Basics

1. Introduction to TypeScript

Exercise:

Write a TypeScript program that prints "Hello, TypeScript!" to the console.

Solution:

```typescript
// hello.ts
console.log("Hello, TypeScript!");
```

To compile and run:

```sh
tsc hello.ts
node hello.js
```

2. Type Annotations

Exercise:

Create a TypeScript function that accepts a string and a number and returns a formatted string.

Solution:

```typescript
function formatMessage(message: string, id: number): string {
    return `Message: ${message}, ID: ${id}`;
}

console.log(formatMessage("Hello", 1));
```

Advanced Types

3. Interfaces

Exercise:

Define an interface for a person with `name` and `age` properties and create a function that accepts this interface and prints a greeting message.

Solution:

```typescript
interface Person {
    name: string;
    age: number;
}

function greet(person: Person): void {
    console.log(`Hello, ${person.name}! You are ${person.age} years old.`);
}

const person: Person = { name: "John", age: 30 };
greet(person);
```

4. Classes

Exercise:

Create a class `Animal` with properties `name` and `sound` and a method `makeSound`. Then, create a subclass `Dog` that overrides the `makeSound` method.

Solution:

```typescript
class Animal {
    name: string;
    sound: string;

    constructor(name: string, sound: string) {
        this.name = name;
        this.sound = sound;
    }

    makeSound(): void {
        console.log(`${this.name} says ${this.sound}`);
    }
}

class Dog extends Animal {
    constructor(name: string) {
        super(name, "Woof");
    }
```

```
    makeSound(): void {
        console.log(`${this.name} barks: ${this.sound}`);
    }
}

const dog = new Dog("Rex");
dog.makeSound();
```

5. Functions

Exercise:

Write a TypeScript function `multiply` with default parameters that multiplies two numbers. If the second number is not provided, it should multiply the first number by 2.

Solution:

```typescript
function multiply(a: number, b: number = 2): number {
    return a * b;
}
```

```
console.log(multiply(5)); // 10
console.log(multiply(5, 3)); // 15
```

Type Features

6. Generics

Exercise:

Create a generic function that returns the length of an array of any type.

Solution:

```typescript
function getArrayLength<T>(arr: T[]): number {
  return arr.length;
}

console.log(getArrayLength([1, 2, 3])); // 3
console.log(getArrayLength(["a", "b", "c"])); // 3
```

7. Modules

Exercise:

Create two modules, one exporting a function and the other importing and using it.

Solution:

```typescript
// math.ts
export function add(x: number, y: number): number {
    return x + y;
}

// app.ts
import { add } from "./math";
console.log(add(2, 3)); // 5
```

8. Type Assertions

Exercise:

Write a TypeScript function that accepts a variable of type `any` and returns its length if it's a string.

Solution:

```typescript
function getStringLength(value: any): number | undefined {
    if (typeof value === "string") {
        return (value as string).length;
    }
    return undefined;
}

console.log(getStringLength("Hello")); // 5
console.log(getStringLength(123)); // undefined
```

9. Utility Types

Exercise:

Create an interface `Todo` and use the `Partial` utility type to create a function that updates a `Todo` object.

Solution:

```typescript
interface Todo {
    title: string;
    description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>): Todo {
    return { ...todo, ...fieldsToUpdate };
}

const todo1: Todo = { title: "Learn TypeScript", description: "Study TypeScript utility types" };
const todo2 = updateTodo(todo1, { description: "Master TypeScript" });

console.log(todo2);
```

Advanced Topics

10. Decorators

Exercise:

Create a class decorator that logs the creation of an instance of a class.

Solution:

```typescript
function logClass(constructor: Function) {
    console.log(`Class ${constructor.name} is created`);
}

@logClass
class Person {
    constructor(public name: string) {}
}

const person = new Person("John");
```

11. Mixins

Exercise:

Create a mixin that adds a `timestamp` property to a class and use it in another class.

Solution:

```typescript
type Constructor<T = {}> = new (...args: any[]) => T;

function Timestamped<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    timestamp = new Date();
  };
}

class User {
  constructor(public name: string) {}
}
```

```
const TimestampedUser = Timestamped(User);

const user = new TimestampedUser("John");
console.log(user.name); // John
console.log(user.timestamp); // Current timestamp
```

12. Namespaces and Modules

Exercise:

Create a namespace `Shapes` with a class `Circle` and use it in a program.

