Typescript Fundamentals v3

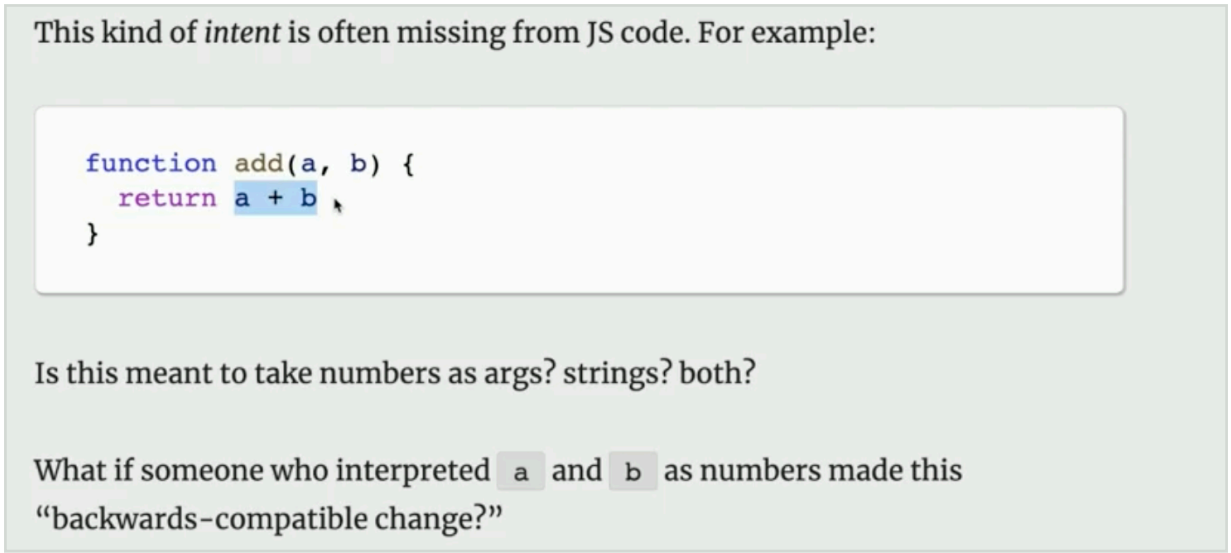
Instructor: Mike North - Senior Staff Engineer @ LinkedIn & Frontend Masters
Course Link: https://www.typescript-training.com/course/fundamentals-v3

What is TypeScript?
-a syntactic superset of Javascript
-compiles to readable JS
-three parts: language, language server and compiler
-open source project maintained by Microsoft
-goal is to add types to Javascript
-kind of like a fancy linter
-increasingly popular

A good way to think of TS files:

- .ts files contain both type information and code that runs
- .js files contain only code that runs
- .d.ts files contain only type information

Why developers want types
-Allows you as a code author to leave more of your intent “on the page”, intent matters



```
function add(a, b, c = 0) {
  return a + b + c
}
```

We're headed for trouble if we decided to pass strings in for `a` and `b`!

Types make the author's intent more clear

-Types make the author's intent more clear

```
function add(a: number, b: number): number {
  return a + b
}
add(3, "4")
```

```
Argument of type 'string' is not assignable to parameter of type
'number'.
```

Try

It has the potential to move some kinds of errors from *runtime* to *compile time*

1

Examples:

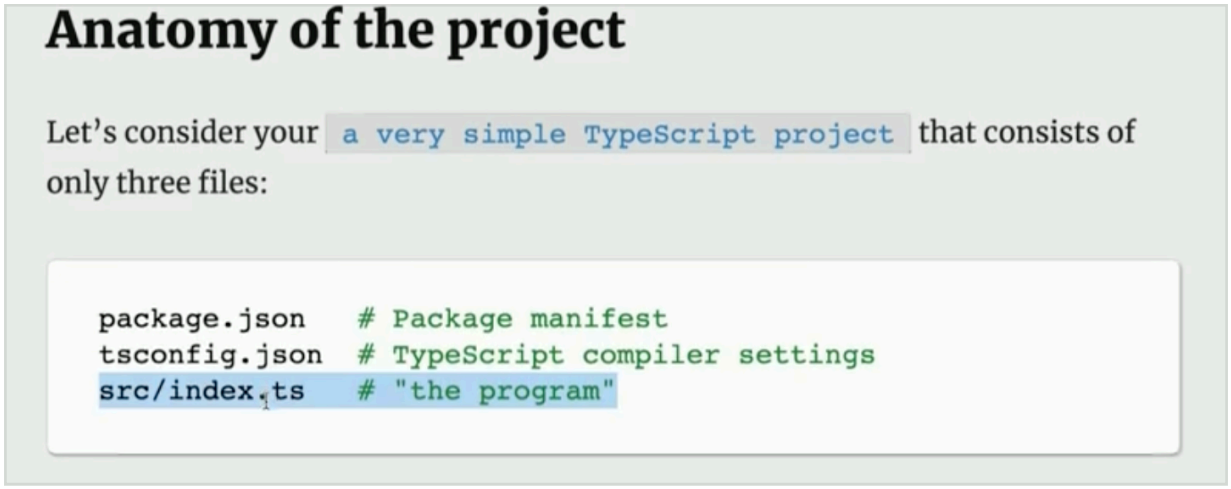
- Values that are potentially absent (`null` or `undefined`)
- Incomplete refactoring
- Breakage around *internal code contracts* (e.g., an argument *becomes* required)

-It serves as the foundation for a great code authoring experience, for example in-editor autocomplete.

Agenda
-Using tsc and compiling TS code into Javascript
-Variables and simple values
-Objects and arrays
-Categorizing type systems

-Set theory, Union, and Intersection types
-Interfaces and Type Aliases
-Hack: Writing types for JSON values
-Functions
-Classes in TypeScript
-Top and bottom types
-User-defined Type guards
-Handling nullish values
-Generics
-Hack: higher-order functions for dictionaries

Compiling a Typescript Program



package.json

```
{
  "name": "hello-ts",
  "license": "NOLICENSE",
  "devDependencies": {
    "typescript": "^4.3.2"
  },
  "scripts": {
    "dev": "tsc --watch --preserveWatchOutput"
  }
}
```

Note that...

- We just have one dependency in our package.json: `typescript`.
- We have a `dev` script (this is what runs when you invoke `yarn dev-hello-ts` from the project root)
  - It runs the TypeScript compiler in "watch" mode (watches for source changes, and rebuilds automatically).

