
TS

The Course

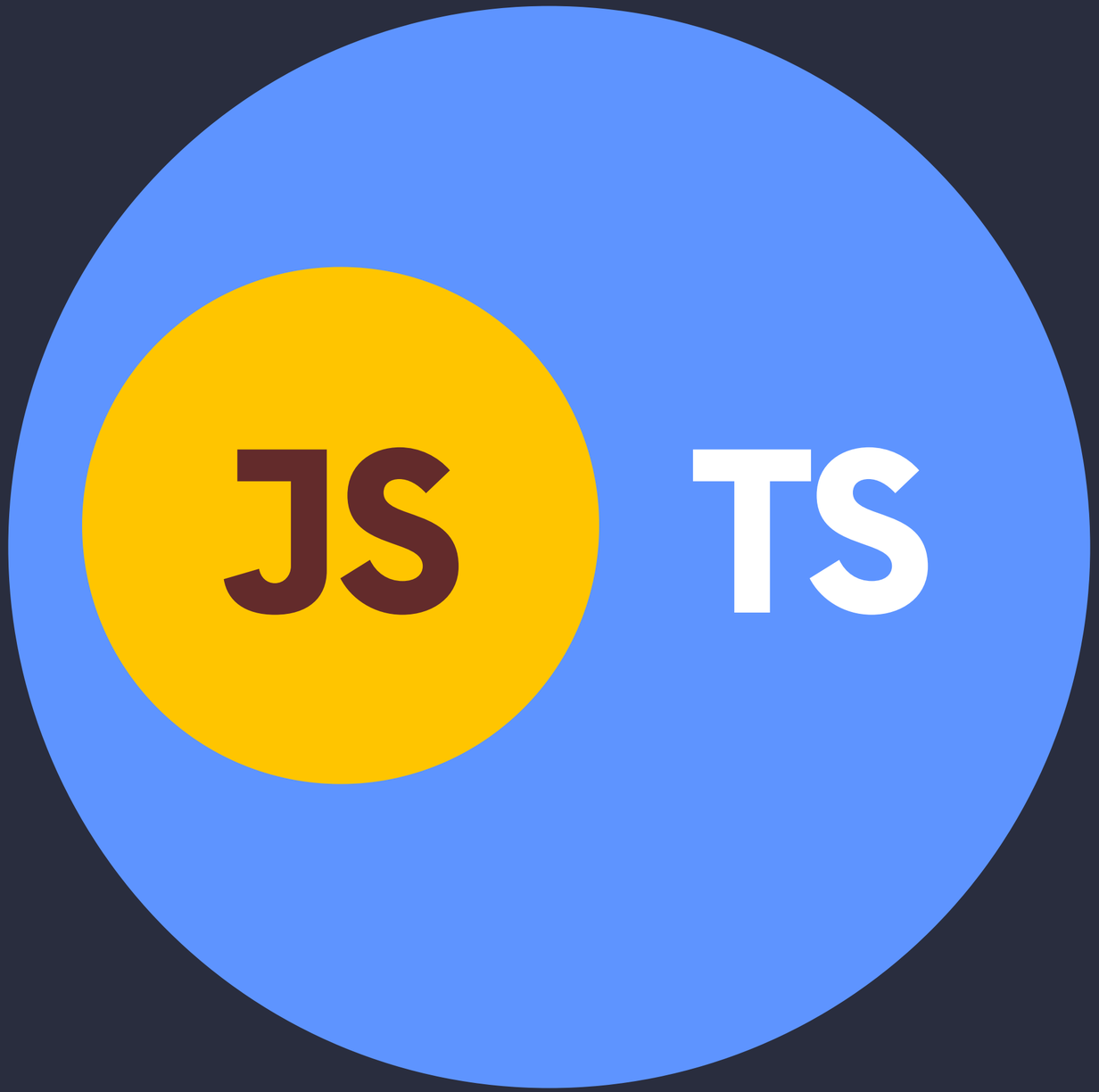
Installation	Union Types	JS Classes	Modules
Type Basics	Tuples/Enums	TS Classes	Webpack + TS
Functions	Interfaces	Generics	React + TS
Object Types	TS Compiler	Narrowing	
Array Types	DOM Mini Project	Declarations	

The Course

Installation	Union Types	JS Classes	Modules
Type Basics	Tuples/Enums	TS Classes	Webpack + TS
Functions	Interfaces	Generics	React + TS
Object Types	TS Compiler	Narrowing	
Array Types	DOM Mini Project	Declarations	

What is TypeScript?

