name of your class ?

↓

All type equivalence questions are based on the name of the associated class.

with structural type systems all objects with the shape or structure matching that of the parameter will be accepted.

*) Duck typing : gets its name from the duck test i.e. "If it looks like a duck, swims like a duck, and quack like a duck, then it is probably a duck."

↓

similar to structural typing → but used for dynamic type systems.

↓

in the sense that they have no concern for whether you came out of a class or an object literal as long as you have what is needed to carry out the operation.

*) strong vs weak types

no agreed upon technical definition. context of typescript,
strong generally means static & weak refers to dynamic

→ Union & Intersection Types

·) can be thought of as logical boolean operators

·) union: OR for types → represented using '|'

intersection: AND for types → represented using '&'

·) extending to tuples structured as follows:

[0]: either "success" or "failure"

[1]: depending on the value of [0]

·) { name: string; email: string }

·) "error" case: an Error instance

OR in the sense that some or all values from different types, similar to how the boolean OR behaves.

↓

common pattern used whenever we're collecting errors and returning them instead of throwing them and interrupting the execution of the program.

↓

```
function maybeGetUserInfo(): 
    | ["error", Error]
    | ["success", { name: string; email: string }] {};

const output = maybeGetUserInfo();
```

return types

extra pipe ← auto code formatting ↓ not harmful ↓ but also not required

const [first, second] = outcome;

↓

on trying to access fields on these variables, we see that we can only safely access the fields that are guaranteed to be on the union, i.e present on all types we're taking a union of

↓

i.e. when a value has a type that has a type that includes a union, then we're only able to use the common behaviour that's guaranteed to be here.

↓

This is addressed using type narrowing.

•) Narrowing with type guards : branching code using type guards to narrow down on our fields that are accessible depending on the type we narrow done in a specific branch of code.

↓

type guards connect build time validation with run time behaviour, such that at runtime we'll only enter a branch of code if the type predicate is true or validated.

↓

```
if ( second instanceOf Error ) {
        // second has type Error
} else {
        // second has user info type
}
```

type guard

·) Discriminated unions

putting a check on one of the values, typescript is able to link & interpret the second

```
if ( output[0] === "error" ) {
        // output of the type ["error", Error]
} else {
        // output of the type ["success", `userInfo`]
}
```

discriminated union

↓

we have a convenient key to use in combination with a type guard that allows us to switch b/w different possibilities.

↓

property on object, value in a type

↓

also referred to as a tagged union type

•) intersection types represented by & includes value from all included types as required, whereas with union all values are optional.

AND in the sense that all values should be present, like for a if condition with multiple predicates combined using &, all have to be true.

↓

If 2 types have the member of different type, then it's automatically marked as never.

↓

If a value is assigned, then an issue, and if not assigned a value, then an issue.

↓

∴, we should never & on types that have same member of different type.

→ Type aliases & Interfaces

•) way to give friendly name to our types. (still remember that we are operating in a structural type system)

•) : { name : string ; email : strang } syntax gets more complicated as more properties are added & so on. Increased possibility of issues. Not DRY.

↓

types aliases allow for reusing of type definitions by allowing us to:

i) define a more meaningful name for a type

ii) declare the particulars at one place

iii) import & export types from modules

↓

titleCase

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

→ will disappear as part of the build process

•) at the end of the day, structure needs to match up.

•) can only have one type alias of a given name in a given scope. Isn't true for interfaces.

•) Inheritance : taking an existing type alias and create something that builds on top of it. (pseudo inheritance)

↓

using `&` types.

•) Interfaces, more limited than type aliases, in that they can only be used to define object types;

↓

like an instance of a class or an object with props

↓

union make something not an object type

•) Inheritance

typescript uses heritage clauses to describe ancestory in an object oriented hierarchy of sort.

↓

extends : describe inheritance b/w like things

implements : describe inheritance b/w unlike things

↓

classes can extend from classes, interfaces can extend from interfaces

↓

classes implements interfaces

```
interface AnimalLike {
    eat(food): void
}

class Dog implements AnimalLike {
    eat(food) {
        console.log(food);
    }
```

- Typescript doesn't support the multiple inheritance (extending from more than one class), but `implements` can be done to validate interface contracts across one or more interfaces.

↓

Also, both extends and implements can be used together.

↓

```
class Dog
    extends LivingOrganism
    implements AnimalLike, CanBark
{ }
```

- possible to implement type alias, but if it deviates from the object type rules, then it can cause trouble.

•) Open Interfaces : typescript interfaces are open, meaning that unlike type aliases, we can have multiple declarations in the same scope. The declarations would basically augment over each other or be merged in a way.

↓

This can be used to extend existing interfaces, like adding a property on the window interface, available to be used throughout the project

↓

consuming some library and the type information is incomplete, and we want to tag something on.

↓

i.e. augmenting an interface.

•) choosing which one to use

i) most situations, either one is okay

ii) anything that doesn't align with the object type, use type aliases

iii) use interfaces if we want to employ 'implements'

iv) to allow consumers to our types to augment them, use an interface.

•) Recursive types : self referential, and are often used to define infinitely nestable types.

↓

```
type NestedNumbers = number | NestedNumbers[]
```

- JSON types exercise