The following is just about the simplest possible config file for the TS compiler:

`tsconfig.json` (view source)

```
{
  "compilerOptions": {
    "outDir": "dist", // where to put the TS files
    "target": "ES3" // which level of JS support to target
  },
  "include": ["src"] // which files to compile
}
```

-In TypeScript, variables are "born" with their types.
-const <value> = literal type
-When typing functions, you can type the arguments and the return type
-JSDoc does not enforce types, but can be a good helper tool (before Typescript)
-JSDoc requires a lot of intention to code in the type information and easy to miss/ forget, but Typescript will alert if something is inconsistent
-Object types - describe the shape of an object (names of properties and types of properties)
-May have to pass in | undefined after object type to increase strictness

-Optional properties can be noted with the question mark (?) at the end of the key's name. (optionalProp?: string)
-Optional properties are not the same as (type | undefined). When you use type | undefined, it still requires the property to have the undefined value

Arrays & Tuples
-number[] = array type
-[number, string] = tuple type

Nominal vs structural type systems
-Nominal type systems like Java depend on the name of the type
-Structural type systems like TypeScript look at the shape of the type

Union Types

Union Types in TypeScript

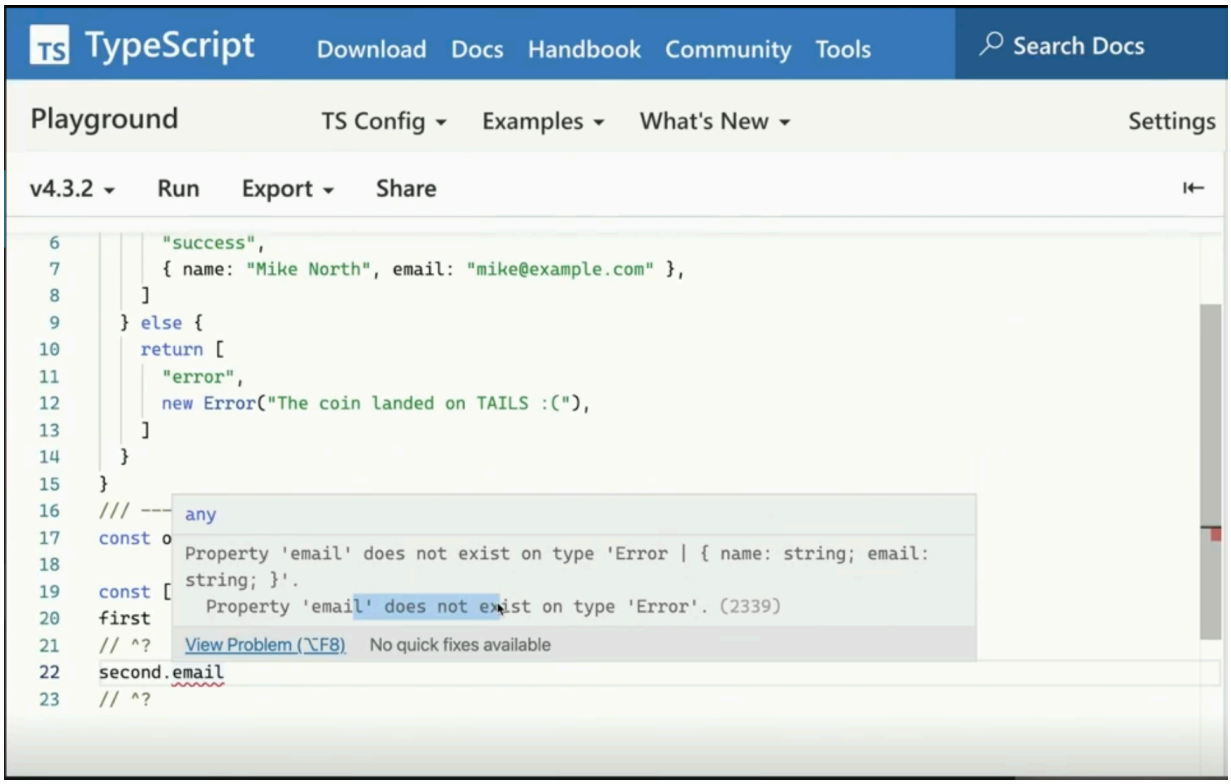
Union types in TypeScript can be described using the `|` (pipe) operator.

For example, if we had a type that could be one of two strings, `"success"` or `"error"`, we could define it as

```
"success" | "error"
```

```
function flipCoin(): "heads" | "tails" {
  if (Math.random() > 0.5) return "heads"
  return "tails"
}

function maybeGetUserInfo():
  | ["error", Error]
  | ["success", { name: string; email: string }] {
  if (flipCoin() === "heads") {
    return [
      "success",
      { name: "Mike North", email: "mike@example.com" },
    ]
  } else {
    return [
      "error",
      new Error("The coin landed on TAILS :("),
    ]
  }
```



-When a value has a type that includes a union, we're only able to use its common behavior

We can see that the autocomplete information for the first value suggests that it's a string. This is because, regardles of whether this happens to be the specific `"success"` or `"error"` string, it's definitely going to be a string.

The second value is a bit more complicated — only the `name` property is available to us. This is because, both our "user info object, and instances of the `Error` class have a `name` property whose value is a string.

What we are seeing here is, when a value has a type that includes a union, we are only able to use the "common behavior" that's guaranteed to be there.

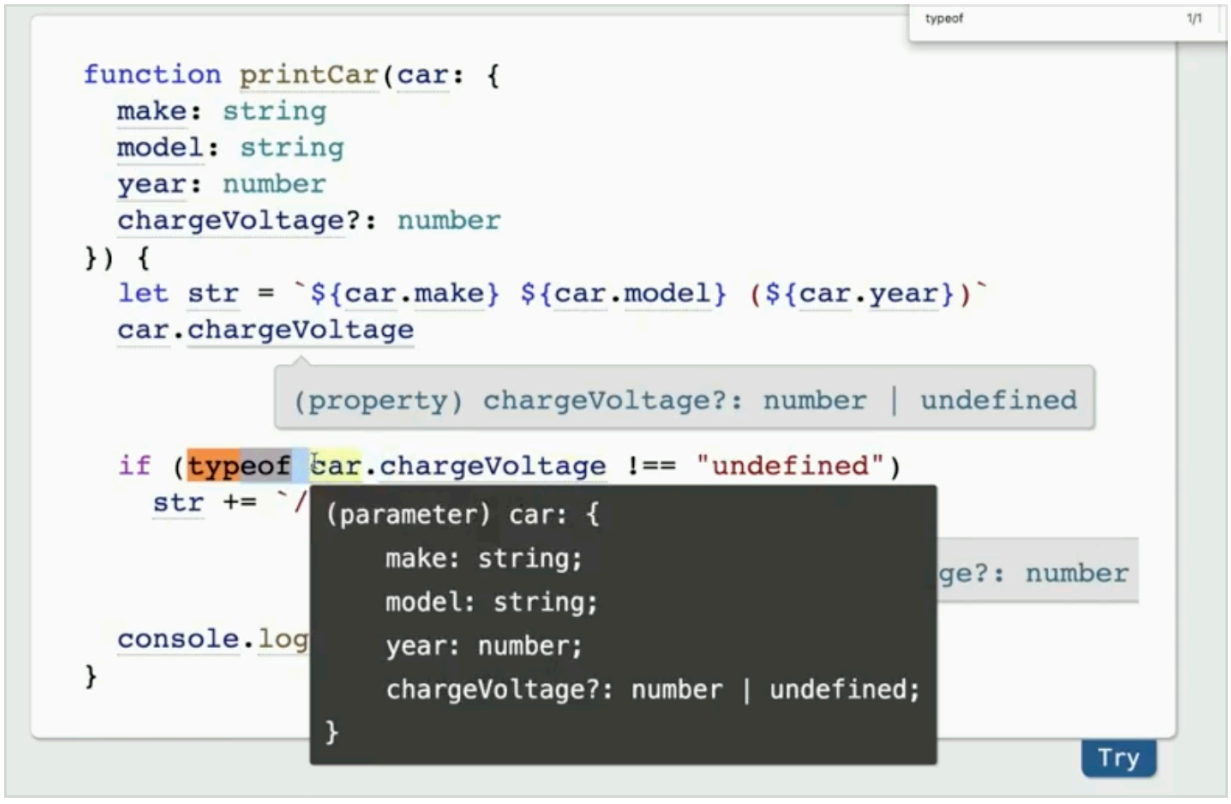
Type Guards (typeof, instanceof)
https://www.typescriptlang.org/docs/handbook/2/narrowing.html

