Lab: Debugging Dart Applications Using Cloud IDE

Estimated time: **30** minutes

Skills Network

Welcome to this hands-on lab, where you will learn how to debug Dart applications using Visual Studio Code. This lab will guide you through setting up your environment, creating a more complex Dart application with intentional errors, and using the debugging tools in VS Code.

About Cloud IDE

Running the lab

This lab is designed to be completed using a Cloud IDE environment. You will be writing and executing Dart code directly within the IDE.

About Skills Network Cloud IDE

Skills Network Cloud IDE (based on Theia and Docker) provides an environment for hands-on labs for course and project-related labs. It is an open-source integrated development environment (IDE).

Important note about this lab environment

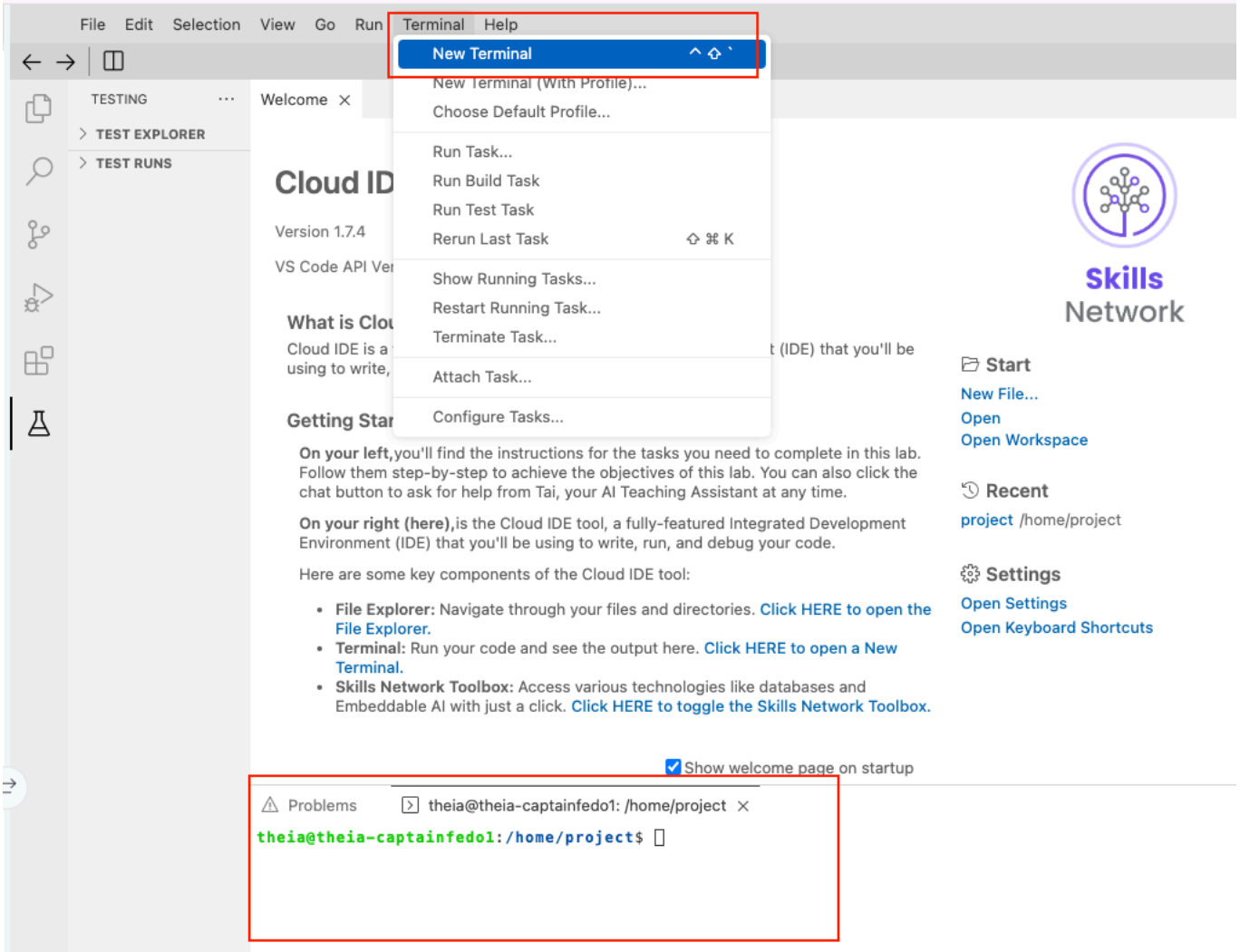
Please be aware that sessions for this lab environment do not persist. Every time you connect to this lab, a new environment is created for you. Any data you may have saved in the earlier session would get lost. Plan to complete these labs in a single session to avoid losing your data.

