = sphere;

  // If we add in a type which structurally contains all of
  // the members of Ball and Sphere, then it also can be
  // set to be a ball or sphere.

  interface Tube {
    diameter: number;
    length: number;
  }

  let tube: Tube = { diameter: 12, length: 3 };

  tube = ball;
  ball = tube;

  // Because a ball does not have a length, then it cannot be
  // assigned to the tube variable. However, all of the members
  // of Ball are inside tube, and so it can be assigned.

  // TypeScript is comparing each member in the type against
  // each other to verify their equality.

  // A function is an object in JavaScript and it is compared
  // in a similar fashion. With one useful extra trick around
  // the params:

  let createBall = (diameter: number) => ({ diameter });
  let createSphere = (diameter: number, useInches: boolean) => {
    return { diameter: useInches ? diameter * 0.39 : diameter };
```

```
  };

  createSphere = createBall;
  createBall = createSphere;

  // TypeScript will allow (number) to equal (number, boolean)
  // in the parameters, but not (number, boolean) -> (number)

  // TypeScript will discard the boolean in the first assignment
  // because it's very common for JavaScript code to skip passing
  // params when they're not needed.

  // For example the array's forEach's callback has three params,
  // value, index and the full array - if TypeScript didn't
  // support discarding parameters, then you would have to
  // include every option to make the functions match up:

  [createBall(1), createBall(2)].forEach((ball, _index, _balls) => {
    console.log(ball);
  });

  // No one needs that.

  // Return types are treated like objects, and any differences
  // are compared with the same object equality rules above.

  let createRedBall = (diameter: number) => ({ diameter, color:
"red" });

  createBall = createRedBall;
  createRedBall = createBall;

  // Where the first assignment works (they both have diameter)
  // but the second doesn't (the ball doesn't have a color).
```


14_type-guards.ts

```
// https://www.typescriptlang.org/play/?q=40
// Type Guarding is the term where you influence the code
// flow analysis via code. TypeScript uses existing JavaScript
// behavior which validates your objects at runtime to influence
// the code flow. This example assumes you've read example:code-flow

// To run through these examples, we'll create some classes,
// here's a system for handling internet or telephone orders.

interface Order {
    address: string;
  }
  interface TelephoneOrder extends Order {
    callerNumber: string;
  }
  interface InternetOrder extends Order {
    email: string;
  }

  // Then a type which could be one of the two Order subtypes or
undefined
  type PossibleOrders = TelephoneOrder | InternetOrder | undefined;

  // And a function which returns a PossibleOrder
  declare function getOrder(): PossibleOrders;
  const possibleOrder = getOrder();

  // We can use the "in" operator to check whether a particular
  // key is on the object to narrow the union. ("in" is a JavaScript
  // operator for testing object keys.)

  if ("email" in possibleOrder) {
    const mustBeInternetOrder = possibleOrder;
  }

  // You can use the JavaScript "instanceof" operator if you
  // have a class which conforms to the interface:

  class TelephoneOrderClass {
    address: string;
    callerNumber: string;
  }

  if (possibleOrder instanceof TelephoneOrderClass) {
    const mustBeTelephoneOrder = possibleOrder;
  }

  // You can use the JavaScript "typeof" operator to
  // narrow your union. This only works with primitives
  // inside JavaScript (like strings, objects, numbers).

  if (typeof possibleOrder === "undefined") {
    const definitelyNotAnOder = possibleOrder;
```

```ts
// A const enum's value is replaced by TypeScript during
// transpilation of your code, instead of being looked up
// via an object at runtime.

const enum MouseAction {
  MouseDown,
  MouseUpOutside,
  MouseUpInside,
}

const handleMouseAction = (action: MouseAction) => {
  switch (action) {
    case MouseAction.MouseDown:
      console.log("Mouse Down");
      break;
  }
};

// If you look at the transpiled JavaScript, you can see
// how the other enums exist as objects and functions,
// however MouseAction is not there.

// This is also true for the check against MouseAction.MouseDown
// inside the switch statement inside handleMouseAction.

// Enums can do more than this, you can read more in the
// TypeScript handbook:
//
// https://www.typescriptlang.org/docs/handbook/enums.html
```


15_type-widening-and-narrowing.ts

```ts
// https://www.typescriptlang.org/play/?q=280
// It might be easiest to start of the discussion of
// widening and narrowing with an example:

const welcomeString = "Hello There";
let replyString = "Hey";

// Aside from the text differences of the strings, welcomeString
// is a const (which means the value will never change)
// and replyString is a let (which means it can change).

// If you hover over both variables, you get very different
// type information from TypeScript:
//
//   const welcomeString: "Hello There"
//
//   let replyString: string

// TypeScript has inferred the type of welcomeString to be
// the literal string "Hello There", whereas replyString
// is general string.

// This is because a let needs to have a wider type, you
// could set replyString to be any other string - which means
// it has a wider set of possibilities.

replyString = "Hi :wave:";

// If replyString had the string literal type "Hey" - then
// you could never change the value because it could only
// change to "Hey" again.

// Widening and Narrowing types is about expanding and reducing
// the possibilities which a type could represent.

// An example of type narrowing is working with unions, the
// example on code flow analysis is almost entirely based on
// narrowing: example:code-flow

// Type narrowing is what powers the strict mode of TypeScript
// via the nullability checks. With strict mode turned off,
// markers for nullability like undefined and null are ignored
// in a union.

declare const quantumString: string | undefined;
// This will fail in strict mode only
quantumString.length;

// In strict mode the onus is on the code author to ensure
// that the type has been narrowed to the non-null type.
// Usually this is as simple as an if check:

if (quantumString) {
  quantumString.length;
}

// In strict mode the type quantumString has two representations.
// Inside the if, the type was narrowed to just string.