Narrowing with type guards

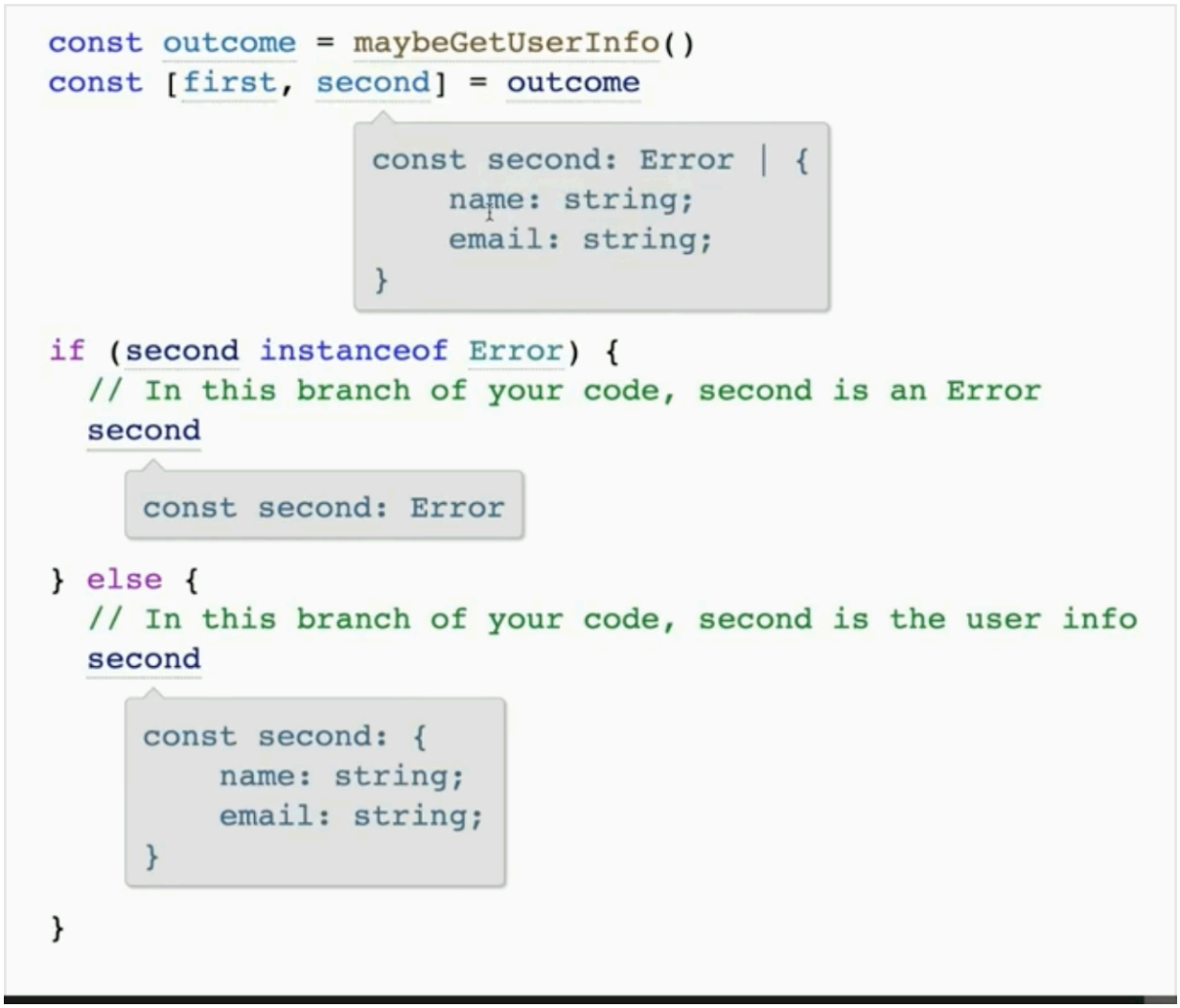
Ultimately, we need to "separate" the two potential possibilities for our value, or we won't be able to get very far. We can do this with type guards.

> *Type guards are expressions, which when used with control flow statement, allow us to have a more specific type for a particular value.*