Key terms

- **Breakpoint**: A point in the program where the execution will stop, allowing you to inspect the state of the application. Breakpoints are essential for investigating the behavior of code at specific stages or upon certain conditions.
- **Exception**: An error that occurs during the execution of a program, disrupting the normal flow of instructions. Exceptions can be caught and handled to prevent them from causing the program to terminate unexpectedly.
- **Debugger**: A tool that is used to test and debug programs by allowing the developer to step through code, inspect variables, and control the execution flow. Debuggers are integral to diagnosing and fixing issues in code.
- **Stack trace**: A report that provides the path of execution through the program at the point where an exception occurs. It shows the sequence of function calls that led to the error, which can help pinpoint the location of a problem within the code.
- **Variable inspection**: The process of checking the values of variables at certain points in the program during debugging. This is crucial for understanding the state of the application and determining the cause of issues.
- **Step over/into/out**: Debugging commands that control the execution of the program:
  - **Step over**: Executes the next line of code but does not dive into any functions called by it.
  - **Step into**: Executes the next line of code and if it is a function call, the debugger enters the called function, allowing you to debug the function's code.
  - **Step out**: Continues executing the remaining lines of the current function and returns to the calling function.
- **Watch expression**: A feature in many debuggers that allows you to specify expressions based on variables in the code, which the debugger will evaluate and display the results of as the program runs. This helps monitor the changes in data throughout the execution.
- **Conditional breakpoint**: A breakpoint that triggers only when a specified condition is true. This allows developers to focus on debugging specific scenarios and can significantly speed up the debugging process.

Step 1: Create a new Dart project

1. Open your terminal or command prompt.



2. Create a new Dart project by running:

```
dart create dart_debug
```

This should give you an ouput as the following:

```
dart create dart_debug
 Creating dart_debug using template console...
   .gitignore
   analysis_options.yaml
   CHANGELOG.md
   pubspec.yaml
   README.md
   bin/dart_debug.dart
   lib/dart_debug.dart
   test/dart_debug_test.dart
 Running pub get...                      1.4s
   Resolving dependencies...
   Downloading packages...
   Changed 50 dependencies!
   3 packages have newer versions incompatible with dependency constraints.
   Try `dart pub outdated` for more information.
 Created project dart_debug in dart_debug! In order to get started, run the following commands:
   cd dart_debug
   dart run
```

3. This command has created a new directory for your application and quite a few files. Navigate into your newly created project directory:

```
cd dart_debug
```

You can use ls to list the files created for you:

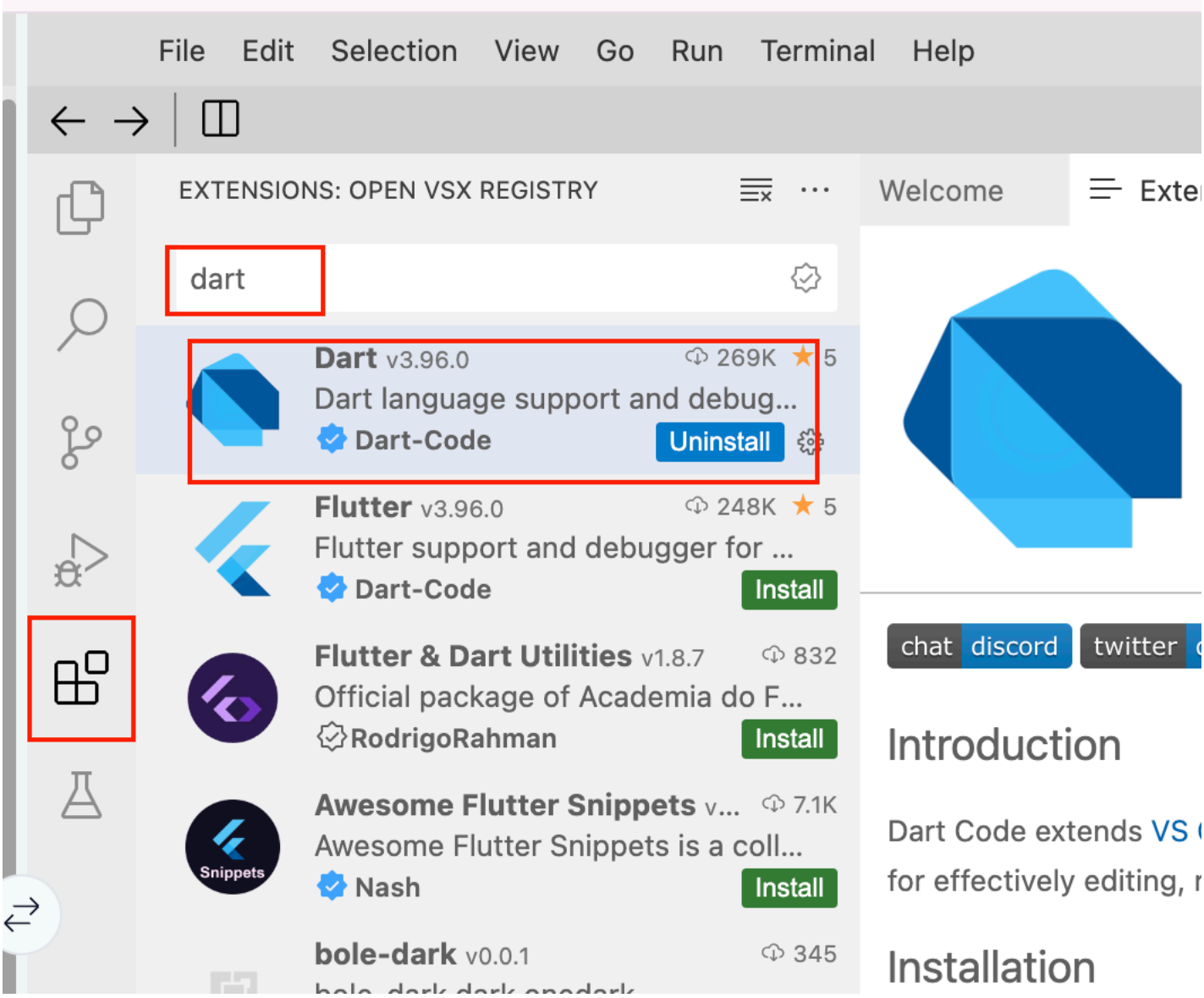
- `pubspec.yaml`: This file serves as the project's manifest. It includes metadata about the project such as name, version, and dependencies. It is used by the Dart package manager to manage the Dart packages that your project depends on.
- `lib/`: This directory contains the public Dart code of the project. For a typical application or package, most Dart files, including the entry point (usually a file like main.dart), are placed here.
- `bin/`: This directory holds executable Dart files. These files are entry points for command-line or server applications, as opposed to lib/, which is intended for library code.
- `analysis_options.yaml`: This file is used to configure Dart's static analysis tool. You can customize rules to follow specific guidelines or ignore certain rules across your project.
- `README.md`: A Markdown file that usually contains an overview of the project, how to set it up, and how to use it. This is often the first file that users and contributors will look at when encountering your repository.
- `CHANGELOG.md`: This Markdown file logs all the changes made over time to the project. It typically includes updates, fixes, and other important notes that are relevant for users to know what has changed from one version to the next.
  - `test/`: Contains Dart files for testing the application or library. Dart uses a powerful package named test to write and run unit tests.
- `pubspec.lock`: This file is automatically generated by the Dart package manager and includes a list of all packages your application depends on, along with the specific versions that were installed. This ensures consistent environments and version control for all project dependencies.