// You can see more examples of narrowing in:
```

```
//
// example:union-and-intersection-types
// example:discriminate-types

// And even more resources on the web:
//
// https://mariusschulz.com/blog/literal-type-widening-in-typescript
// https://sandersn.github.io/manual/Widening-and-Narrowing-in-
Typescript.html
```


16_enums.ts

```
// https://www.typescriptlang.org/play/?q=489
// Enums are a feature added to JavaScript in TypeScript
// which makes it easier to handle named sets of constants.

// By default an enum is number based, starting at zero,
// and each option is assigned an increment by one. This is
// useful when the value is not important.

enum CompassDirection {
    North,
    East,
    South,
    West,
  }

  // By annotating an enum option, you set the value;
  // increments continue from that value:

  enum StatusCodes {
    OK = 200,
    BadRequest = 400,
    Unauthorized,
    PaymentRequired,
    Forbidden,
    NotFound,
  }

  // You reference an enum by using EnumName.Value

  const startingDirection = CompassDirection.East;
  const currentStatus = StatusCodes.OK;

  // Enums support accessing data in both directions from key
  // to value, and value to key.

  const okNumber = StatusCodes.OK;
  const okNumberIndex = StatusCodes["OK"];
  const stringBadRequest = StatusCodes[400];

  // Enums can be different types, a string type is common.
  // Using a string can make it easier to debug, because the
  // value at runtime does not require you to look up the number.

  enum GamePadInput {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
  }

  // If you want to reduce the number of objects in your
  // JavaScript runtime, you can create a const enum.
```

```
  // This means you can extend an interface by declaring it
  // a second time.

  interface Kitten {
    purrs: boolean;
  }

  interface Kitten {
    colour: string;
  }

  // In the other case a type cannot be changed outside of
  // its declaration.

  type Puppy = {
    color: string;
  };

  type Puppy = {
    toys: number;
  };

  // Depending on your goals, this difference could be a
  // positive or a negative. However for publicly exposed
  // types, it's a better call to make them an interface.

  // One of the best resources for seeing all of the edge
  // cases around types vs interfaces, this stack overflow
  // thread is a good place to start:

  // https://stackoverflow.com/questions/37233735/typescript-
interfaces-vs-types/52682220#52682220
```


17_nominal-typing.ts

```
// https://www.typescriptlang.org/play/?q=188
// A nominal type system means that each type is unique
// and even if types have the same data you cannot assign
// across types.

// TypeScript's type system is structural, which means
// if the type is shaped like a duck, it's a duck. If a
// goose has all the same attributes as a duck, then it also
// is a duck. You can learn more here: example:structural-typing

// This can have drawbacks, for example there are cases
// where a string or number can have special context and you
// don't want to ever make the values transferrable. For
// example:
//
// -  User Input Strings (unsafe)
// -  Translation Strings
// -  User Identification Numbers
// -  Access Tokens

// We can get most of the value from a nominal type
// system with a little bit of extra code.

// We're going to use an intersectional type, with a unique
// constraint in the form of a property called __brand (this
// is convention) which makes it impossible to assign a
// normal string to a ValidatedInputString.

type ValidatedInputString = string & { __brand: "User Input Post
Validation" };

// We will use a function to transform a string to
// a ValidatedInputString - but the point worth noting
// is that we're just _telling_ TypeScript that it's true.

const validateUserInput = (input: string) => {
  const simpleValidatedInput = input.replace(/\</g, "≤");
  return simpleValidatedInput as ValidatedInputString;
};

// Now we can create functions which will only accept
// our new nominal type, and not the general string type.

const printName = (name: ValidatedInputString) => {
  console.log(name);
};

// For example, here's some unsafe input from a user, going
// through the validator and then being allowed to be printed:
```

```
const input = "alert('bobby tables')";
const validatedInput = validateUserInput(input);
printName(validatedInput);

// On the other hand, passing the un-validated string to
// printName will raise a compiler error:

printName(input);

// You can read a comprehensive overview of the
// different ways to create nominal types, and their
// trade-offs in this 400 comment long GitHub issue:
//
// https://github.com/Microsoft/TypeScript/issues/202
//
// and this post is a great summary:
//
// https://michalzalecki.com/nominal-typing-in-typescript/
```


18_types-vs-interfaces.ts

```
// https://www.typescriptlang.org/play/?q=230
// There are two main tools to declare the shape of an
// object: interfaces and type aliases.
//
// They are very similar, and for the most common cases
// act the same.

type BirdType = {
    wings: 2;
  };

  interface BirdInterface {
    wings: 2;
  }

  const bird1: BirdType = { wings: 2 };
  const bird2: BirdInterface = { wings: 2 };

  // Because TypeScript is a structural type system,
  // it's possible to intermix their use too.

  const bird3: BirdInterface = bird1;

  // They both support extending other interfaces and types.
  // Type aliases do this via intersection types, while
  // interfaces have a keyword.

  type Owl = { nocturnal: true } & BirdType;
  type Robin = { nocturnal: false } & BirdInterface;

  interface Peacock extends BirdType {
    colourful: true;
    flies: false;
  }
  interface Chicken extends BirdInterface {
    colourful: false;
    flies: false;
  }

  let owl: Owl = { wings: 2, nocturnal: true };
  let chicken: Chicken = { wings: 2, colourful: false, flies: false };

  // That said, we recommend you use interfaces over type
  // aliases. Specifically, because you will get better error
  // messages. If you hover over the following errors, you can
  // see how TypeScript can provide terser and more focused
  // messages when working with interfaces like Chicken.

  owl = chicken;
  chicken = owl;

  // One major difference between type aliases vs interfaces
  // are that interfaces are open and type aliases are closed.
```