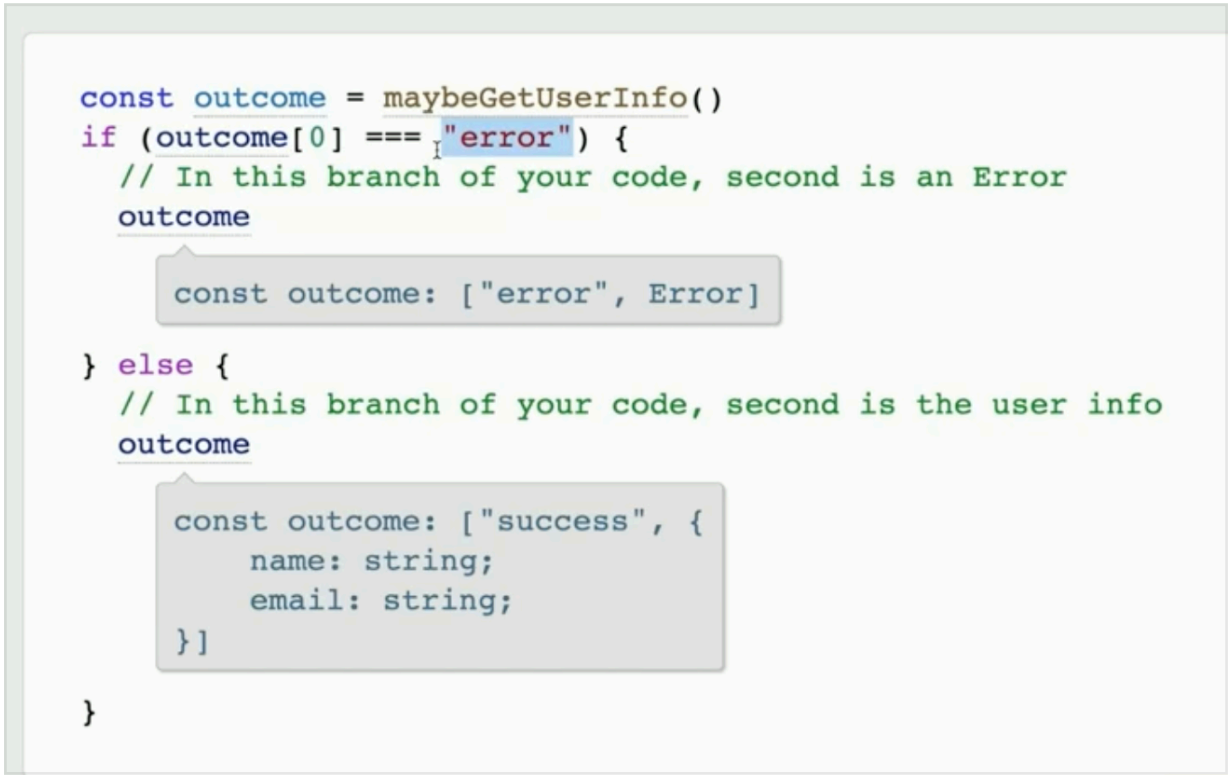
I like to think of these as "glue" between the compile time type-checking and runtime execution of your code. We will work with one that you should already be familiar with to start: `instanceof`.



-Type guards are predicates combined with a control flow (if/else/case/switch) to assert a more specific type



Discriminated Unions (“tagged union type”)
1 scenario or the other (only 2 cases, if/else)
-used with specific key/string for control flow (if this, or that)



Intersection Types

Intersection Types in TypeScript

Intersection types in TypeScript can be described using the `&` (ampersand) operator.

For example, what if we had a `Promise`, that had extra `startTime` and `endTime` properties added to it?

```
function makeWeek(): Date & { end: Date } {
  //⬅ return type

  const start = new Date()
  const end = new Date(start.valueOf() + ONE_WEEK)

  return { ...start, end } // kind of Object.assign
}

const thisWeek = makeWeek()
thisWeek.toISOString()
```

```
const thisWeek: Date & {
```

-& = merging of two types

Interfaces & Type Aliases
https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#interfaces
https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-aliases

Recursion

Type aliases

Think back to the `: {name: string, email: string}` syntax we've used up until this point for type annotations. This syntax will get increasingly complicated as more properties are added to this type. Furthermore, if we pass objects of this type around through various functions and variables, we will end up with a *lot* of types that need to be manually updated whenever we need to make any changes!

Type aliases help to address this, by allowing us to:

- define **a more meaningful name** for this type
- declare the particulars of the type **in a single place**
- **import and export** this type from modules, the same as if it were an exported value

Let's look at the declaration syntax for a moment:

```
type UserContactInfo = {
  name: string
  email: string
}
```

Try

A few things to point out here:

1. This is a rare occasion where we see type information on the right hand side of the assignment operator (`=`)
2. We're using `TitleCase` to format the alias' name. This is a common convention
3. As we can see below, we can only declare an alias of a given name *once* within a given scope. This is kind of like how a `let` or `const` variable declaration works

-Type aliases are just giving a type a name

Inheritance

You can create type aliases that combine existing types with new behavior by using Intersection (`&`) types.

```
type SpecialDate = Date & { getReason(): string }

const newYearsEve:                                {
  ...new Date(),
  getReason: () => "Last day of the year",
}
newYearsEve.getReason
```

interface Date

getDate
getDay
getFullYear
getHours

Interfaces

An interface is a way of defining an *object type*. An "object type" can be thought of as, "an instance of a class could conceivably look like this".

For example, `string | number` is not an object type, because it makes use of the **union type operator**.

```
interface UserInfo {
  name: string
  email: string
}
function printUserInfo(info: UserInfo) {
  info.name

        (property) UserInfo.name: string

}
```

Try

-interfaces are more limited than type aliases and can only define object types. type aliases can describe object types, union types, intersection types, etc.

Heritage clauses
extends - describe inheritance between similar things (class and class, interface and interface)
implements - describe inheritance between unlike things (class and interface)

-interfaces are great ways to define contracts between things
-interfaces can be augmented with the same name and the change applies globally

-multiple inheritance - Typescript (and Javascript) does not support true multiple inheritance (extending from more than one base class), but it does support multiple usage of the implements keyword

```
class LivingOrganism {
  isAlive() {
    return true
  }
}
interface AnimalLike {
  eat(food): void
}
interface CanBark {
  bark(): string
}

class Dog
  extends LivingOrganism
  implements AnimalLike, CanBark
{
  bark() {
    return "woof"
```

-It is not recommended to use implements with a type alias since type aliases can describe more than object types

TypeScript interfaces are “open”, meaning that unlike in type aliases, you can have multiple declarations in the same scope:

```
interface AnimalLike {
  isAlive(): boolean
}
function feed(animal: AnimalLike) {
  animal.eat
          (method) AnimalLike.eat(food: any): void
  animal.isAlive
           (method) AnimalLike.isAlive(): boolean
}

// SECOND DECLARATION OF THE SAME NAME
interface AnimalLike {
  eat(food): void
}
```

Try

-when you redeclare an interface with the same name, you are augmenting the new information to that existing type (adding to it).

Add items to an existing interface

```
window.document // an existing property
          (property) document: Document
window.exampleProperty = 42
              (property) Window.exampleProperty: number
// tells TS that `exampleProperty` exists
interface Window {
  exampleProperty: number
}
```

Try

Choosing which to use

In many situations, either a `type` alias or an `interface` would be perfectly fine, however...

1. **If you need to define something other than an object type** (e.g., use of the `|` union type operator), you must use a type alias
2. If you need to define a type **to use with the `implements` heritage term**, it's best to use an interface
3. If you need to **allow consumers of your types to *augment* them**, you must use an interface.

Recursive Types (as of Typescript 3.7)
-types that reference themselves (self-referential)

You may read or see things that indicate you must use a combination of `interface` and `type` for recursive types. As of TypeScript 3.7 this is now much easier, and works with either type aliases or interfaces.