Step 2: Verify the debugging environment

Ensure the Dart extension is installed in Cloud IDE by following these steps:

1. Go to the Extensions view by clicking on the square icon on the sidebar or pressing Ctrl+Shift+X.

2. Search for “Dart” and install the extension if not already installed.



Step 3: Create a complex Dart app

Understanding the Game logic

Before you start coding, let's walk through what the Dart application does. This program simulates a simple text-based adventure game where the player can perform different actions represented by an enumeration: move, take, open, and look. The player starts at a specific location, “start,” and can move to other predefined locations like “forest” and “cabin” by performing actions. Each action affects the game state:

- **Move**: Changes the player's current location to a new one. For example, moving from “start” to “forest”.
- **Take**: Attempts to pick up an item. In the forest, the player can find a hidden treasure if they are in the correct location.
- **Open**: Opens something in the current location. For instance, if the player is at the “cabin”, they can open its door.
- **Look**: Provides a description of the current location, helping the player understand where they are and hinting at possible actions.

The game is designed to loop through these actions based on player input, updating and responding to the player's actions. However, a typo in the code misdirects one of the actions, leading to incorrect game behavior. Your task will be to run the game, identify why an expected action doesn't produce the right outcome, and correct the error using debugging tools in VS Code.

Create and launch the application

1. Create a new Dart file named `complex_debug.dart` in your `lib` folder.

```
touch /home/project/dart_debug/lib/complex_debug.dart
```

2. Insert the following code, which simulates a simple text-based game engine with an intentional logic error for debugging:

```
// Define an enum for different game actions
enum Action { move, take, open, look }
// Class to simulate game logic
class Game {
  // Store player's current location
  String location = 'start';
  // Process actions taken by the player
  void handleAction(Action action) {
    switch (action) {
      case Action.move:
        location = 'forrest';  // Intentional typo in location name.
        break;
      case Action.take:
        if (location == 'forest') {
          print('You found a hidden treasure!');
        } else {
          print('There is nothing to take here.');
        }
        break;
      case Action.open:
        if (location == 'cabin') {
          print('You opened the cabin door.');
        } else {
          print('There is no cabin here to open.');
        }
        break;
      case Action.look:
        print('You are at $location');
        break;
      default:
        print('Unknown action.');
    }
  }
  // Start the game loop
  void start() {
    print('Game started. You are at the $location.');
    handleAction(Action.move);
    handleAction(Action.look);
    handleAction(Action.take);
  }
}
void main() {
  var game = Game();
  game.start();
}
```

Step 4: Run and observe the program

1. Open the terminal if you closed it earlier.

2. Run the program by typing

```
dart run lib/complex_debug.dart
```

3. Observe the output. You will see outputs indicating your actions and location, such as “You are at forrest.” However, the expected output should say “You are at forest.” This discrepancy is due to a typo in the `handleAction` method under the `move` case where “forest” is misspelled as “forrest”. This error prevents the `take` action from finding the hidden treasure, as the condition checks for “forest”.

