Building a Type-Safe Game in TypeScript

A step-by-step deconstruction of **Union Types, Type Guards**, and **State Management**.

Key Skills You Will Master

Construct Union Types: Use the pipe operator (`|`) to declare variables that can hold multiple specific data types.

Manage Mutable State: Differentiate between `const` and `let` to manage static configuration versus dynamic application state.

Implement Type Guards: Apply runtime checks with `typeof` to safely handle variables with multiple potential types.

Control Variable Scope: Utilize block scope to declare variables precisely where they are needed, keeping code clean and conflict-free.

Our Destination: The Complete Number Guessing Game

```
import * as readline from 'node:readline/promises';
import { stdin as input, stdout as output } from 'node:process';

const rl = readline.createInterface({ input, output });

// 1. CONFIGURATION
const secretNumber: number = Math.floor(Math.random() * 10) + 1;

// 2. STATE MANAGEMENT
let isGameRunning: boolean = true;
console.log('Welcome to the Number Guessing Game!');

while (isGameRunning) {
    const rawInput = await rl.question('Enter a number between 1-10 (or type "quit" to exit): ');

    // 3. UNION TYPE VARIABLE
    let playerGuess: number | string;

    if (rawInput.toLowerCase() === 'quit') {
        playerGuess = 'quit';
    } else {
        playerGuess = parseFloat(rawInput);
    }

    // 4. TYPE GUARDS & LOGIC
    if (typeof playerGuess === 'number') {
        if (isNaN(pleyerGuess)) {
            console.log("That doesn't look like a valid number.");
        } else {
        // BLOCK SCOPE variable
        let difference = Math.abs(secretNumber - playerGuess);
        if (difference === 0) {
            console.log('You guessed it! You win!');
            isGameRunning = false;
        } else {
            console.log(`You were off by ${difference}. Try again!`);
        }
    }
    } else if (playerGuess === 'quit') {
        console.log('Goodbye!');
        isGameRunning = false;
    }
}
rl.close();
```

We will deconstruct this application step-by-step, explaining the key TypeScript concepts that make it robust and type-safe.

Part 1: Configuration vs. State

Every application has two kinds of data: configuration values that are **set once**, and **state values** that change **over time**. TypeScript helps us enforce this distinction.

```
import { Game, base } from 'src/nonset';

// 1. CONFIGURATION
// Generate a random integer between 1 and 10
const secretNumber: number = Math.floor(Math.random() * 10) + 1;

// State management adicting state
const secretNumber: number = secretNumber[[]];

game loop = () {
  if (game.ilen(game.state == 10) {
    secretNumber(secretNumber) + game.apoloalop(2);
  }
}
```

`const` **for Configuration:** `secretNumber` is a static value. It's set once and never changes during the game's execution. We use `const` to enforce this immutability.

Managing Mutable State with `let`

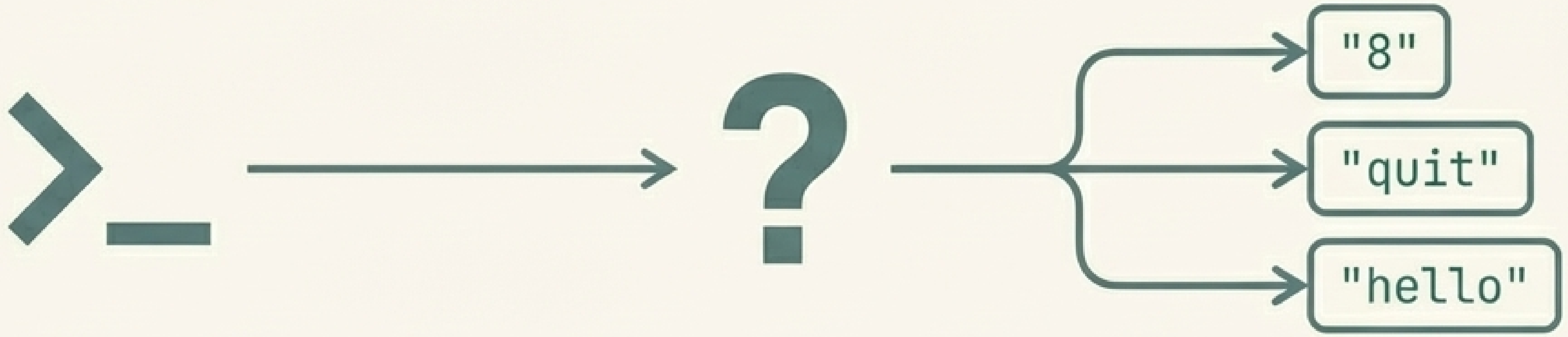
To control the game's flow, we need a flag that can change. This is called "mutable state", and we use `let` to declare variables whose values are expected to be reassigned.

```
// 2. STATE MANAGEMENT
// This is "Mutable State" - it changes from true to false later.
let isGameRunning: boolean = true;

// 2. Minuttes for isGameRunning
const {game: "#5A7D7C" = true";

// 3. Mutables for game only
const comemander, nunt = false;
```