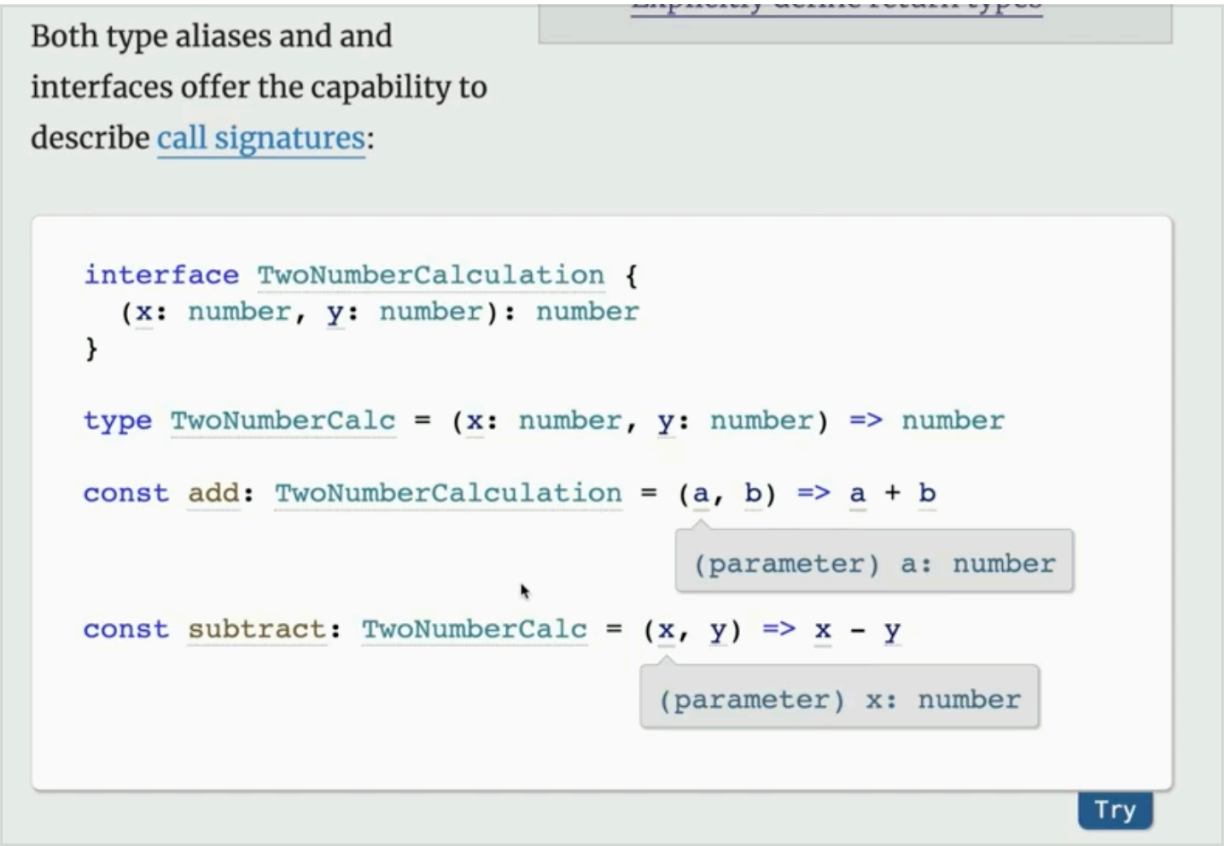
```
type NestedNumbers = number | NestedNumbers[]

const val: NestedNumbers = [3, 4, [5, 6, [7], 59], 221]

if (typeof val !== "number") {
  val.push(41)
        (method) Array<NestedNumbers>.push(...items: NestedNu
  val.push("this will not work")
Argument of type 'string' is not assignable to parameter of type
'NestedNumbers'.
}
```

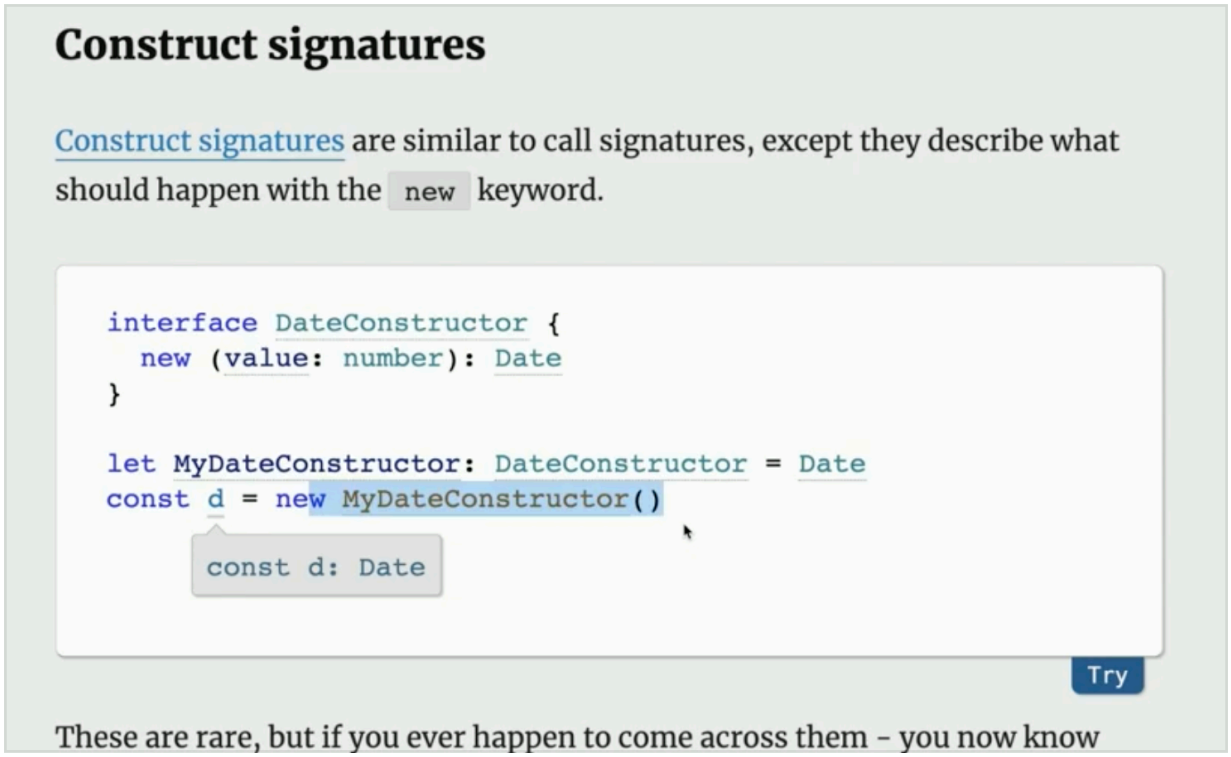
Try

Function Calls



-void should only appear as a function return type, it means the return value should be ignored
-void vs undefined, undefined means you must return undefined, void means ignore whatever is returned