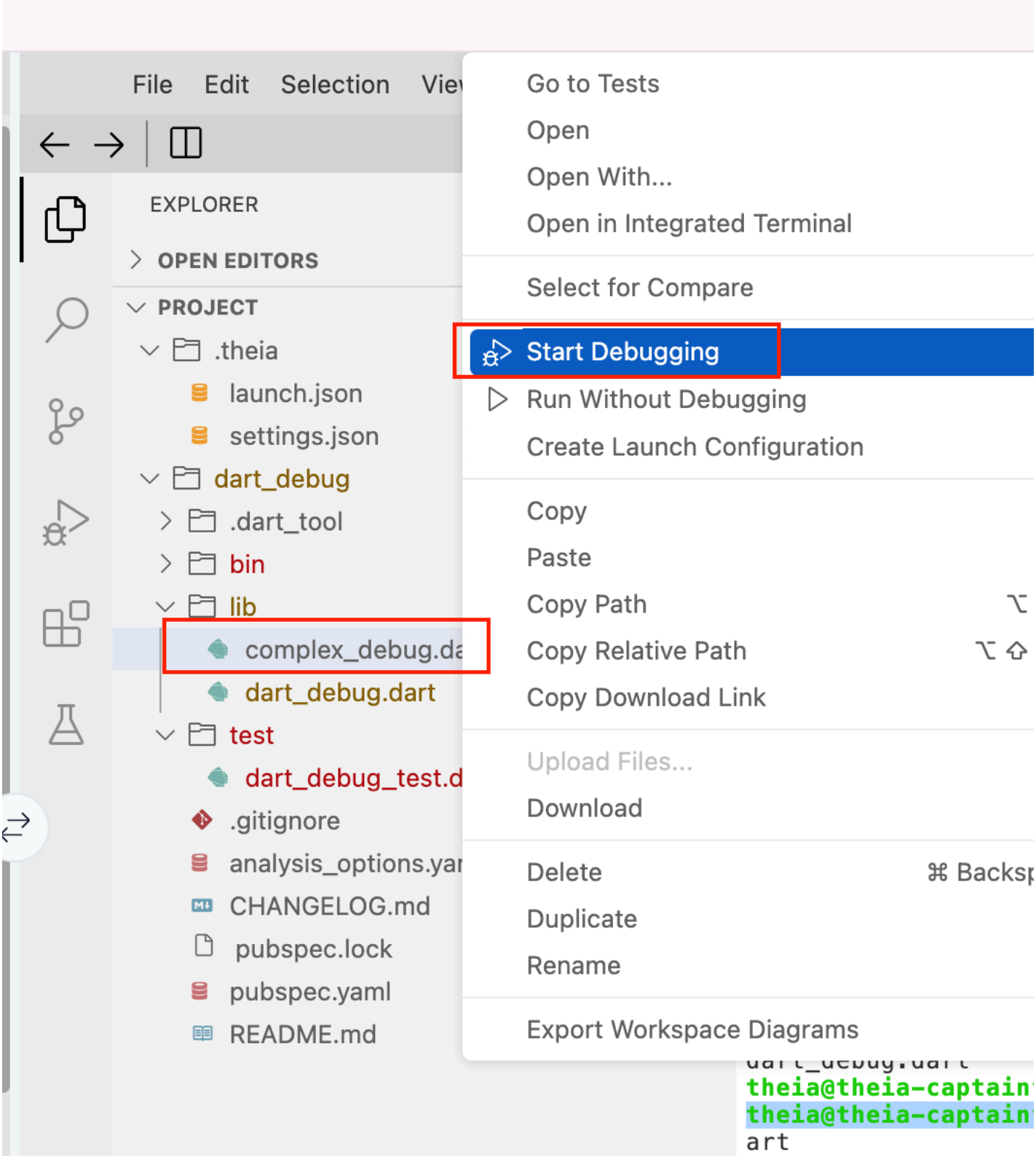
   You should see an output similar to the following:

```
Game started. You are at the start.
You are at forrest
There is nothing to take here.
```

This does not seem right. The user is supposed to find a treasure when they are in the forest! Let's see if we can debug this.

Step 5: Launch the debugger

1. You can start the debugger by right clicking on the file and selecting `Start Debugging` or using the `Run -> Start Debugging` menu.



This will open the debug view.

