TypeScript: Asynchronous Code & Modules

Building a Modern Interactive Command-Line App

NotebookLM

From Static to Interactive

The Challenge

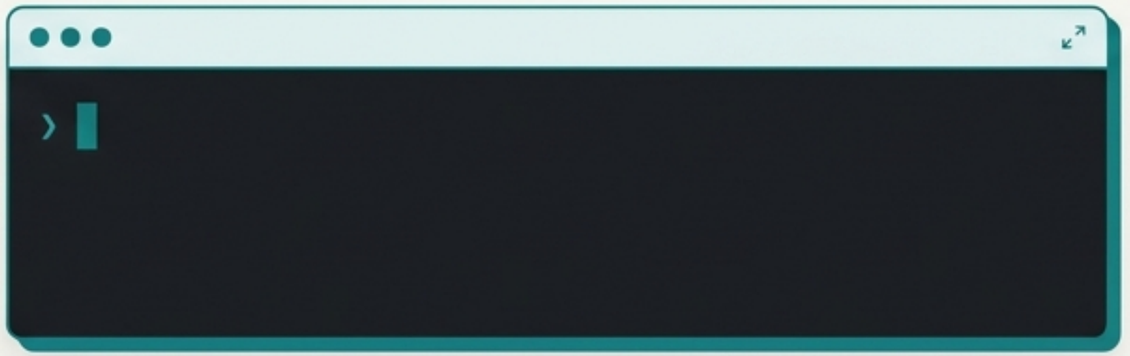
How do we write code that can **pause and wait** for a user? How do we keep that code **clean, organized, and readable** as our application grows?

```
// Old Way
function calculateTip(bill: number, percentage: number): number {
  return bill * (percentage / 100);
}

const billAmount = 100; // Hardcoded
const tipPercentage = 20; // Hardcoded

const tip = calculateTip(billAmount, tipPercentage);
console.log(`Tip: ${tip}`);
```

Our Goal

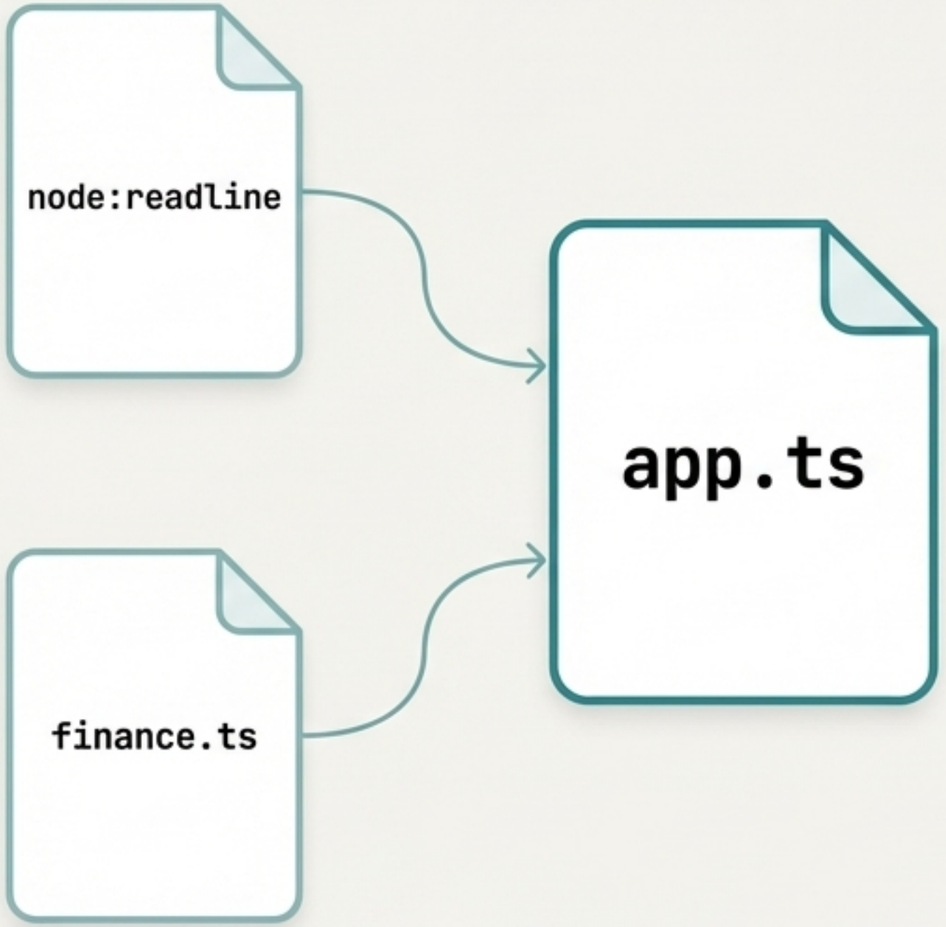
Master the core TypeScript concepts to build a polished, interactive CLI application by understanding:

1. **Modularity**: Organizing our code into logical, reusable files.
2. **Asynchronous Flow**: Handling unpredictable events like user input.
3. **Modern Syntax**: Writing cleaner, more readable code with top-level `await`.



NotebookLM

Part 1: Mastering Modularity



The foundational principle of scalable software is simple: don’t put everything in one file.

Modules allow us to break our application into logical, reusable pieces using ES Module syntax (`import`/`export`). This makes our code easier to maintain, debug, and share across projects.

We will connect two types of modules:

- **Built-in Node.js APIs**: Powerful tools provided by the Node.js platform itself.
- **Our Own Custom Code**: The specific business logic unique to our application.

NotebookLM

Importing Built-in Node.js Modules

To build an interactive CLI, we first need the tools to read user input. We use the `import` keyword to pull in the `readline` library, which is essential for this task.

app.ts

```
// Import the entire 'readline/promises' module as a single object.
import * as readline from 'node:readline/promises';

// Import specific objects from the 'process' module and rename them.
import { stdin as input, stdout as output } from 'node:process';
```

The `node:` prefix is the modern, recommended way to specify a built-in Node.js module.

The `as` keyword lets us rename imports for better readability in our code.

NotebookLM

Separating Logic from Execution with Custom Modules

We isolate our core calculation logic in its own file (`finance.ts`) and use `export` to make the functions available. Our main application file (`app.ts`) can then `import` and use them.

`finance.ts` (Our Logic)

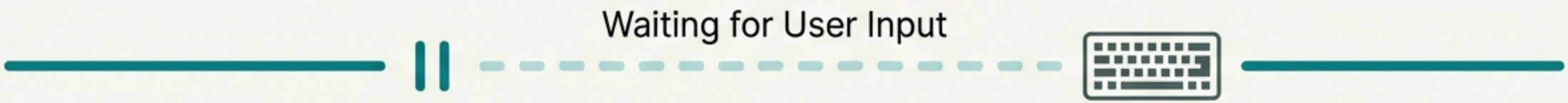
```
// We use 'export' to make these functions
// available to other files.

export const calculateTip = (bill: number, rate:
number): number => {
  return bill * (rate / 100);
};

export const calculateTotal = (bill: number, tip:
number): number => {
  return bill + tip;
};
```

`app.ts` (Our Application)

```
// Notice the './' which signifies a local file in
// the same directory.
import { calculateTip, calculateTotal } from
'./finance';

// Now we can use these functions as if they were
// defined here!
const tipAmount = calculateTip(bill, rate);
```

NotebookLM

Part 2: Taming Asynchronicity



The Problem: Code Doesn't Wait

By default, code executes instantly, line by line. But user input can happen at any time—or not at all. We need a way to **pause execution** and wait for the user to act. This is an **asynchronous operation**.

The Solution: `async` and `await`

TypeScript provides the `async`/`await` syntax to handle these situations gracefully. It makes complex, asynchronous code look and behave like simple, linear, easy-to-read code.

NotebookLM

The `await` Keyword: Pausing for Input

The `await` keyword is our primary tool for managing asynchronicity. It tells the program: “Pause execution on this line until the operation on the right is complete, then assign its result and continue.”

```
const rl = readline.createInterface({ input, output });
console.log("--- 🍽️ Modular Tip Calculator 🍽️ ---");

// 1. Get Input (Asynchronously)
// The program STOPS HERE until the user presses 'Enter'.
const strBill = await rl.question("Total bill amount: $");

// This line will only run AFTER the user provides the bill amount.
const strRate = await rl.question("Tip percentage (e.g. 20): ");
```

Pauses execution here

`Promise` Explained

`rl.question` doesn't immediately return the user's text. It returns a `Promise`—a placeholder for a value that will be available later. `await` effectively pauses the function and waits for that promise to be fulfilled with the user's input.