`let` for State: `isGameRunning` is our game's 'on/off' switch. It starts as `true` and will be set to `false` when the game ends. `let` allows this reassignment.

The Challenge: Unpredictable User Input

The core of our interactive loop involves waiting for user input. But what is the *type* of that that input? It could be a number, the word 'quit', or something else entirely. This is a classic typing challenge.



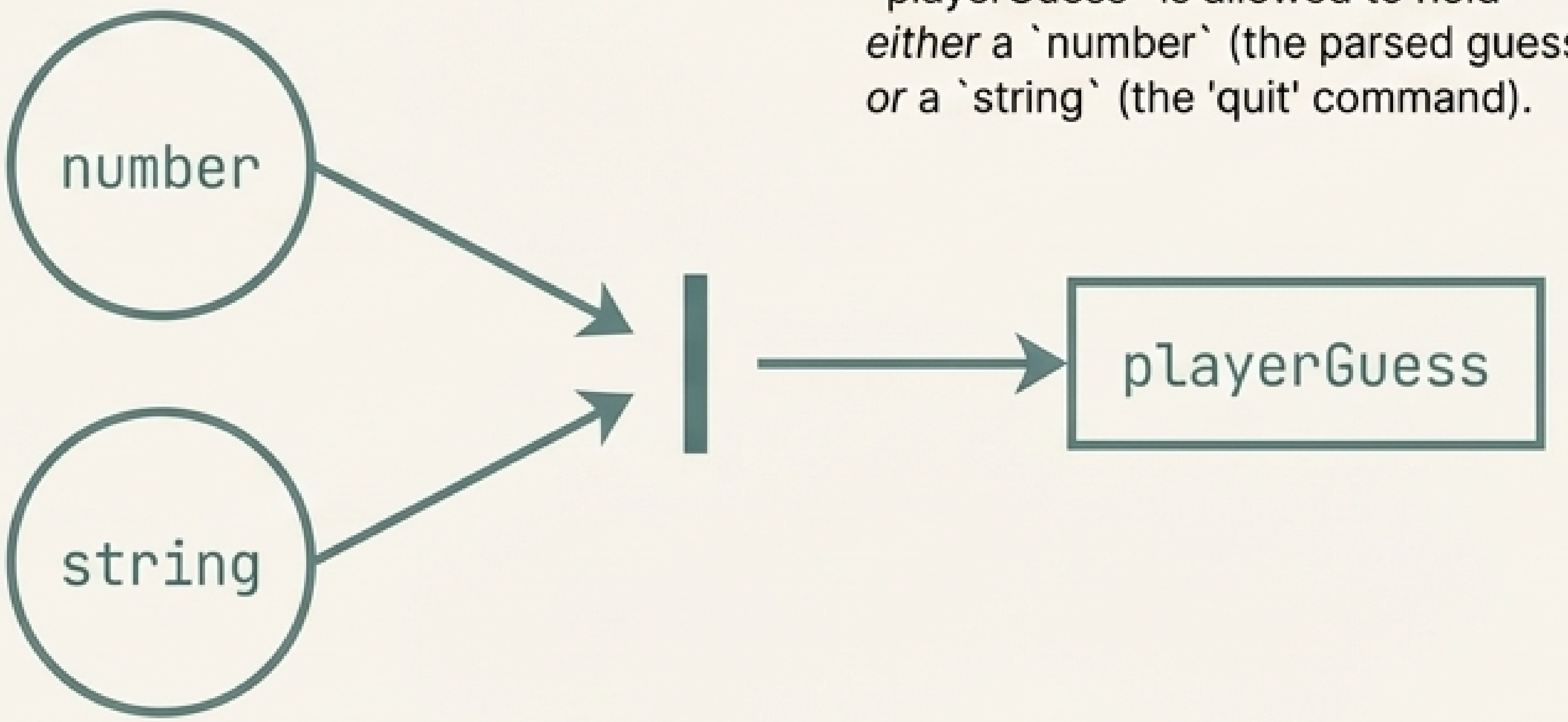

```
while (isGameRunning) {
  // Ask for input (awaits the user's response each time)
  const rawInput = await rl.question('Enter a number between 1-10 (or type "quit" to exit): ');

  // How do we handle rawInput safely?
}
```