Constructors



Function Overloads
2 Heads (of the function) and implementation of the function (has braces), implementation has to be compatible with the heads

```
function handleMainEvent(
  elem: HTMLFormElement,
  handler: FormSubmitHandler
)
function handleMainEvent(
  elem: HTMLIFrameElement,
  handler: MessageHandler
)
function handleMainEvent(
  elem: HTMLFormElement | HTMLI
  handler: FormSubmitHandler | MessageHandler
) {}
```

```
type MessageHandler = (evt: MessageEvent<any>) => void
```

this types set scope/context

JSON Types Example

```ts
type JSONPrimitive = string | number | boolean | null
type JSONObject = { [k: string]: JSONValue }
type JSONArray = JSONValue[]
type JSONValue = JSONArray | JSONObject | JSONPrimitive

////// DO NOT EDIT ANY CODE BELOW THIS LINE //////
function isJSON(arg: JSONValue) {}

// POSITIVE test cases (must pass)
isJSON("hello")
isJSON([4, 8, 15, 16, 23, 42])
isJSON({ greeting: "hello" })
isJSON(false)
isJSON(true)
isJSON(null)
isJSON({ a: { b: [2, 3, "foo"] } })

// NEGATIVE test cases (must fail)
// @ts-expect-error
isJSON(() => "")
// @ts-expect-error
isJSON(class {})
// @ts-expect-error
isJSON(undefined)
// @ts-expect-error
isJSON(new BigInt(143))
// @ts-expect-error
isJSON(isJSON)
```

Classes

Param Properties

-are short handed ways of writing classes by using constructor and access modifiers to implicitly declare properties and their types.

TypeScript provides a more concise syntax for code like this, through the use of *param properties*:

```
class Car {
  constructor(
    public make: string,
    public model: string,
    public year: number
  ) {}
}

const myCar = new Car("Honda", "Accord", 2017)
myCar.make
      make
      model
```

Top Types
-any & unknown
-Set Theory - thinking of types as defining a set of values a variable or function argument might be. (i.e. x is of type boolean, you may select anything from the following set of values: true or false; one of these two things, the set describes all allowed things that 'x' can be.

```
const x: boolean
```

x could be either item from the following set `{true, false}`. Let's look at another example:

```
const y: number
```

y could be **any number**. If we wanted to get technical and express this in terms of set builder notation, this would be `{y | y is a number}` [1]

Let's look at a few more, just for completeness:

```
let a: 5 | 6 | 7 // anything in { 5, 6, 7 }
let b: null // anything in { null }
let c: {
  favoriteFruit?: "pineapple" // { "pineapple", undefined }
}
```

```
(property) favoriteFruit?: "pineapple" | undefined
```

Try

any

Top types

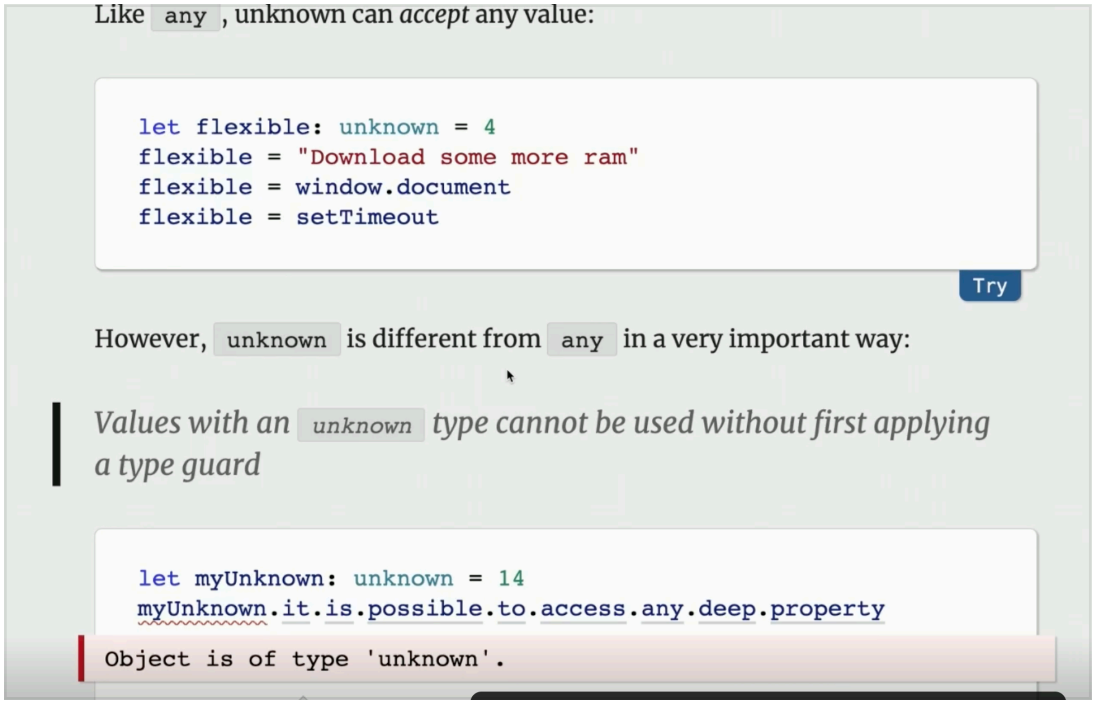
A top type (symbol: `⊤`) is a type that describes **any possible value allowed by the system**. To use our set theory mental model, we could describe this as `{x| x could be anything }`

TypeScript provides two of these types: `any` and `unknown`.