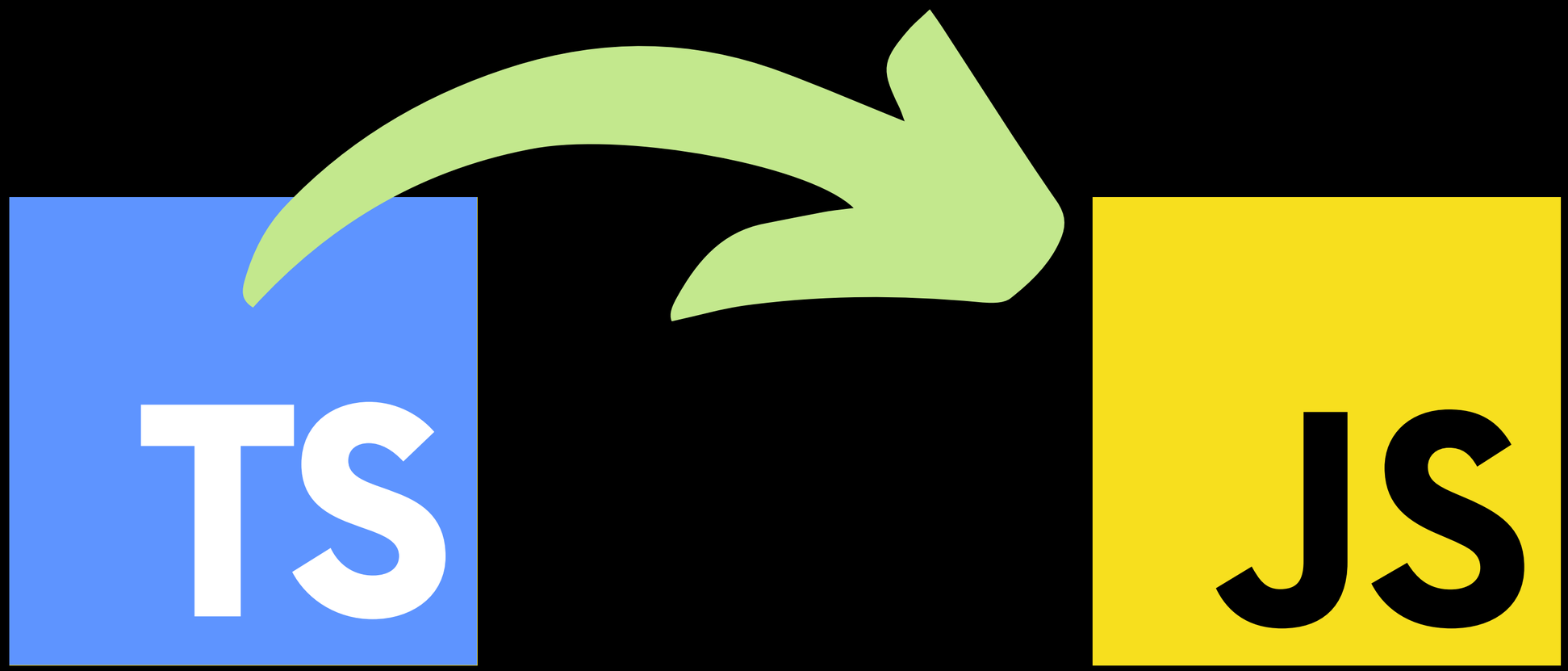
TypeScript is...
JavaScript with types


JS
TS

Why Types?

TypeScript's Type System...

- Helps us find errors!
- Analyzes our code as we type
- Only exists in development


TS
JS

JavaScript

Primitive Types

- Number
- String
- Boolean
- Null
- Undefined
- Void

Object Types

- Object
- Array
- Function

Primitive Types

Number	Null	Any
String	Undefined	Never
Boolean	Void	Unknown

Object Types

Object	Tuple
Array	Enum
Function	Others!

"This function returns a boolean"

"This function accepts two numbers and returns a number"

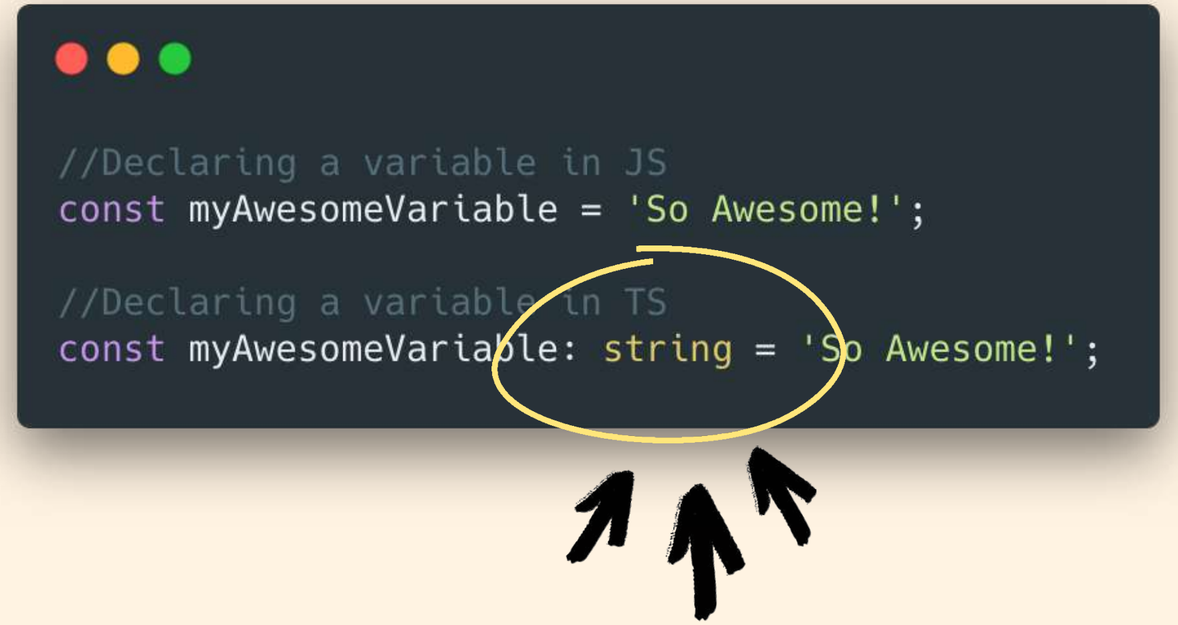
"This object must have a property called colors, set to an array of strings"

"This variable is a string"

Variable Types

Assigning a basic type to a variable is easy - just add **:Type** after the variable name!

(Also called '**Type Annotation**')



```
let myVar: type = value
```

Strings

Strings represent character values like
'I love Typescript!'