Solution:

```typescript
namespace Shapes {
    export class Circle {
        constructor(public radius: number) {}
        getArea(): number {
            return Math.PI * this.radius ** 2;
        }
    }
```

```
}

const circle = new Shapes.Circle(10);
console.log(circle.getArea()); // 314.159...
```

13. Type Guards

Exercise:

Create a type guard function that checks if a variable is a number.

Solution:

```typescript
function isNumber(value: any): value is number {
    return typeof value === "number";
}

function checkValue(value: any) {
    if (isNumber(value)) {
        console.log(`${value} is a number`);
    } else {
```

```
        console.log(`${value} is not a number`);
    }
}

checkValue(123); // 123 is a number
checkValue("Hello"); // Hello is not a number
```

14. Advanced Types and Concepts

Exercise:

Create a function that uses union types and a discriminated union to handle different shapes (circle and square).

Solution:

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

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

type Shape = Circle | Square;

function getArea(shape: Shape): number {
    switch (shape.kind) {
        case "circle":
            return Math.PI * shape.radius ** 2;
        case "square":
            return shape.sideLength ** 2;
    }
}

const circle: Circle = { kind: "circle", radius: 10 };
const square: Square = { kind: "square", sideLength:
5 };

console.log(getArea(circle)); // 314.159...
```

```
console.log(getArea(square)); // 25
```

Integration and Tools

15. Tooling and Frameworks

Exercise:

Set up a simple TypeScript project using Node.js. Create a `tsconfig.json` file and compile a TypeScript file.

Solution:

1. Initialize a new Node.js project:

   ```sh
   npm init -y
   ```

2. Install TypeScript:

   ```sh
   npm install typescript --save-dev
   ```

3. Create a `tsconfig.json` file:

```json
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "./dist"
  },
  "include": ["src"]
}
```

4. Create a `src` folder and add a TypeScript file (`src/index.ts`):

```typescript
const message: string = "Hello, Node.js with TypeScript!";
console.log(message);
```

```

5. Compile and run the TypeScript file:
   ```sh
   npx tsc
   node dist/index.js
   ```

16. Configuration and Compilation

Exercise:

Create a `tsconfig.json` file and configure it to compile TypeScript files into a `dist` folder.

Solution:

1. Create a `tsconfig.json` file:
   ```json
   {
     "compilerOptions": {
       "target": "ES5",
       "module": "commonjs",
       "strict": true,

```
    "esModuleInterop": true,
    "outDir": "./dist"
  },
  "include": ["src"]
}
```

2. Create a `src` folder and add a TypeScript file (`src/index.ts`):

```typescript
const message: string = "Hello, TypeScript!";
console.log(message);
```

3. Compile the TypeScript files:

```sh
npx tsc
```

4. Run the compiled JavaScript file:

```sh
```

```
node dist/index.js
```

```
```

17. Migration to TypeScript

Exercise:

Migrate a simple JavaScript function to TypeScript and add type annotations.

Solution:

1. JavaScript file (`sum.js`):

```javascript
function sum(a, b) {
    return a + b;
}

console.log(sum(2, 3));
```

2. Rename the file to `sum.ts` and add type annotations:

```typescript
function sum(a: number, b: number): number {
    return a + b;
}

console.log(sum(2, 3));
```

3. Compile and run:

```sh
tsc sum.ts
node sum.js
```

Best Practices

18. Best Practices

Exercise:

Create a TypeScript program following best practices: use strict types, avoid `any`, and organize code into modules.

Solution:

1. Create a `tsconfig.json` file with strict type checking:

```json
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "./dist"
  },
  "include": ["src"]
}
```

2. Create a `src` folder and add a TypeScript file (`src/index.ts`):

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

function greet(user: User): string {
    return `Hello, ${user.name}! You are ${user.age} years old.`;
}

const user: User = { name: "John", age: 30 };
console.log(greet(user));
```

3. Compile and run:
```sh
npx tsc
node dist/index.js
```