The Solution: Union Types for Flexibility

TypeScript's Union Types let us declare that a variable can be one of several specific types. We use the pipe (`|`) symbol to combine them. This allows for flexibility while maintaining type safety.

```
// 3. UNION TYPE VARIABLE
// Re-declared inside the loop for each new guess
let playerGuess: number | string;
```

``number | string``: This tells the TypeScript compiler that `playerGuess` is allowed to hold *either* a `number` (the parsed guess) *or* a `string` (the 'quit' command).



Parsing Input into our Union Type

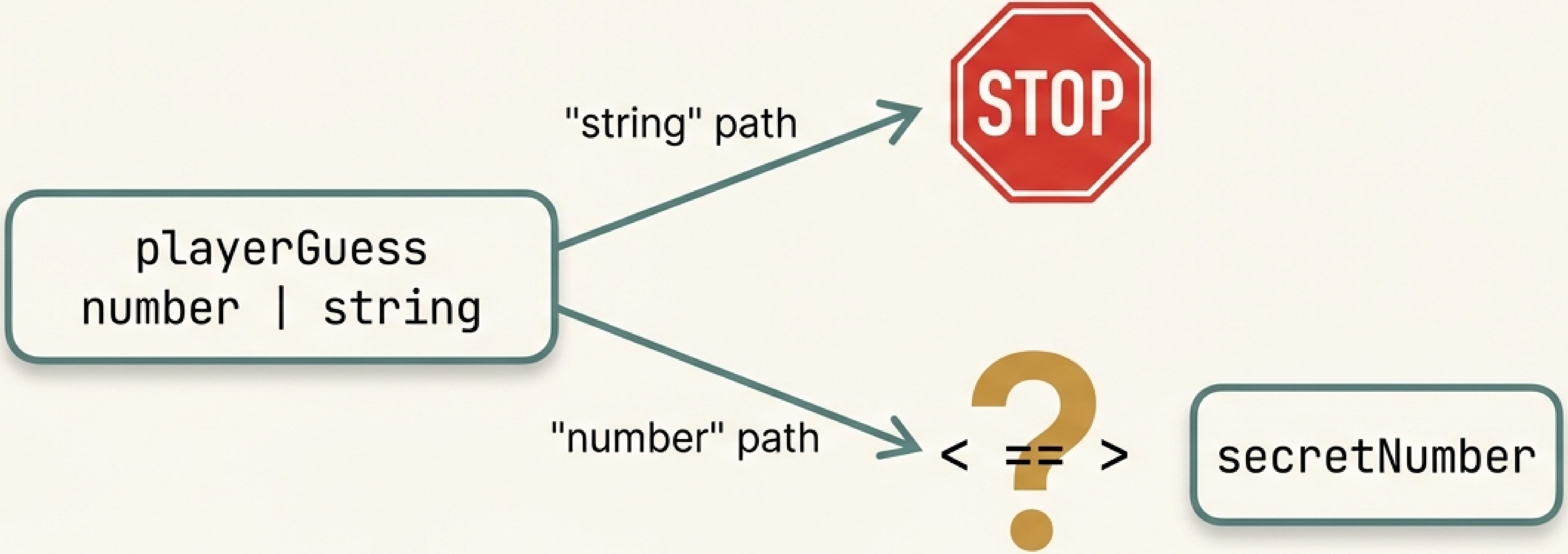
Next, we implement logic to parse the `rawInput` and assign the correct value and type to `playerGuess`.

```
// Faded surrounding code...
let playerGuess: number | string;

// PARSING LOGIC
if (rawInput.toLowerCase() === 'quit') {
   playerGuess = 'quit';
} else {
   playerGuess = parseFloat(rawInput);
}

// Faded surrounding code...
```

After this block, `playerGuess` now holds either the string `quit` or a floating-point number. Our Union Type makes this possible.