DEBUG

dart_debug

THREADS

CALL STACK

VARIABLES

WATCH

BREAKPOINTS

- [ ] All Exceptions
- [ ] Uncaught Exceptions

complex_debug.dart

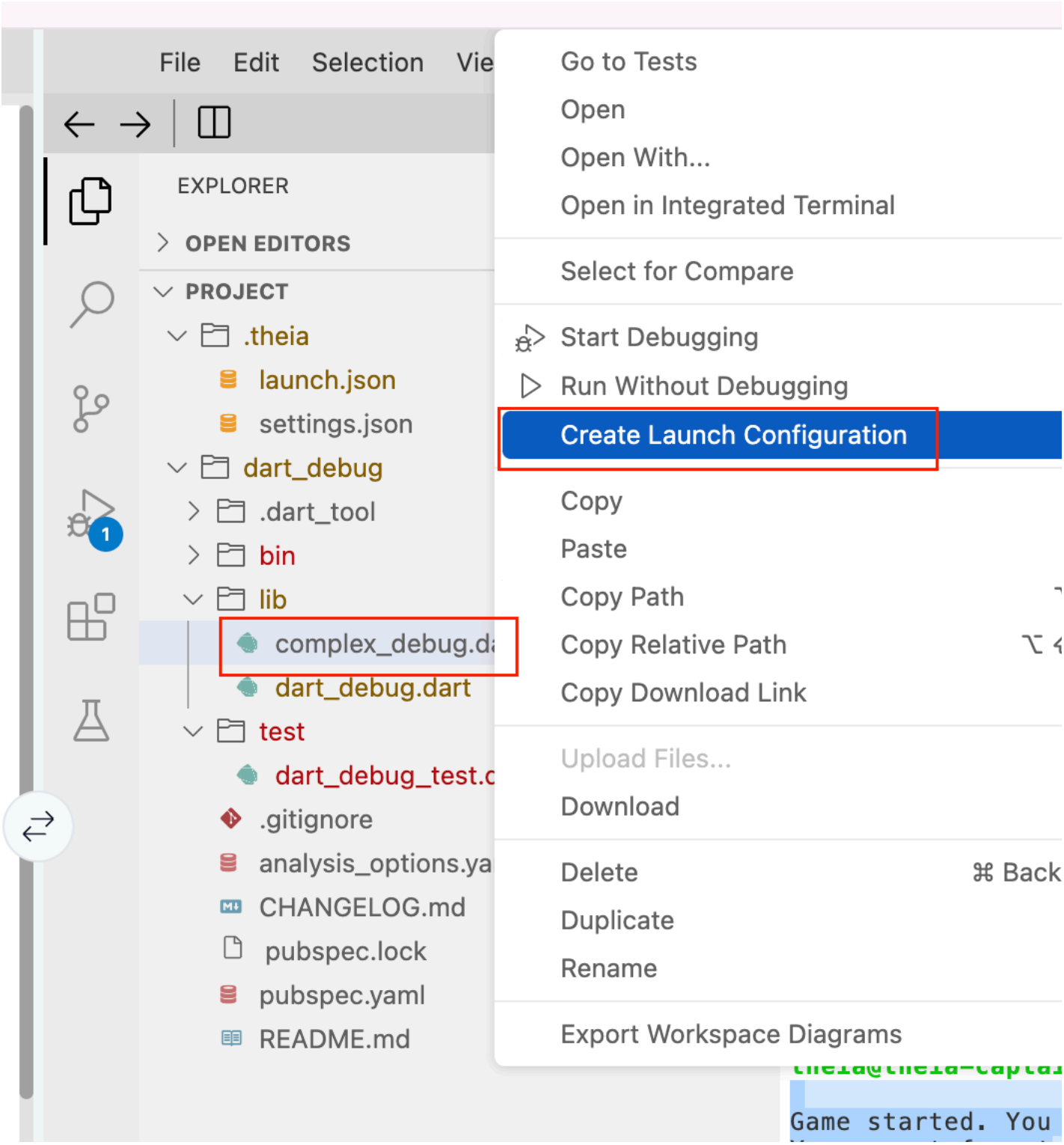
dart_debug > lib >

```
1   // Define an
2   enum Action {
3
4   // Class to s
5   class Game {
6     // Store pl
7     String loca
8
9     // Process
10    void handle
11      switch (a
12        case Ac
13          locat
14          break
15        case Ac
16          if (l
17            pri
18          } els
```

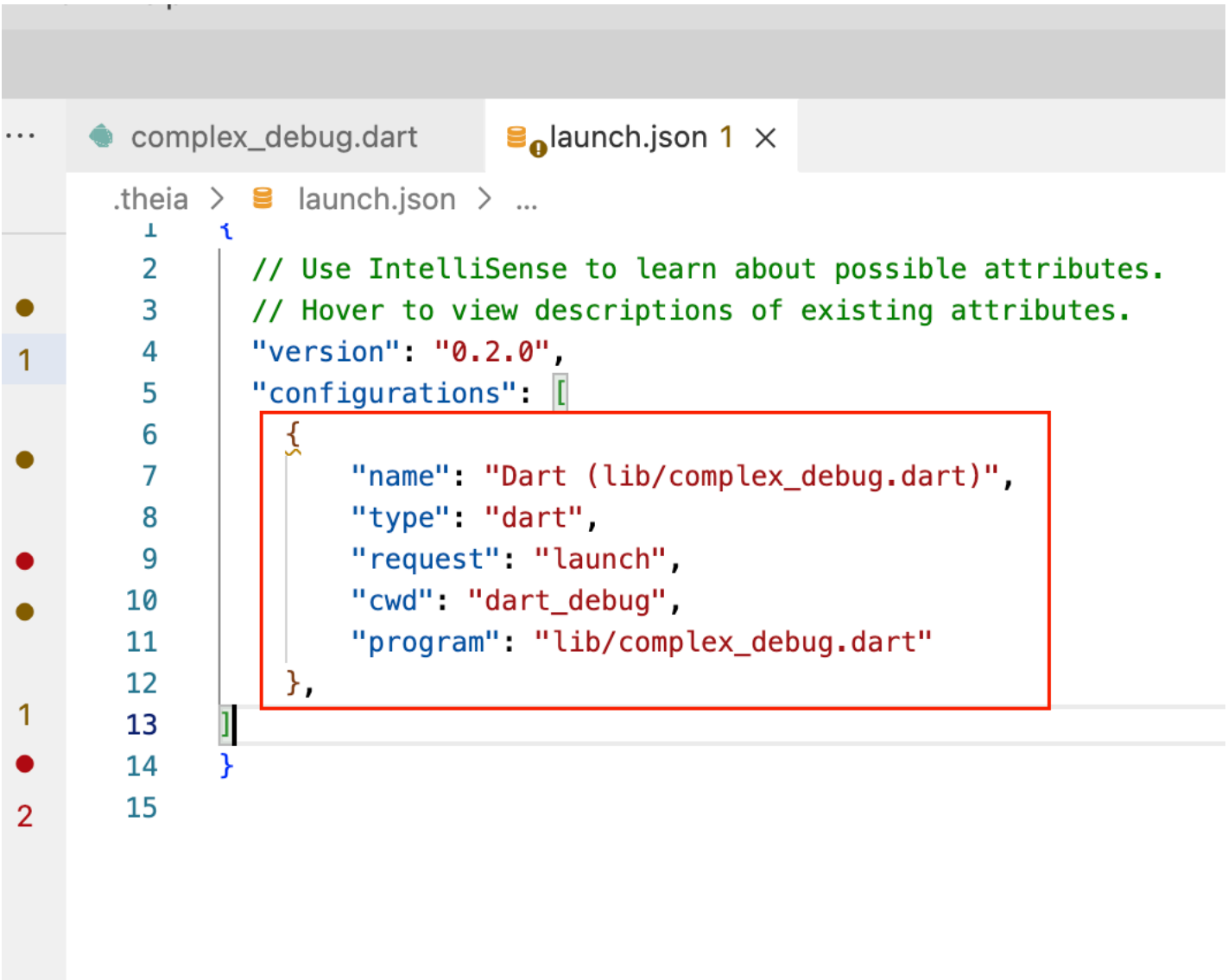
Debug Console

```
Connecting to VM Se
Connected to the VM
Game started. You a
You are at forrest
There is nothing to
Exited.
```