We can tell TypeScript that something is a string using the type annotation of **string** (all lowercase)

```
//Declaring a string variable
let myString: string = "Words!!!";

//CAN'T reassign to a different type
❌ myString = 100;

//CAN reassign to a value of same type
✅ myString = "New words!!!";
```

```
//Declaring a number variable
let myNumber: number = 42;

//CAN'T reassign to a different type
❌ myNumber = "I'm a string!";

//CAN reassign to a value of same type
✔ myNumber = 60;
```

Numbers

Some programming languages have many number types - **float**, **int**, etc., in Typescript (as well as Javascript) numbers are just numbers.

Numbers can be typed with a simple Type Annotation of **number** (all lowercase)

```
//Declaring a boolean variable
const myBoolean: boolean = true;

//CAN'T reassign to a different type
❌ myBoolean = 87;

//CAN reassign to a value of same type
✅ myBoolean = false;
```

Booleans

Boolean variables represent simple **true** or **false** values.

Booleans can be typed with a simple type annotation of **boolean** (all lowercase)

Type Inference

Type Inference refers to the Typescript compiler's ability to infer types from certain values in your code.

Typescript can remember a value's type even if you didn't provide a type annotation, and it will enforce that type moving forward.

```
// Creating a variable with a value,
// but without a type annotation
let x = 27;
x = 'Twenty-seven';
// ERROR - Type 'string' is not assignable
// to type 'number'.
```

Any

'any' is an escape hatch! It turns off type checking for this variable so you can do your thing.

NOTE: it sort of defeats the purpose of TS and types, so use it sparingly!

```
//Declaring a variable with type 'any'
const myComplicatedData: any = "I'm going to be complicated!";

//CAN reassign to any type -
//type checks are off for this var now!
   myComplicatedData = 87;
✓  myComplicatedData = 'abc...';
✓  myComplicatedData = true;
✓
```

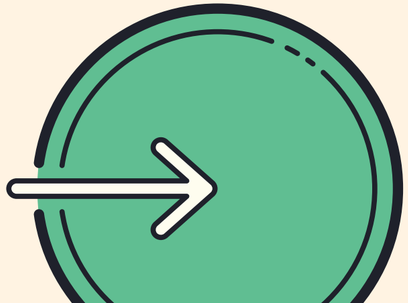
```
//Creating a function with typed args
const encourageStudent = (name: string) => {
  return `Hey, ${name}, you're doing GREAT!`;
}

//CAN call this function with a string
✓ encourageStudent('you');
    // --> 'Hey, you, you're doing GREAT!'

//CAN'T call this function with other type args
✗ encourageStudent(85);
    // --> Typescript error!!
```

Function Parameter types

In TypeScript, we can specify the type of function parameters in a function definition.

This allows Typescript to enforce the types for the values being passed into your function.

Typing parameters is just like typing variables!



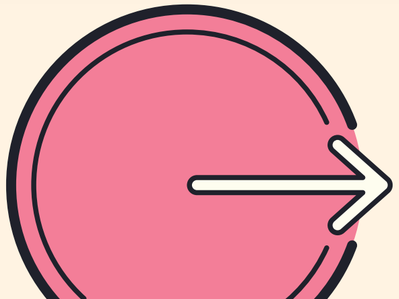
```
function greet(person: string){
   return ` Hi, ${person}!`
}
```

Function Return Types

We can specify the type returned by a function. Even though TypeScript can often infer this, I prefer the explicit annotations.

Add the type annotation after the function parameter list.

```
//Creating a function with a return type
const addNums = (x: number, y: number): number => {
  return x + y;
}

addNums(5, 5);
// --> 10
```



```ts
const numbers = [1, 2, 3];

// Contextual typing on an arrow function
numbers.forEach(num => {
    return num.toUppercase(); //Error!
    //.toUpperCase() doesn't work for nums
});
```

Anonymous Functions

When Typescript can infer how an unnamed function is going to be called, it can automatically infer its parameters' types.

```
//A function that doesn't return anything
const annoyUser = (num: number): void => {
    for(let i = 0; i < num; i++) {
      alert('HIIIIII!!');
    }
}
```

Void

Void is a return type for functions that don't return anything. It means just that - this function is void of any data.

Typescript can infer this type fairly well, but sometimes it may want you to annotate a function with a void return explicitly.

Never

The never type represents values that NEVER occur. We might use it to annotate a function that always throws an exception, or a function that never finishes executing.

Don't confuse with **void** - void returns undefined or null, which is technically still a type of value. With **never**, a function doesn't even finish executing.

```
//A function that doesn't finish running
const neverStop = (): never => {
    while(true) {
      console.log("I'm still going!");
    }
}

//A function that throws an exception
const giveError = (msg: string) => {
    throw new Error(msg);
}
```

{Object} + [Array] Types

Objects

Objects can be typed by declaring what the object should look like in the annotation.

Accessing a property that isn't defined or performing operations without keeping types in mind will throw errors!

```
//A function with an object type parameter
const printName = (name: { first: string; last: string }) => {
    return `Name: ${first} ${last}`;
}

printName({first: 'Will', last: 'Ferrell'});
//Name: Will Ferrell
```

```
//A type alias!
type Person = {
  name: string;
  age: number;
};

//Use the type alias in the annotation
const sayHappyBirthday = (person: Person) => {
    return `Hey ${person.name}, congrats on turning ${age}!`;
}

sayHappyBirthday({name: 'Jerry', age: 42});
```

Type Alias

Instead of writing out object types in an annotation, we can declare them separaretly in a **type alias,** which is simply the desired shape of the object.

This allows us to make our code more readable and even reuse the types elsewhere in our code.