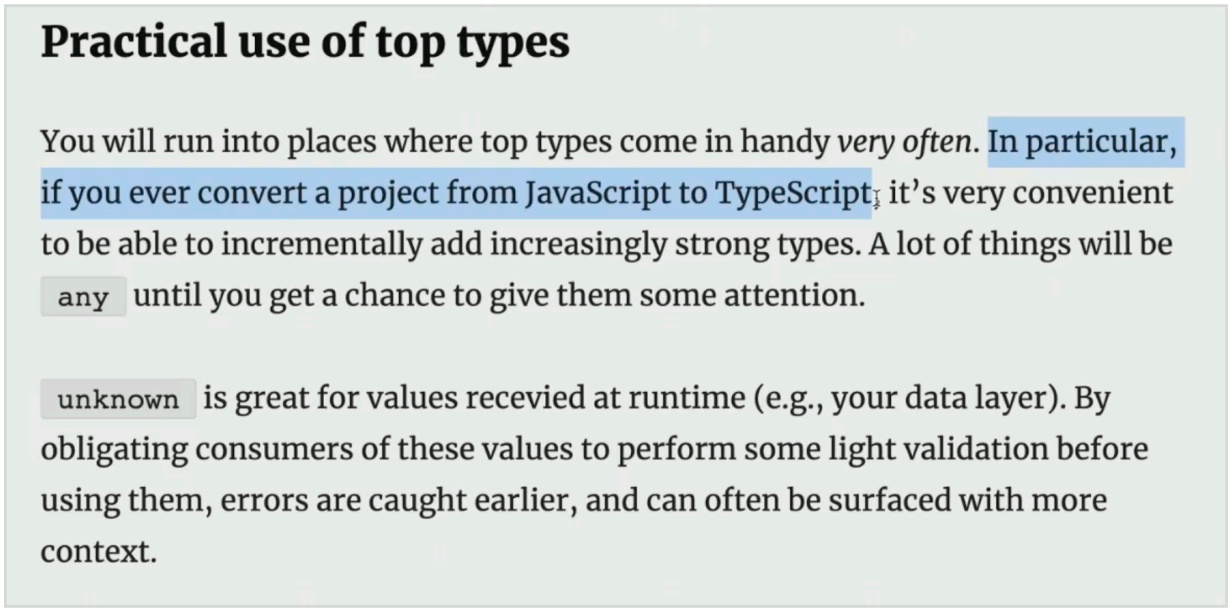
`any`

You can think of values with an `any` type as "playing by the usual JavaScript rules". Here's an illustrative example:

unknown



When to use any or unknown



Bottom Types
-never can be good for exhaustive conditionals

Bottom type: `never`

A bottom type (symbol: `⊥`) is a type that describes **no possible value allowed by the system**. To use our set theory mental model, we could describe this as "any value from the following set: `{ }` (intentionally empty)"

TypeScript provides one bottom type: `never`.

At first glance, this may appear to be an *extremely abstract* and *pointless* concept, but there's one use case that should convince you otherwise. Let's take a look at this scenario below.

Exhaustive conditionals

Type Guards & Narrowing

Built-in type guards

asserts

Writing hi

There are a bunch of type guards that are included with TypeScript. Below is an illustrative example of a wide variety of them:

```
let value:
  | Date
  | null
  | undefined
  | "pineapple"
  | [number]
  | { dateRange: [Date, Date] }

// instanceof
if (value instanceof Date) {
  value
    let value: Date
}
// typeof
else if (typeof value === "string") {
  value
    let value: "pineapple"
}
// Specific value check
else if (value === null) {
  value
    let value: null
}
// Truthy/falsy check
else if (!value) {
  value
    let value: undefined
}
// Some built-in functions
else if (Array.isArray(value)) {
  value
    let value: [number]
}
// Property presence check
else if ("dateRange" in value) {
  value
    let value: {
        dateRange: [Date, Date];
    }
} else {
  value
    let value: never
}
```

User-Defined Typeguards

User-defined type guards

If we lived in a world where we only had the type guards we've seen so far, we'd quickly run into problems as our use of built-in type guards become more complex.

For example, how would we validate objects that are type-equivalent with our `CarLike` interface below?

```
interface CarLike {
  make: string
  model: string
  year: number
}

let maybeCar: unknown

// the guard
if (
  maybeCar &&
  typeof maybeCar === "object" &&
  "make" in maybeCar &&
  typeof maybeCar["make"] === "string" &&
  "model" in maybeCar &&
  typeof maybeCar["model"] === "string" &&
  "year" in maybeCar &&
  typeof maybeCar["year"] === "number"
) {
  maybeCar
      let maybeCar: object

}
```

Try

Validating this type *might* be possible, but it would almost certainly involve casting.

Even if this did work, it is getting messy enough that we'd want to refactor it out into a function or something, so that it could be reused across our codebase.

Let's see what happens when we try to do this:

```ts
interface CarLike {
  make: string
  model: string
  year: number
}

let maybeCar: unknown

// the guard
function isCarLike(valueToTest: any) {
  return (
    valueToTest &&
    typeof valueToTest === "object" &&
    "make" in valueToTest &&
    typeof valueToTest["make"] === "string" &&
    "model" in valueToTest &&
    typeof valueToTest["model"] === "string" &&
    "year" in valueToTest &&
    typeof valueToTest["year"] === "number"
  )
}

// using the guard
if (isCarLike(maybeCar)) {
  maybeCar
      let maybeCar: unknown

}
```

Try

As you can see, the broken/imperfect narrowing effect of this conditional has disappeared.

As things stand right now, TypeScript seems to have no idea that the return value of `isCarLike` *has anything to do with the type of* `valueToTest`

`value is Foo`

The first kind of user-defined type guard we will review is an `is` type guard. It is perfectly suited for our example above because it's meant to work in cooperation with a control flow statement of some sort, to indicate that different branches of the "flow" will be taken based on an evaluation of `valueToTest`'s type. **Pay very close attention to `isCarLike`'s return type**

```
interface CarLike {
  make: string
  model: string
  year: number
}

let maybeCar: unknown

// the guard
function isCarLike(
  valueToTest: any
): valueToTest is CarLike {
  return (
    valueToTest &&
    typeof valueToTest === "object" &&
    "make" in valueToTest &&
    typeof valueToTest["make"] === "string" &&
    "model" in valueToTest &&
    typeof valueToTest["model"] === "string" &&
    "year" in valueToTest &&
    typeof valueToTest["year"] === "number"
  )
}

// using the guard
if (isCarLike(maybeCar)) {
  maybeCar
      let maybeCar: CarLike

}
```

Try

`asserts value is Foo`

There is another approach we could take that eliminates the need for a conditional. **Pay very close attention to `assertsIsCarLike`'s return type**:

```
interface CarLike {
  make: string
  model: string
  year: number
}

let maybeCar: unknown

// the guard
function assertsIsCarLike(
  valueToTest: any
): asserts valueToTest is CarLike {
  if (
    !(
      valueToTest &&
      typeof valueToTest === "object" &&
      "make" in valueToTest &&
      typeof valueToTest["make"] === "string" &&
      "model" in valueToTest &&
      typeof valueToTest["model"] === "string" &&
      "year" in valueToTest &&
      typeof valueToTest["year"] === "number"
    )
  )
    throw new Error(
      `Value does not appear to be a CarLike${valueToTest}`
    )
}

// using the guard
maybeCar
    let maybeCar: unknown
assertsIsCarLike(maybeCar)
maybeCar
    let maybeCar: CarLike
```

Try

Conceptually, what's going on behind the scenes is very similar. By using this special syntax to describe the return type, we are informing TypeScript that **if `assertsIsCarLike` throws an error, it should be taken as an indication that the `valueToTest` is NOT type-equivalent to `CarLike`**.

Therefore, if we get past the assertion and keep executing code on the next line, the type changes from `unknown` to `CarLike`.