NotebookLM

The Elegance of Top-Level Await

Historically, `await` could only be used inside a function marked with `async`. This led to extra wrapper functions. Modern TypeScript, with a simple configuration change, allows **Top-Level Await**, letting us write **cleaner code** at the root of our file.

The Old Way (Requires a Wrapper)

```
async function main() {
  const rl = readline.createInterface(...);
  const strBill = await rl.question(...);
  const strRate = await rl.question(...);
  // ... rest of the logic
  rl.close();
}

main(); // Don't forget to call the function!
```

The Modern Way (Clean & Linear)

```
const rl = readline.createInterface(...);

// No wrapper function needed!
const strBill = await rl.question(...);
const strRate = await rl.question(...);
// ... rest of the logic
rl.close();
```

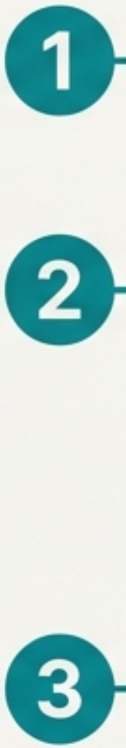
Configuration Note: Enable this feature in your `tsconfig.json` by setting "target": "ESNext" and "module": "ESNext".

NotebookLM

Part 3: The Complete Application

Putting It All Together

Here is the final `app.ts` file. Notice how the concepts we've discussed—modules and top-level await—come together to create a readable, logical flow from top to bottom.




```
// 1. MODULES: Importing built-in and custom code
import * as readline from 'node:readline/promises';
import { stdin as input, stdout as output } from 'node:process';
import { calculateTip, calculateTotal } from './finance';

const rl = readline.createInterface({ input, output });
console.log("--- 🍽️ Modular Tip Calculator 🍽️ ---");

// 2. ASYNC: Using top-level await to pause for user input
const strBill = await rl.question("Total bill amount: $");
const strRate = await rl.question("Tip percentage (e.g. 20): ");

// Parse from string to number
const bill = parseFloat(strBill);
const rate = parseFloat(strRate);

if (isNaN(bill) || isNaN(rate)) {
  console.log("Invalid numbers entered.");
} else {
  // 3. LOGIC: Using our imported functions
  const tipAmount = calculateTip(bill, rate);
  const totalAmount = calculateTotal(bill, tipAmount);

  console.log(`\nTip: $${tipAmount.toFixed(2)}`);
  console.log(`Total: $${totalAmount.toF1xed(2)}`);
}

rl.close(); // Cleanly close the input stream
```

NotebookLM

The Result: An Interactive Experience

When we compile and run our application, the asynchronous flow and modular logic combine to create a seamless, interactive command-line experience.

```
$ ts-node app.ts
--- 🍽️ Modular Tip Calculator 🍽️ ---
Total bill amount: $120.50
Tip percentage (e.g. 20): 18

Tip: $21.69
Total: $142.19
```

NotebookLM

Learning Objectives: Revisited

We have successfully applied modern TypeScript features to build a complete interactive application.

Import Node Modules

We used `import ... from 'node:...'` to integrate Node.js's `readline/promises` API.

Manage Asynchronous Flow

We explained why user input is async and used `await` to pause execution until data was received.

Implement Top-Level Await

We leveraged the `ESNext` target to write clean, linear async code without wrapper functions.

Construct CLI Interfaces

We built a complete, interactive command-line app, from prompting the user to cleanly closing the input stream.

NotebookLM

The Building Blocks of Modern Applications

The principles of modularity and asynchronous control are not just for simple CLIs. They are the foundation for building everything from complex backend APIs to dynamic front-end web applications. Mastering them is essential for any modern developer.

For further exploration, view the complete source code:
github.com/example/ts-cli-tutorial

NotebookLM