Nested Objects

```
//A function with a nested object type parameter
const describePerson = (person: {
    name: string;
    age: number;
    parentNames: {
        mom: string;
        dad: string;
    }
}) => {
    return `Person: ${name},
    Age: ${age},
    parents: ${parentNames.mom}, ${parentNames.dad}.`;
}

describePerson({name: 'Jimmy', age: 10, parents: {mom: 'Kim', dad: 'Steve'}});
//Person: Jimmy, Age: 10, Parents: Kim, Steve
```

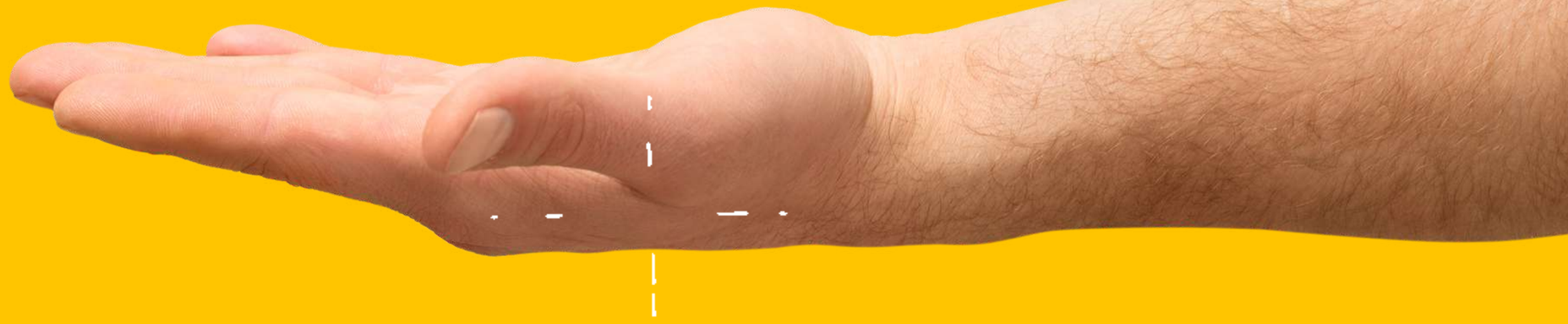
Array Types

Arrays can be typed using a type annotation followed by empty array brackets, like **number[]** for an array of numbers.

Note: these arrays only allow data of that one type inside them. More on that later!

```
//Using Brackets:
  //string array
  let names: string[] = ["hello", "world"];
  //number array
  let ages: number[] = [24, 32, 19, 29];

//An alternate syntax:
  //string array
  let names: Array<string> = ["hello", "world"];
  //number array
  let ages: Array<number> = [24, 32, 19, 29];
```

Other Types



```
//A function with a union type parameter
const guessAge = (age: number | string) => {
  return "Your age is: " + age);
}

// CAN pass a number or a string!
✓ guessAge(30);
✓ guessAge("28");

// CAN'T pass something else
✗ guessAge({ age: 32 });
```

Union Types

Union types allow us to give a value a few different possible types. If the eventual value's type is included, Typescript will be happy.

We can create a union type by using the single | **(pipe character)** to separate the types we want to include. It's like saying, "This thing is allowed to be this, this, or this". Typescript will enforce it from there.

Unions - Narrowing the Type

Narrowing the Type is simply doing a type check before working with a value. If your value is a string you may want to use it differently than if you got a number.

Since unions allow multiple types for a value, it's good to check what came through before working with it.

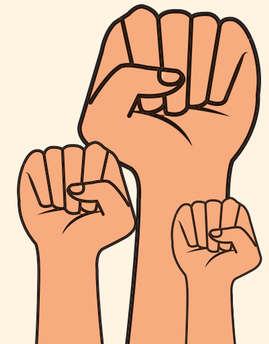
```
const isTeenager = (age: number | string) => {
  if (typeof age === "string") {
    // If age is a string, do this
    console.log(age.charAt(0) === 1);
  }
  if (typeof age === "number") {
    // If age is a number, do this
    console.log(age > 12 && age < 20);
  }
}

isTeenager('20'); //false
isTeenager(13); //true
```

Type Assertions

```
//Typescript infers a type of HTMLElement
const myPic = document.querySelector("profile-image");

//But we know a more specific type, so let's assert it!
const myPic = document.querySelector("profile-image") as HTMLImageElement;
```

Sometimes you might have more specific information about a value's type, and you want to make sure Typescript knows it too.

You can assert a value's type by using the **'as'** keyword, followed by the specific type you want to assert.