Nullish Values

-null, undefined, void

-Non-null assertion operator & definite assignment operator

`null`

> `null` *means: there is a value, and that value is nothing. While some people believe that [null is not an important part of the JS language](), I find that it's useful to express the concept of a "nothing" result (kind of like an empty array, but not an array).*

This *nothing* is very much a defined value, and is certainly a presence — not an absence — of information.


```
const userInfo = {
  name: "Mike",
  email: "mike@example.com",
  secondaryEmail: null, // user has no secondary email
}
```

`undefined`

> `undefined` *means the value isn't available (yet?)*

In the example below, `completedAt` will be set *at some point* but there's a period of time when we haven't yet set it. `undefined` is an unambiguious indication that there *may be something different there in the future*:

```
const formInProgress = {
  createdAt: new Date(),
  data: new FormData(),
  completedAt: undefined, //
}
function submitForm() {
  formInProgress.completedAt = new Date()
}
```

`void`

We gave already covered this in the functions chapter, but as a reminder:

> `void` *should exclusively be used to describe that a function's return value should be ignored*

```
console.log(`console.log returns nothing.`)
```

```
(method) Console.log(...data: any[]): void
```

Try

Non-null assertion operator

The non-null assertion operator (`.!`) is used to cast away the possibility that a value might be `null` or `undefined` .

Keep in mind that the value could still be `null` or `undefined` , this operator just tells TypeScript to ignore that possibility.

If the value *does* turn out to be missing, you willl get the familiar `cannot call foo on undefined` family of errors at runtime:

(!.) - similar to optional chaining
-tells Typescript at compile time to disregard the possibility of null / undefined
-but beware runtime errors could still pop up if not used properly
-useful in test suites because a throw (runtime error) could be considered a test failure

Definite assignment operator

The definite assignment `!:` operator is used to suppress TypeScript's objections about a class field being used, when it can't be proven[1] that it was initialized.

-very useful with async methods in the constructor since Typescript can't pick up the variables/assignments

Generics

```
function listToDict<T>(
  list: T[],
  idGen: (arg: T) => string
): { [k: string]: T } {
  const dict: { [k: string]: T } = {}
  return dict
}
```

Let's look at what this code means.

The TypeParam, and usage to provide an argument type

- **<T> to the right of** `listDict`
  means that the type of this function is now parameterized in terms of a type parameter `T` (which may change on a per-usage basis)

```ts
function isDefined<T>(arg: T | undefined): arg is T {
  return typeof arg !== 'undefined'
}

let x = 4 as (number | undefined)


if (isDefined(x)) {
  x
}
```

Some things you may observe.

- **When using the primitive types `string` and `number` we can see that the union of these two types results in a `never`**. In other words, there is no `string` that can be also regarded as a `number`, and no `number` that can also be regarded as a `string
- **When using the interface types `String` and `Number`, we can see that the union does *not* result in a `never`**.

```ts
interface Dict<T> {
  [k: string]: T
}

// Array.prototype.map, but for Dict
function mapDict<T, S>(
  inputDict: Dict<T>,
  mapFunction: (original: T, key: string) => S
): Dict<S> {
  const outDict: Dict<S> = {}
  for (let k of Object.keys(inputDict)) {
    const thisVal = inputDict[k]
    outDict[k] = mapFunction(thisVal, k)
  }
  return outDict
}
// Array.prototype.filter, but for Dict
function filterDict<T>(
  inputDict: Dict<T>,
  filterFunction: (value: T, key: string) => boolean
): Dict<T> {
  const outDict: Dict<T> = {}
  for (let k of Object.keys(inputDict)) {
    const thisVal = inputDict[k]
    if (filterFunction(thisVal, k)) outDict[k] = thisVal
  }
  return outDict
}
// Array.prototype.reduce, but for Dict
function reduceDict<T, S>(
  inputDict: Dict<T>,
  reducerFunction: (
    currentVal: S,
    dictItem: T,
    key: string
  ) => S,
  initialValue: S
): S {
  let value = initialValue
  for (let k of Object.keys(inputDict)) {
    const thisVal = inputDict[k]
    value = reducerFunction(value, thisVal, k)
  }
  return value
}
```

`T extends` vs `class extends`

Scopes and TypeParams

Best Practices

Generic Constraints

Generic constraints allow us to describe the “minimum requirement” for a type param, such that we can achieve a high degree of flexibility, while still being able to safely assume *some* minimal structure and behavior.

Describing the constraint

The way we define constraints on generics is by using the `extends` keyword.

The correct way of making our function generic is shown in the 1-line change below:

```
- function listToDict(list: HasId[]): Dict<HasId> {
+ function listToDict<T extends HasId>(list: T[]): Dict<T> {
```

Note that our "requirement" for our argument type (`HasId[]`) is now represented in two places:

- `extends HasId` as the constraint on `T`
- `list: T[]` to ensure that we still receive an array

`T EXTENDS` VS `CLASS EXTENDS`

The `extends` keyword is used in object-oriented inheritance, and while not strictly equivalent to how it is used with type params, there is a conceptual connection:

When a class extends from a base class, it's guaranteed to at least align with the base class structure. In the same way, `T extends HasId` guarantees that "T is at least a HasId".

Scopes and TypeParams

When working with function parameters, we know that "inner scopes" have the ability to access "outer scopes" but not vice versa:

```
function receiveFruitBasket(bowl) {
  console.log("Thanks for the fruit basket!")
  // only `bowl` can be accessed here
  eatApple(bowl, (apple) => {
    // both `bowl` and `apple` can be accessed here
  })
}
```

Type params work a similar way:

```
// outer function
function tupleCreator<T>(first: T) {
  // inner function
  return function finish<S>(last: S): [T, S] {
    return [first, last]
  }
}
const finishTuple = tupleCreator(3)
const t1 = finishTuple(null)
      const t1: [number, null]
const t2 = finishTuple([4, 8, 15, 16, 23, 42])
      const t2: [number, number[]]
```

Try

The same design principles that you use for deciding whether values belong as **class fields vs. arguments** passed to members should serve you well here.

Remember, this is not exactly an *independent decision* to make, as types belong to the same scope as values they describe.

Best Practices

- **Use each type parameter *at least twice***. Any less and you might be casting with the `as` keyword. Let's take a look at this example:

```ts
function returnAs<T>(arg: any): T {
  return arg // 🚨 an `any` that will _seem_ like a `T`
}

// 🚨 DANGER! 🚨
const first = returnAs<number>(window)
const sameAs = window as any as number
```

Try

In this example, we have told TypeScript a lie by saying `window` is a `number` (but it is not...). Now, TypeScript will fail to catch errors that it is suppose to be catching!

- Define type parameters as simply as possible. Consider the two options for `listToDict`:

```ts
interface HasId {
  id: string
}
interface Dict<T> {
  [k: string]: T
}

function ex1<T extends HasId[]>(list: T) {
  return list.pop()
}
function ex2<T extends HasId>(list: T[]) {
  return list.pop()
}
```

Try

Finally, only use type parameters when you have a real need for them. They introduce complexity, and you shouldn't be adding complexity to your code unless it is worth it!