// code-1 type Fish = {swim: () => void}; type Bird = {fly: () => void}; function isFish(pet: Fish | Bird): pet is Fish{ return (pet as Fish).swim !== undefined } function getFood(pet: Fish | Bird){ if (isFish(pet)) { pet return "fish food" e}se { pet return "bird Food" }}// code-2 interface Circle { kind: "circle", radius: number } interface Square { kind: "square" side: number } interface Rectangle { kind: "rectangle", length: number, width: number } type Shape = Circle | Square | Rectangle function getTrueShape(shape: Shape){ if (shape.kind === "circle") { return Math.PI * shape.radius ** 2 } re/turn shape.side * shape.side } function getArea(shape: Shape){ switch(shape.kind){ case "circle": return Math.PI * shape.radius ** 2 case "square": return shape.side * shape.side case "rectangle": return shape.length * shape.width default: const _defaultforshape: never = shape return _defaultforshape }}explain me following thing from code-1 - isFish function and from code-2 _defaultForShape:never in deep dive with simple explanation

---

Let me explain both concepts in a simple way with examples.

Code-1: `isFish` function (Type Predicate)

The `isFish` function is a special kind of function called a "type predicate" in TypeScript. Its purpose is to help TypeScript understand what type a variable is at runtime.

typescript

Copy

Download

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (pet as Fish).swim !== undefined
}
```

Key points:

1. The `pet is Fish` return type tells TypeScript: "If this function returns true, then the argument is definitely a Fish"
2. Inside the function, we check if the `swim` property exists (by doing a type assertion with `as Fish`)
3. When we use this function in an `if` condition, TypeScript will automatically narrow the type inside the `if` block

Example usage:

typescript

Copy

Download

```
let myPet: Fish | Bird = { swim: () => console.log("swimming") };

if (isFish(myPet)) {
    myPet.swim();  // TypeScript knows myPet is Fish here
} else {
    myPet.fly();   // TypeScript knows myPet is Bird here
}
```

Code-2: `_defaultforshape: never` (Exhaustiveness Checking)

This is a pattern used to ensure you've handled all possible cases in a union type. The `never` type represents something that should never occur.

typescript

Copy

Download

```
default:
```

```
default:
    const _defaultforshape: never = shape
    return _defaultforshape
```

How it works:

1. `never` is a type that can't have any value (it represents unreachable code)
2. In the `switch` statement, if all cases are handled, `shape` should never reach the `default` case
3. If you later add a new shape (like `Triangle`) but forget to handle it in the `switch`, TypeScript will give you an error because `shape` could be `Triangle`, which can't be assigned to `never`

Example of why it's useful:

typescript

Copy

Download

```
// Later, if we add a new shape:
interface Triangle {
    kind: "triangle",
    base: number,
    height: number
}

type Shape = Circle | Square | Rectangle | Triangle;

// Now TypeScript will give an error in the default case because
// shape could be Triangle, but we can't assign Triangle to never
```

This pattern ensures your code stays type-safe even when the types evolve, forcing you to handle all possible cases. It's like a safety net that catches you if you forget to update your code when adding new types.