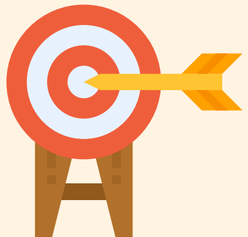


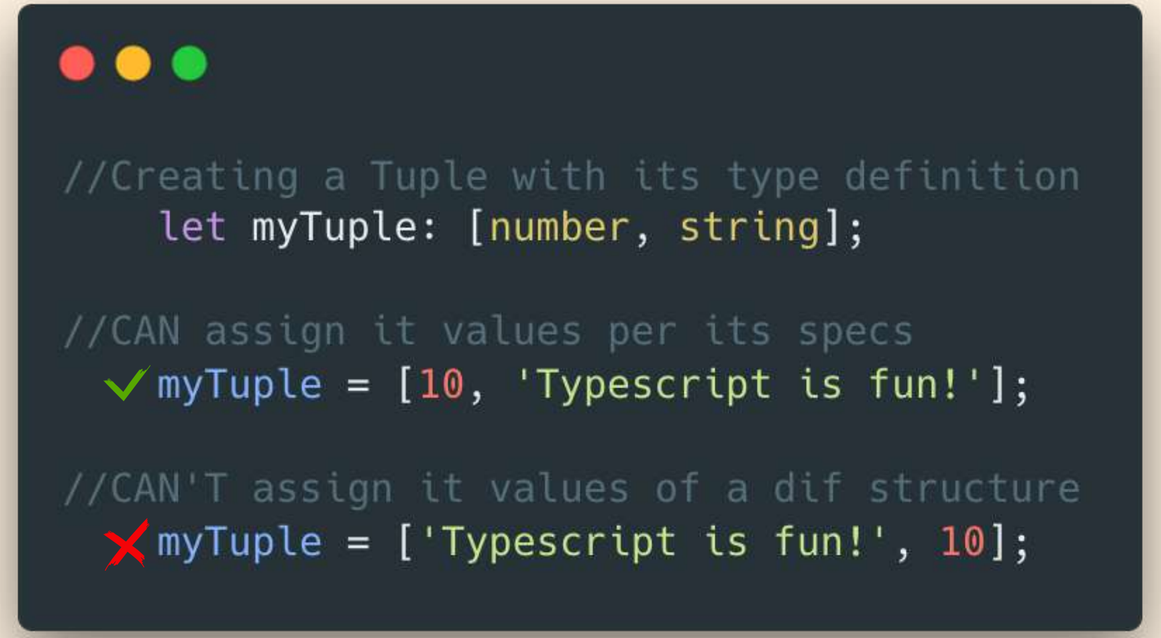
Literal Types

Literal types are not just types - but the values themselves too!

On it's own, they may not seem useful, but combine them with unions and you can have very fine-tuned type options for Typescript to enforce.

```typescript
//A function with a literal+union type parameter
const giveAnswer = (answer: "yes" | "no" | "maybe") => {
    return `The answer is ${answer}.`;
}

// CAN provide one of the literals in the union
✔ giveAnswer("no") // The anwer is no.

// CAN'T provide anything else
✘ giveAnswer("oh boy I'm not sure");
```





Tuples

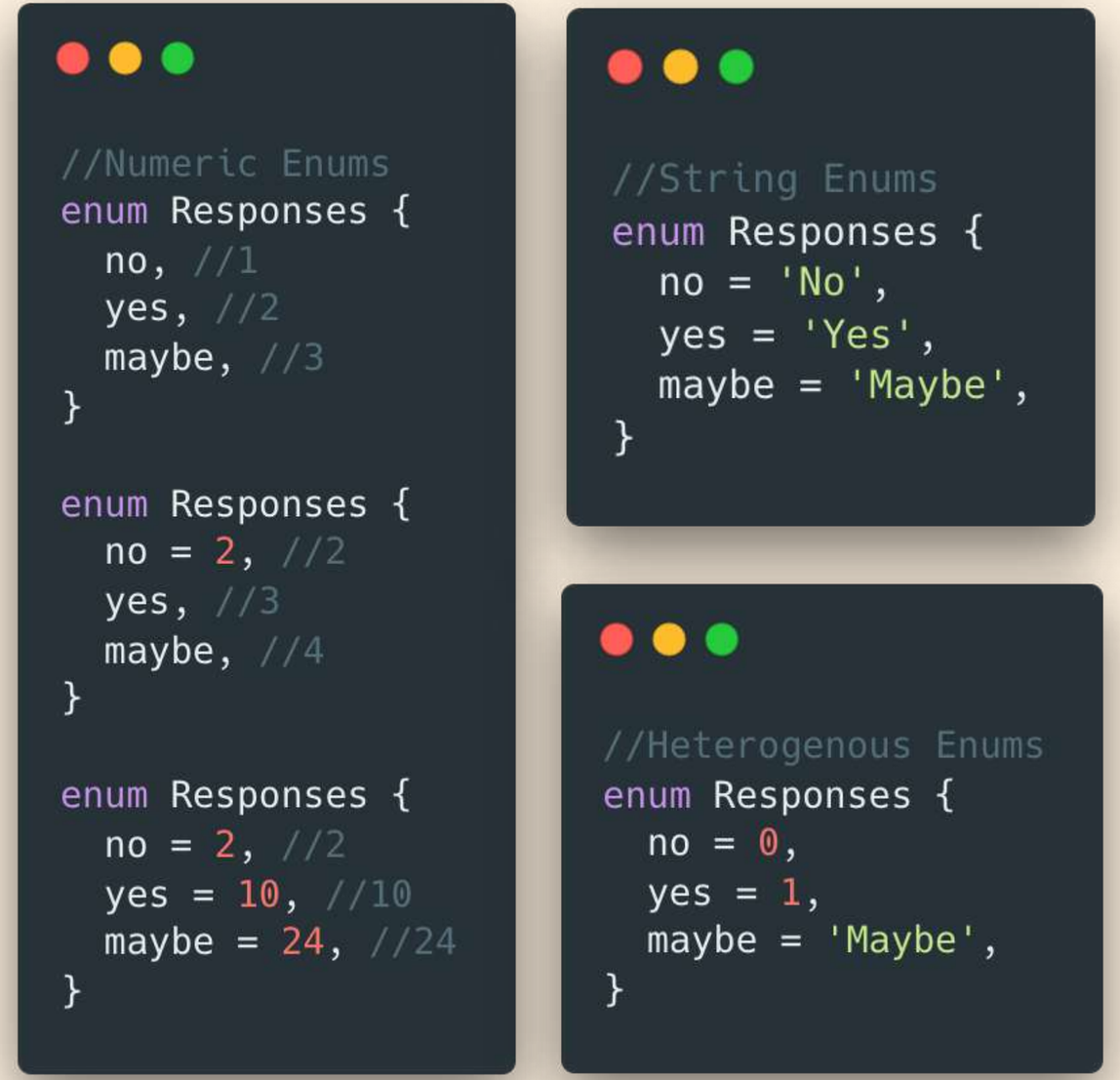
Tuples are a special type exclusive to TypeScript (they don't exist in JS)

Tuples are arrays of fixed lengths and ordered with specific types - like super rigid arrays.