The Next Challenge: How to Compare a `number | string`?

Our code is now more flexible, but it creates a new problem. How can we safely compare `playerGuess` to `secretNumber`? TypeScript won't let us perform math operations on a variable that *might* be a string.



The Solution: Type Guards for Runtime Checks

Type Guards are conditional checks that TypeScript understands. Inside a block guarded by a `typeof` check, TypeScript will “narrow” the variable's type, allowing you to safely use type-specific methods and operations.

```
playerGuess: number | string

// 4. TYPE GUARDS & LOGIC
if (typeof playerGuess === 'number') {    playerGuess: number
    // Inside this block, TypeScript KNOWS playerGuess is a number.

} else if (playerGuess === 'quit') {      playerGuess: string
    // Inside this block, TypeScript KNOWS playerGuess is the string 'quit'.
}
```

NotebookLM

Inside the ‘Number’ Path

With our Type Guard in place, we can now confidently treat `playerGuess` as a number. But we should still validate it—what if the user typed 'hello'?

```
if (typeof playerGuess === 'number') {

    if (isNaN(playerGuess)) {
        console.log("That doesn't look like a valid number.");
    } else {
        // ... game logic here ...
    }
}
```

The `isNaN()` check handles cases where `parseFloat` fails on non-numeric input (e.g., `parseFloat('abc')` results in `NaN`).

Mastering Scope with `let`

Notice the `difference` variable is declared inside the `else` block. This is 'Block Scope'. The variable only exists where it is needed, preventing clutter and potential conflicts in the wider program.

```
} else {
    // BLOCK SCOPE variable
    let difference = Math.abs(secretNumber - playerGuess);

    if (difference === 0) {
        // ... win logic ...
    } else {
        // ... try again logic ...
    }
}
```

Declaring `difference` here is efficient and clean. It cannot be accidentally accessed or modified from outside this specific logic block.

Updating State: The Win and Lose Conditions

The final step in our number logic is to compare the guess and update the application's state accordingly. Setting `isGameRunning` to `false` is what ends the game.

```
if (difference === 0) {
  console.log('You guessed it! You win!');
  // We update state to stop the loop because they won
  isGameRunning = false;
} else {
  console.log(`You were off by ${difference}. Try again!`);
  // Loop continues automatically...
}
}
```

This line is crucial: it breaks the game loop, signaling the end of the session. The game state is updated to reflect victory.

Completing the Logic: The ‘String’ Path

Our Type Guard also allows us to handle the non-numeric cases cleanly. The `else if` block handles the “quit” command, once again updating the state to terminate the loop.

```
} else if (playerGuess === 'quit') {
  console.log('Goodbye!');
  // We update state to stop the loop because they quit
  isGameRunning = false;
}
```

Because of the `typeof` check before this, TypeScript is smart enough to know that if `playerGuess` isn't a number, it must be a string, allowing this direct string comparison.

Tying It All Together

1. **Union Type:** Provided the flexibility to handle different kinds of input.

3. **State Management:** Used `const` for static config and `let` for the mutable state that controlled the game loop.

```typescript
const secretNumber: number = 42;
let isGameRunning: boolean = true;

// Game Loop would start here...

let playerGuess: number | string;

// ...user input happens here...

if (typeof playerGuess === 'number') {
  // Numeric guess logic
  if (playerGuess < secretNumber) {
    console.log('Too low! Try again.');
  } else if (playerGuess > secretNumber) {
    console.log('Too high! Try again.');
  } else {
    // Check win condition
    if (difference === 0) {
      console.log('You guessed it! You win!');
      // We update state to stop the loop because they won
      isGameRunning = false;
    } else {
      console.log(`You were off by ${difference}. Try again!`);
      // Loop continues automatically...
    }
  }
} else if (playerGuess === 'quit') {
  console.log('Goodbye!');
  // We update state to stop the loop because they quit
  isGameRunning = false;
}
```

2. **Type Guard:** Enabled safe, runtime type-checking to run specific logic.

4. **Block Scope:** Kept our code clean by defining variables only within the block where they were needed.