Workshop 2: The TypeScript Pivot

Theme: Pokemon Kart 64
Time Limit: 90 Minutes
Goal: Refactor a messy JavaScript game into robust TypeScript.

Overview

We have a working racing game, but the codebase is fragile. It relies on "Magic Strings," has logic bugs (Special Abilities aren't firing!), and uses loose typing.

Your mission is to port the game to **TypeScript** to fix these bugs and make the code "Enterprise Ready" (or at least "Race Ready").

Learning Objectives

- **Migration:** Moving from `.js` to `.ts`.
- **Inheritance:** Using `abstract` classes to enforce structure.
- **Type Narrowing:** Using `instanceof` to access specific class methods safely.
- **Enums:** Replacing string states ("racing") with explicit Enums.
- **Strict Null Checks:** Handling DOM elements that might not exist.
- **Union Types:** Defining strict options for variables.

Setup

1. Create a new Vite project: `npm create vite@latest ts-kart -- --template vanilla-ts`
2. Delete the default files from `/src`
3. Copy the provided `index.html`, `style.css` and `game.js` into your project.
4. Run `npm install canvas-confetti` and `npm install --save @types/canvas-confetti`
5. Run `npm run dev`.

The Mission

Task 1: The Crash (Rename)

Rename `main.js` to `main.ts`.

- **Observation:** Your editor will light up with red errors. This is good! It means TypeScript is working.
- **Fix:** Add type annotations to function parameters where implicitly `any` is forbidden.
    - *Example:* `constructor(name: string, icon: string, laneId: number)`

Task 2: The Logic Bug (Narrowing)

If you watch the race, **Pikachu never spins** and **Charizard never roars**.

- **The Problem:** Look at the `gameLoop`. It checks `if (driver.name === "Mario")`. But the class is named `Pikachu`! Using strings for logic is dangerous.
- **The Fix:**
    1. Remove the string checks.
    2. Use **Type Narrowing** (`instanceof`) to check the class type.
    3. Once you check `if (driver instanceof Pikachu)`, TypeScript will allow you to call `driver.useStar()` because it knows the type.
    4. Add a random superpower to both Jigglypuff and Bulbasaur, and make use of `instanceof` to call that superpower

```
// Example of Narrowing
if (driver instanceof Pikachu) {
  driver.useStar(); // TS knows this method exists!
}
```

Task 3: State Management (Enums)

Currently, `driver.state` is just a string ("idle", "racing", "finished"). If you typo this string, the car freezes.

- **The Fix:**
    1. Create an Union type: `type RaceState = "idle" | "racing" | "finished";`
    2. Update the `Driver` class to use this Union type instead of strings.

Task 4: Map Safety (Union Types)

The variable `currentMap` is a string. It accepts "Rainbow Road" or "Parking Lot" (which doesn't exist).

- **The Fix:**
    1. Create a Union Type: `type MapName = 'Rainbow Road' | 'Choco Mountain'`.
    2. Type the `currentMap` variable to use this type.
    3. Now, try assigning a wrong map name and see TS stop you.

Task 5: DOM Safety

`document.getElementById()` returns `HTMLElement | null`.

- **The Problem:** TypeScript warns "Object is possibly null" when you try to access `.innerText` or `.style`.
- **The Fix:** Add a null check or an assertion.
    - *Safe:* `if (element) { ... }`
    - *Assertive:* `const el = document.getElementById(...) as HTMLElement;`

Task 6: Abstract Classes (Inheritance)

Currently, `Driver` is just a normal class. However, in our game logic, we never want to create a generic "Driver"—we only want to create specific Pokemon.

- **The Task:** Convert `class Driver` to `abstract class Driver`.
- **The Check:** Try to write `new Driver(...)` in your code. TypeScript should now forbid it. This enforces the Inheritance pattern.

Task 7: The Item Box (Discriminated Unions)

This is the "Hard Mode" task. We want drivers to pick up items.

1. **Define the Types:** Create types for two items:
    - `Mushroom` (has a `speedBoost: number` property).
    - `Coin` (has a `points: number` property).
    - Both should have a `type` property (e.g., `'mushroom'` or `'coin'`).
2. **Create the Union:** Create a type `Item = Mushroom | Coin`.
3. **Update the Class:** Add an `inventory: Item | null` property to the `Driver` class.
4. **Implement Logic:**

- Add a method `useItem()`.
- Inside `useItem`, you **MUST** use a `switch` or `if` on `item.type` to safely access `speedBoost` or `points`. TypeScript should forbid accessing `speedBoost` on a Coin.
- Each player should start with a single item, and the items are picked by random (meaning that it is a 50% chance of receiving a Coin, and a 50% chance of receiving a Mushroom)
- When the race starts again each player should be assigned new items based on the same logic as described above

5. **Add the logic to the game:**
    - When a driver is racing, he has a 10% chance of using an item per frame
6. **Add visual elements:**
    - Items each driver was designated must be visual
    - When the item is used it should be enlarged and should then disappear gradually going from opacity 1 to 0 at a specific interval
    - You are required to modify `index.html` and add custom styles to `style.css` to add the visual elements

Task 8: The Contract (Interfaces)

To ensure our code is extensible, we want to enforce a contract.

1. Create an interface `IRacer`.
2. It should require: `name` (string), `position` (number), and a `move()` method.
3. Update the `Driver` class to implement this interface: `abstract class Driver implements IRacer`.