Enums

Enums allow us to define a **set of named constants.** We can give these constants numeric or string values.

There's quite a few options when it comes to enums!



```
//Numeric Enums
enum Responses {
  no, //1
  yes, //2
  maybe, //3
}

enum Responses {
  no = 2, //2
  yes, //3
  maybe, //4
}

enum Responses {
  no = 2, //2
  yes = 10, //10
  maybe = 24, //24
}
```

```
//String Enums
enum Responses {
  no = 'No',
  yes = 'Yes',
  maybe = 'Maybe',
}
```

```
//Heterogenous Enums
enum Responses {
  no = 0,
  yes = 1,
  maybe = 'Maybe',
}
```

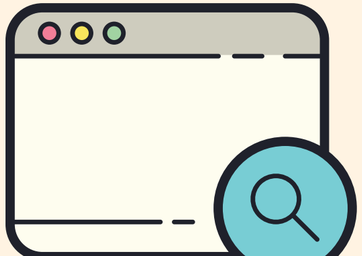
Aliases & Interfaces

Interfaces

Interfaces serve almost the exact same purpose as type aliases (with a slightly different syntax).

We can use them to create reusable, modular types that describe the **shapes of objects**

```
//An interface!
interface Person {
  name: string;
  age: number;
};

//Use the type alias in the annotation
const sayHappyBirthday = (person: Person) => {
    return `Hey ${person.name}, congrats on turning ${age}!`;
}

sayHappyBirthday({name: 'Jerry', age: 42});
```



Types vs. Interfaces

Adding new properties

Types

```ts
type Person = {
    name: string;
}

//Error! TS complains about duplicate types
type Person = {
    age: number;
}

//Error! Age didn't get added to Person type
const person: Person = {
    name: 'Jerry',
    age: 42
}
```

Interfaces

```ts
interface Person {
    name: string;
}

interface Person {
    age: number;
}

//Perfectly okay - Person has name & age
const person: Person = {
    name: 'Jerry',
    age: 42
}
```

Types vs. Interfaces

Extending Properties

Types

```ts
type Name = {
  name: string;
}

type Person = Name & {
  age: number;
}

//Person took Name and added an age property!
const person: Person = {
    name: 'Jerry',
    age: 42
}
```

Interfaces

```ts
interface Name {
  name: string;
}

interface Person extends Name {
  age: number;
}

//Person took Name and added an age property!
const person: Person = {
    name: 'Jerry',
    age: 42
}
```

Types vs. Interfaces

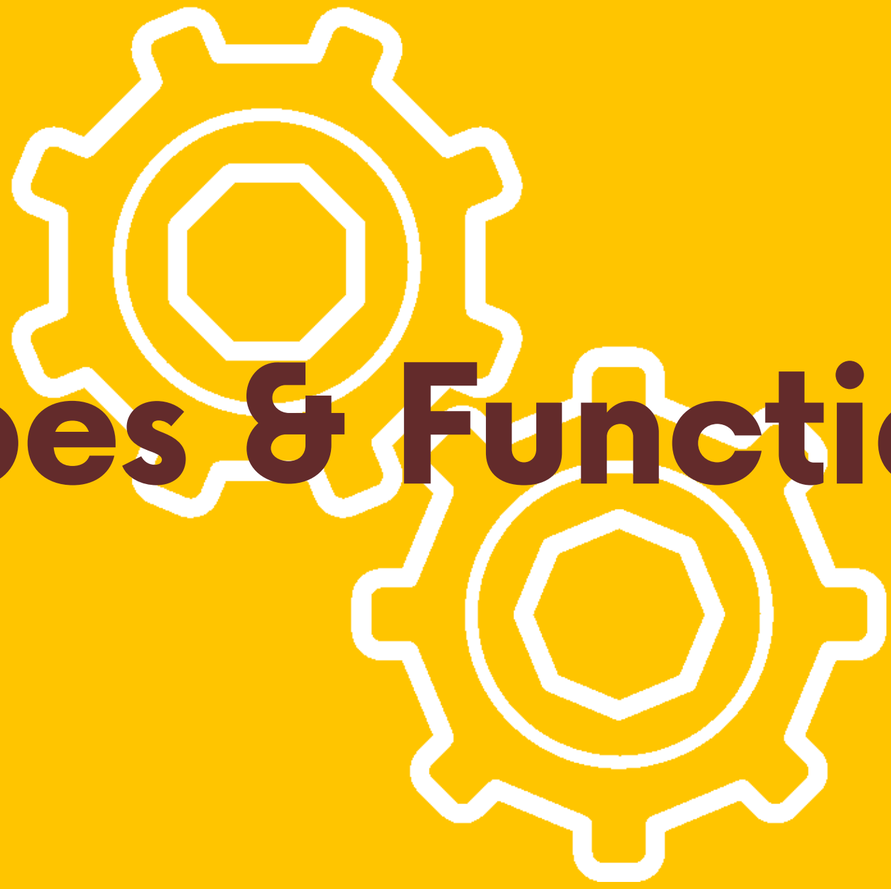
Optional Properties

Types

```type
//Optional Middle Name
type Person = {
    name: string;
    age: number;
    middleName?: string
}
```

Interfaces

```interface
//Optional Middle Name
interface Person {
    name: string;
    age: number;
    middleName?: string
}
```



Types & Functions

Any

'any' is an escape hatch! It turns off type checking for this variable / data so you can do your thing.

NOTE: it sort of defeats the purpose of TS and static types, so use it sparingly!

```
//Declaring a variable with type 'any'
const myComplicatedData: any = "I'm going to be complicated!";

//CAN reassign to any type -
//type checks are off for this var now!
   myComplicatedData = 87;
✓  myComplicatedData = 'abc...';
✓  myComplicatedData = true;
✓
```

Any (cont.)