2. We don't want to right click every time. So instead, we will create a launch configuration to debug the `complex_debug.dart` file. Right click on the file and select `Create Launch Configuration`.

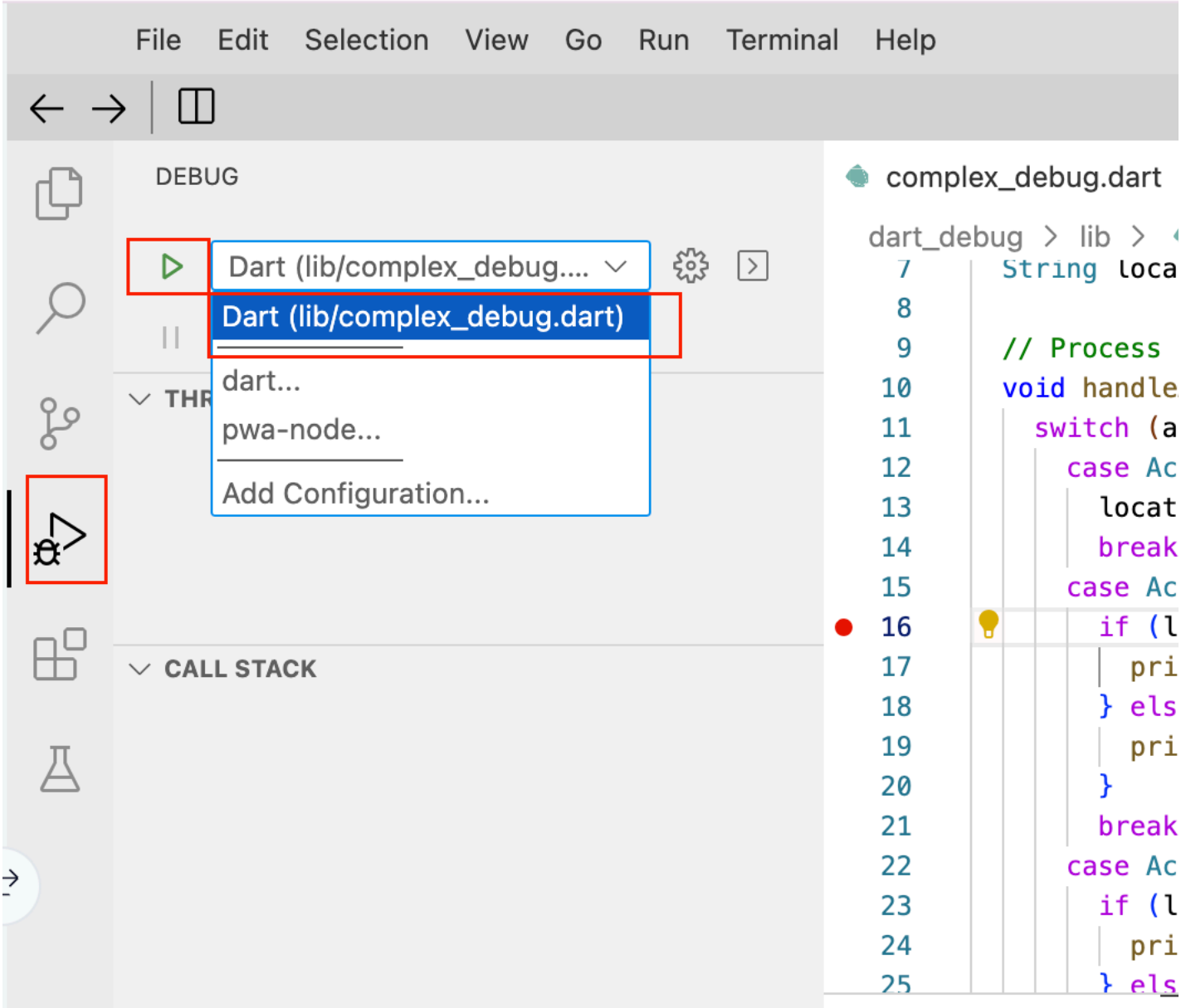


This will create a new configuration in the `launch.json` file.



