Step 1 - Pre-requisites

Before you go through this module, make sure you've gone through basic typescript classes. You understand `interfaces` , `types` and how typescript is used in a simple Node.js application

If you understand the following code, you're good to go!

```
interface User {
    name: string;
    age: number;
}

function sumOfAge(user1: User, user2: User) {
  return a.age + b.age;
};

// Example usage
const result = sumOfAge({
    name: "harkirat",
    age: 20
}, {
    name: "raman",
    age: 21
});
console.log(result); // Output: 9
```

Copy

Recap setup procedure

To recap, if you want to start a Typescript project locally, please do the following -

1. Initialize TS

```
npx tsc --init
```

Copy

1. Change tsconfig as per your needs. Usually changing `rootDir` and `outDir` is a good idea

```
{
    "rootDir": "./src",
    "outDir": "./dist"
}
```

Copy

Pick

`Pick` allows you to create a new type by selecting a set of properties (`Keys`) from an existing type (`Type`).

Imagine you have a User model with several properties, but for a user profile display, you only need a subset of these properties.

```
interface User {
  id: number;
  name: string;
  email: string;
  createdAt: Date;
}

// For a profile display, only pick `name` and `email`
type UserProfile = Pick<User, 'name' | 'email'>;

const displayUserProfile = (user: UserProfile) => {
  console.log(`Name: ${user.name}, Email: ${user.email}`);
};
```

Copy

Partial

`Partial` makes all properties of a type optional, creating a type with the same properties, but each marked as optional.

Specifically useful when you want to do `updates`

```
interface User {
    id: string;
    name: string;
    age: string;
    email: string;
    password: string;
};

type UpdateProps = Pick<User, 'name' | 'age' | 'email'>

type UpdatePropsOptional = Partial<UpdateProps>

function updateUser(updatedProps: UpdatePropsOptional) {
    // hit the database tp update the user
}
updateUser({})
```

Copy

Readonly

When you have a configuration object that should not be altered after initialization, making it `Readonly` ensures its properties cannot be changed.

```
interface Config {
  readonly endpoint: string;
  readonly apiKey: string;
}

const config: Readonly<Config> = {
  endpoint: 'https://api.example.com',
  apiKey: 'abcdef123456',
};

// config.apiKey = 'newkey'; // Error: Cannot assign to 'apiKey' because it is a rea
```

Cop

This is compile time checking, not runtime (unlike const)

`Record` and Map

Record

`Record` let's you give a cleaner type to objects

You can type objects like follows -

```
interface User {
  id: string;
  name: string;
}

type Users = { [key: string]: User };

const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};
```

Copy

or use `Record`

```
interface User {
  id: string;
  name: string;
}

type Users = Record<string, User>;

const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};

console.log(users['abc123']); // Output: { id: 'abc123', name: 'John Doe' }
```

Copy

Map

maps gives you an even fancier way to deal with objects. Very similar to `Maps` in C++

```
interface User {
  id: string;
  name: string;
}

// Initialize an empty Map
const usersMap = new Map<string, User>();

// Add users to the map using .set
usersMap.set('abc123', { id: 'abc123', name: 'John Doe' });
usersMap.set('xyz789', { id: 'xyz789', name: 'Jane Doe' });

// Accessing a value using .get
console.log(usersMap.get('abc123')); // Output: { id: 'abc123', name: 'John Doe' }
```

Exclude

In a function that can accept several types of inputs but you want to exclude specific types from being passed to it.

```
type Event = 'click' | 'scroll' | 'mousemove';
type ExcludeEvent = Exclude<Event, 'scroll'>; // 'click' | 'mousemove'

const handleEvent = (event: ExcludeEvent) => {
  console.log(`Handling event: ${event}`);
};

handleEvent('click'); // OK
```

Type inference in zod

When using zod, we're done runtime validation.
For example, the following code makes sure that the user is sending the right inputs to update their profile information

```
import { z } from 'zod';
import express from "express";

const app = express();

// Define the schema for profile update
const userProfileSchema = z.object({
  name: z.string().min(1, { message: "Name cannot be empty" }),
  email: z.string().email({ message: "Invalid email format" }),
  age: z.number().min(18, { message: "You must be at least 18 years old" }).optional
});

app.put("/user", (req, res) => {
  const { success } = userProfileSchema.safeParse(req.body);
  const updateBody = req.body; // how to assign a type to updateBody?

  if (!success) {
    res.status(411).json({});
    return
  }
  // update database here
```

```
  res.json({
    message: "User updated"
  })
});

app.listen(3000);
```

More details - https://zod.dev/?id=type-inference