Got a variable with an explicit type annotation? Assigning it to a variable with a type of **any** will turn type-checking off here and let it be reassigned, no problem.

```
//Can even assign other variables with different
//types to a variable with type 'any'
const myAnyVar: any = "String for now";
const myNumberVar: number = myAnyVar;

console.log(typeof myNumberVar); // --> string
```

```
//Declaring a variable with type 'unknown'
const unknownVar: unknown;

//CAN'T reassign a var of an explicit type
//to a value of unknown type
let myString: string = 'Howdy!';
mystring = unknownVar;
```

Unknown

Unknown is like a type-safe version of **any**. Variables typed as **unknown** can be assigned to values of any type.

However, variables with explicit types can't be assigned to **unknown** without using a type check first.

```
//Declaring a variable with type 'unknown'
const unknownVar: unknown;

//Do type checks when working with unknown
if(typeof unknownVar === 'string') {
    return unknownVar.charAt(2) // --> 'k'
}
```

Unknown (cont.)

It's important to **narrow the unknown type** when working with these type of variables, to stay type-safe.

Any vs. Unknown?

The Difference

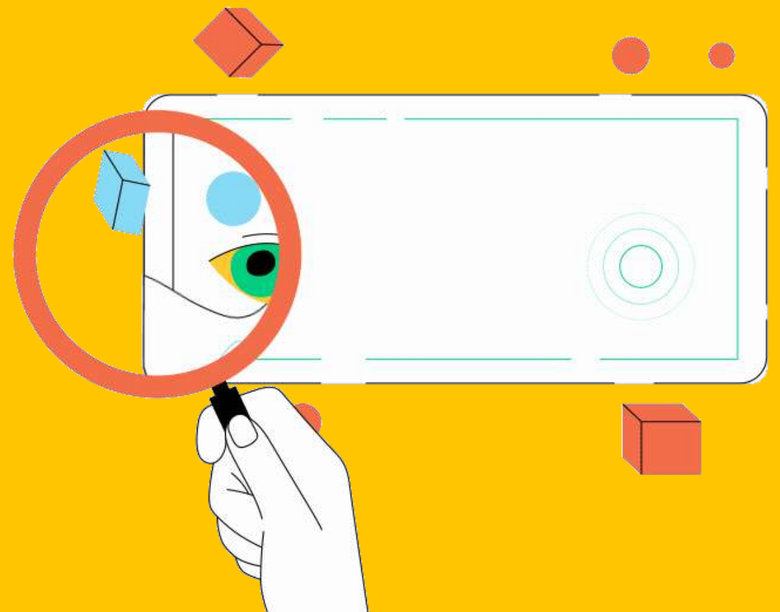
Unknown: "I don't know"

You don't know what the value type is just yet, but it's coming! To work with this variable, we'll do type-checks so we only do certain things depending on the type that comes back.

Any: "I don't care"

Couldn't care less what type this variable is or ends up being, for reasons Typescript won't understand. Turn off type-checking and let's get to work unobstructed.

Type Narrowing



Typeof Guards

typeof Type Guards involve simply doing a type check before working with a value.

Since unions allow multiple types for a value, we can first check what came through before working with it.

```ts
const isTeenager = (age: number | string) => {
  if (typeof age === "string") {
    // If age is a string, do this
    console.log(age.charAt(0) === 1);
  }
  if (typeof age === "number") {
    // If age is a number, do this
    console.log(age > 12 && age < 20);
  }
}

isTeenager('20'); //false
isTeenager(13); //true
```

```
const printLetters = (word: string | null) => {
  if (!word) {
    //If word is null, don't loop over it
    console.log('No word was provided.');
  } else {
    //Only loop if word exists/is truthy
    name.forEach(letter => console.log(letter));
  }
}
```

Truthiness Guards

Truthiness Type Guards involve checking a value for being truthy or falsy before working with it.

This is helpful in avoiding errors when values might be null or undefined.

Equality Narrowing

equality Type Guards involve comparing types to each other before doing certain operations with values.

By checking two values against one another, we can be sure they're both the same before working with them in a type-specific way.

```
const someFunc = (x: string | boolean, y: string | number) => {
  if (x === y) {
    // x and y are strings in this case
    x.toUpperCase();
    y.toLowerCase();
  } else {
    console.log(x);
    console.log(y);
  }
}
```

```ts
type Cat = { meow: () => void };
type Dog = { bark: () => void };

const talk = (creature: Cat | Dog) => {
  if ("meow" in creature) {
    console.log(creature.meow());
  } else {
    console.log(creature.bark());
  }
}

const kitty: Cat = {meow: () => 'MEOWWW'};
talk(kitty); //MEOWWW
```

in Operator Narrowing

Javascript's **in** operator helps check if a certain property exists in an object.

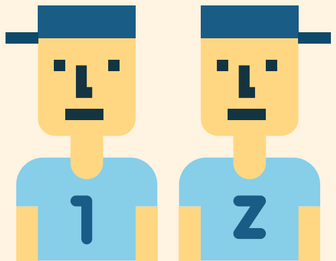
This means we can use it to check if a value exists in an object, according to its type alias or aliases, before working with it.

instanceof Narrowing

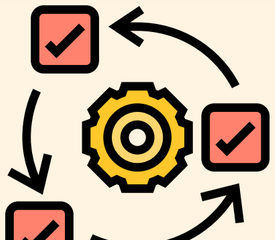
instanceof is a Javascript operator that allows us to check if one thing is an instance of another (remember **prototypes**?).