Your output might look a little different as the current VS Code and Dart extension versions might be different since when the lab was written.

3. You can simply select this configuration to debug the file from now on.



Step 6: Using breakpoints

1. Set a breakpoint at the line inside the `handleAction` method where the location comparison takes place (`if (location == 'forest')`). There are a number of ways to set a breakpoint:
   - Click in the far left margin (gutter) next to a line of code as shown in the screenshot below.
   - Alternatively, you can right-click the gutter and select `Add Breakpoint`.
   - Finally, you can also select the line and select `Debug > Toggle Breakpoint` from the main menu.

complex_debug.dart

dart_debug > lib > complex_debug.dart

```dart
void handleAction(Action action) {
  switch (action) {
    case Action.move:
      location = 'forrest';  // Intentional typo in location n
      break;
    case Action.take:
      if (location == 'forest') {
        print('You found a hidden treasure!');
      } else {
        print('There is nothing to take here.');
      }
      break;
    case Action.open:
      if (location == 'cabin') {
        print('You opened the cabin door.');
      } else {
        print('There is no cabin here to open.');
      }
      break;
    case Action.look:
      print('You are at $location');
      break;
    default:
      print('Unknown action.');
  }
}

// Start the game loop
void start() {
```

2. Run your app. The execution will pause when it hits the breakpoint, allowing you to inspect why the ‘take’ action does not trigger the expected output despite the ‘move’ action supposedly moving the player to the ‘forest’.

complex_debug.dart

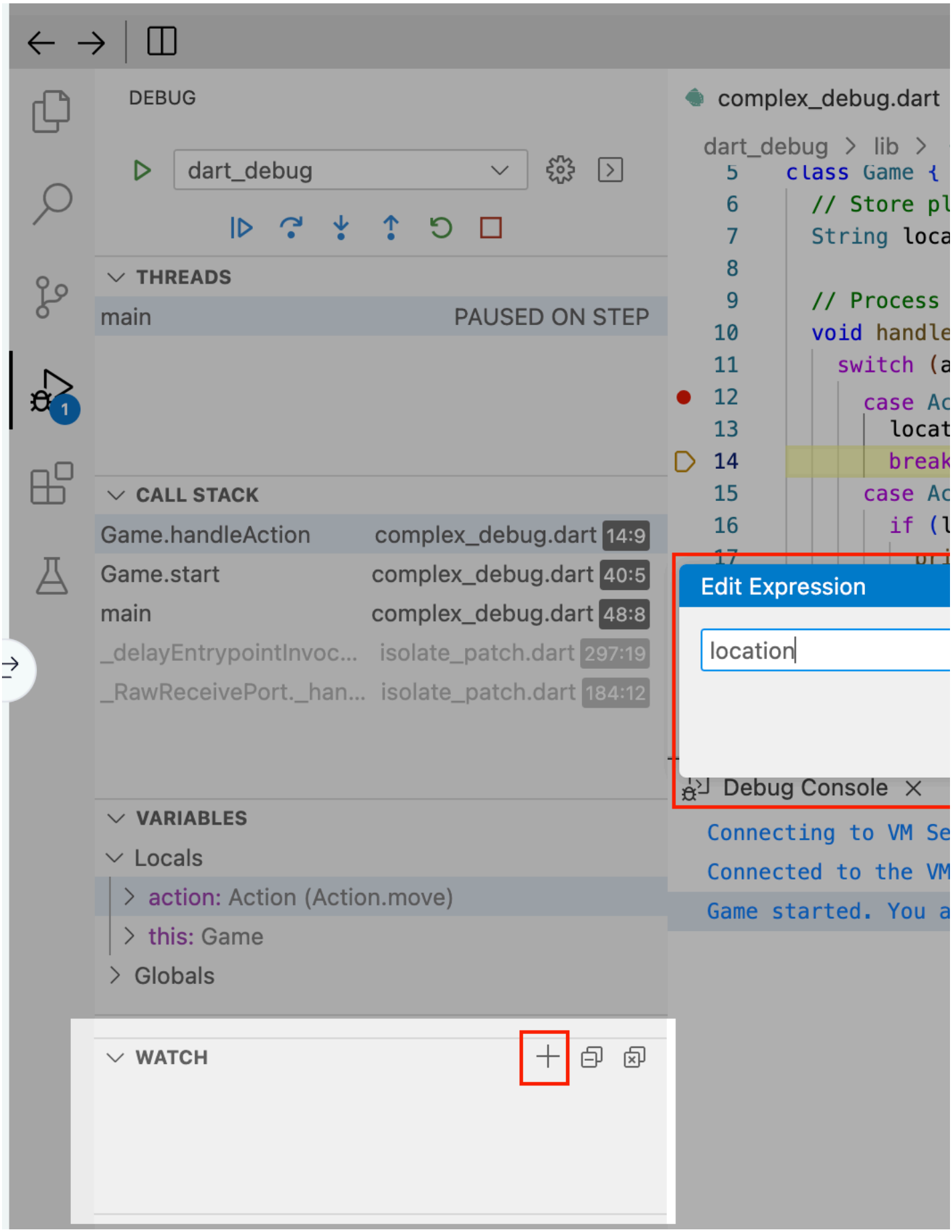
dart_debug > lib > complex_debug.dart > Game > handleActio

```
7   String location = 'start';
8
9   // Process actions taken by the player
10  void handleAction(Action action) {
11    switch (action) {
12      case Action.move:
13        location = 'forrest'; // Intentional typo in locatio
14        break;
15      case Act
16        if ( location == 'forest') {
17          print('You found a hidden treasure!');
18        } else {
19          print('There is nothing to take here.');
20        }
21        break;
22      case Action.open:
23        if (location == 'cabin') {
24          print('You opened the cabin door.');
25        } else {
```