This can help us narrow types when working with things like classes.

```
const printFullDate(date: Date | string) {
    if (date instanceof Date) {
        return date.toUTCString();
    } else {
        return new Date(string).toUTCString();
    }
}
```



```
//": pet is dog" --> type predicate!
const isCat(pet: Cat | Dog): pet is Dog {
  return (pet as Dog).bark !== undefined;
}

let pet = getAnimal();
//pet gets passed to isCat above to narrow type
if (isCat(pet)) {
  pet.meow();
} else {
  pet.bark();
}
```



Type Predicates

Typescript allows us to write custom functions that can narrow the type of a value. These functions have a very special return type called a type predicate.

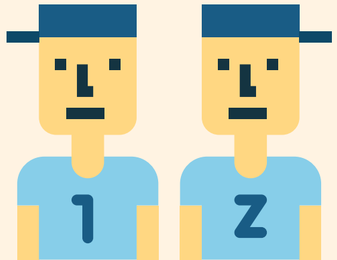
A predicate takes the form **parameterName is Type**

discriminated unions

A common pattern in Typescript involves creating a literal property that is common across multiple types.

We can then narrow the type using that literal property

```
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}
```



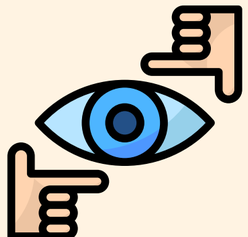
Back to {Objects} Types!

```ts
type SomeType {
    readonly first: string;
    readonly last: string;
}

const giveName = (name: SomeType) => {
  // We can read the values
  console.log(`Your name is now: ${name.first} ${name.last}`);

  // We can't re-assign them
  name.first = "Jimmy"; //ERROR
}
```

readonly Properties

Adding the **readonly** keyword before a type assignment locks that value down after everything compiles. In other words, you can **set** it but you can't **reassign** it!

Works in both type aliases & interfaces!



Index Signatures

Got some objects whose key names are TBD? Specify their types with an index signature!

Syntax: **[key: string]: string;**

```
//These 2 employees have different comp packages
const employee1 = {
  base: 100000,
  yearlyBonus: 20000
};
const employee2 = {
  contract: 110000
};

//But we can still compare their payment using an index sig.!
const totalComp = (salaryObject: { [key: string]: number }) => {
  let income = 0;
  for (const key in salaryObject) {
    income += salaryObject[key];
  }
  return income;
}

totalComp(employee1); // => 120,000
totalComp(employee2); // => 110,000
```

```ts
interface FullName {
  first: string;
  last: string;
}

interface ActualFullName extends FullName {
  middle: string;
  secondMiddleName?: string;
}
```

Extending Types

Got an interface of types? You can extend it into a new interface, adding the old types to the newly declared ones.

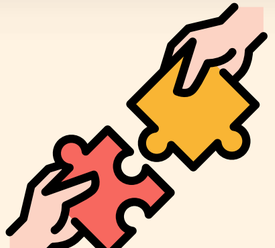
This is helpful if you want to add types to an existing object type.

Intersection Types

Intersections can be created using type aliases, combining two object types together.

This is a nice way to create new types from existing types, or add new types to existing types.

```ts
interface Kids {
  sons?: number;
  daughters?: number;
}
interface Parents {
  moms?: number;
  dads?: number;
}

type Family = Parents & Kids;
```



Type Aliases vs. Interfaces

Type Aliases - Intersection

```ts
type Name = {
  name: string;
}

type Person = Name & {
  age: number;
}

//Person took Name and added an age property!
const person: Person = {
    name: 'Jerry',
    age: 42
}
```

Interfaces - Extending

```ts
interface Name {
  name: string;
}

interface Person extends Name {
  age: number;
}

//Person took Name and added an age property!
const person: Person = {
    name: 'Jerry',
    age: 42
}
```

Type Aliases vs. Interfaces

Type Aliases - Intersection

```ts
type Person = {
    name: string;
};

type NewPerson = Person & {
    name: number;
};

const newKid: NewPerson = {
    name: "Jerry", //ERROR!
};
```

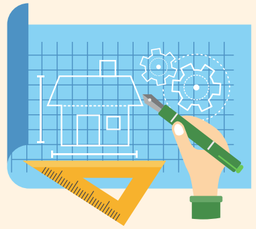
Interfaces - Extending

```ts
interface Person {
    name: string;
}

interface NewPerson extends Person {
    name: number;
}

const newKid: NewPerson = {
    name: "Jerry", //ERROR!
};
```

A Recap of JS Classes

Classes are templates for creating objects in Javascript. They contain a few different important pieces which allow for creation and extension of customized (and nicely organized) objects.

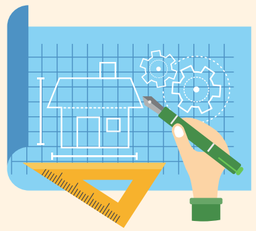
```
//Our blueprint for a person
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        return `Hello ${this.name}!`;
    }
}

//Using our blueprint to make a real person
const jimmy = new Person('Jimmy', 25);

//Using some methods that our new object has
//based on the blueprints
jimmy.greet(); //Hello Jimmy!
```



A Recap of JS Classes

- constructors
- class fields
- getters and setters
- private # fields
- static fields/methods
- inheritance
- super()

```
//Our blueprint for a person
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        return `Hello ${this.name}!`;
    }
}

//Using our blueprint to make a real person
const jimmy = new Person('Jimmy', 25);

//Using some methods that our new object has
//based on the blueprints
jimmy.greet(); //Hello Jimmy!
```



Generics

Generics

Generics allow us to define reusable functions and classes that work with multiple types rather than a single type.

The syntax is...not pretty. They are used all over the place, so it's best to get comfortable with them :)

```
function wrapInArray<T>(element: T): T[] {
  return [element];
}
```