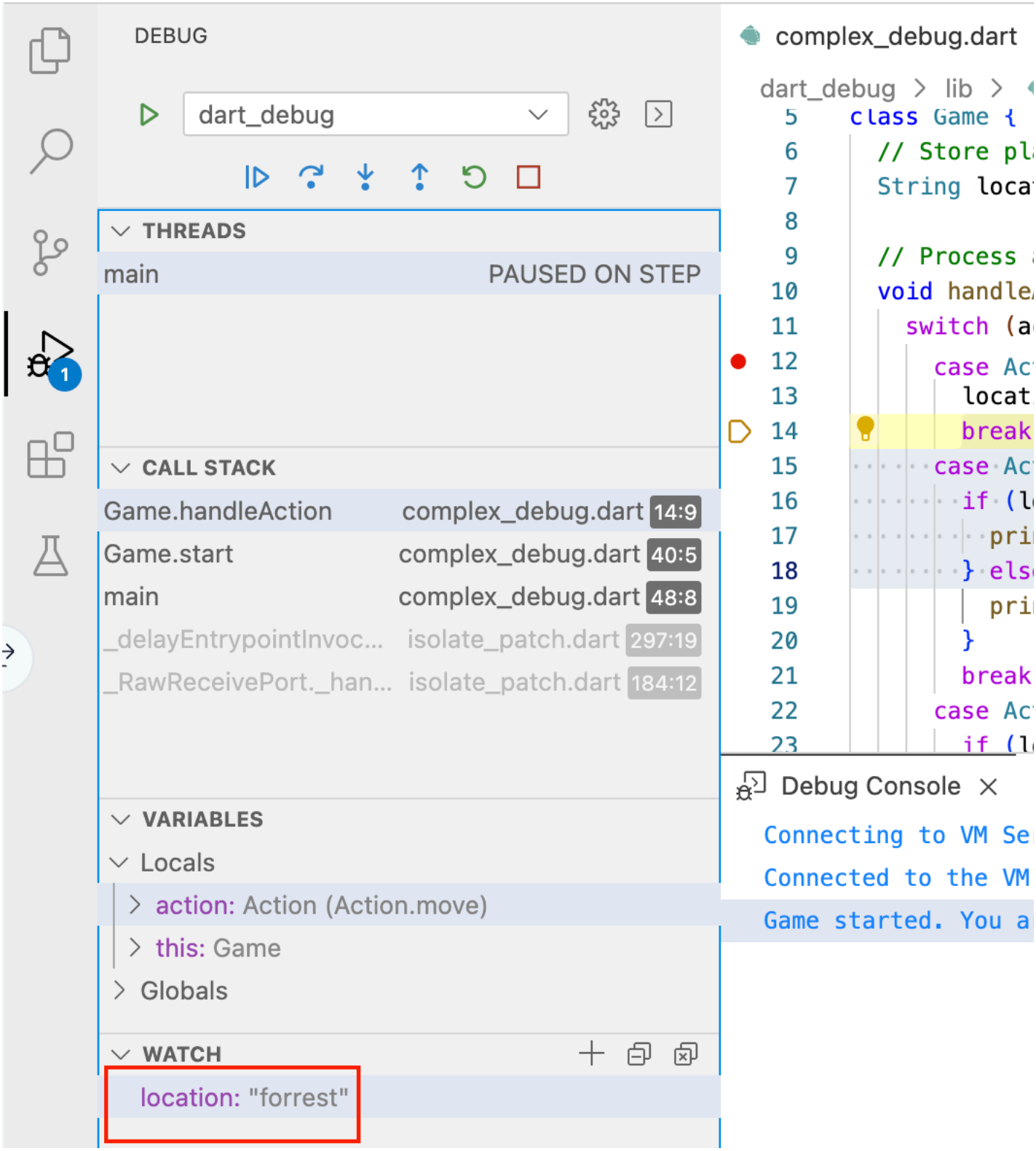
"forrest"

Step 7: Inspect variables and stepping through code

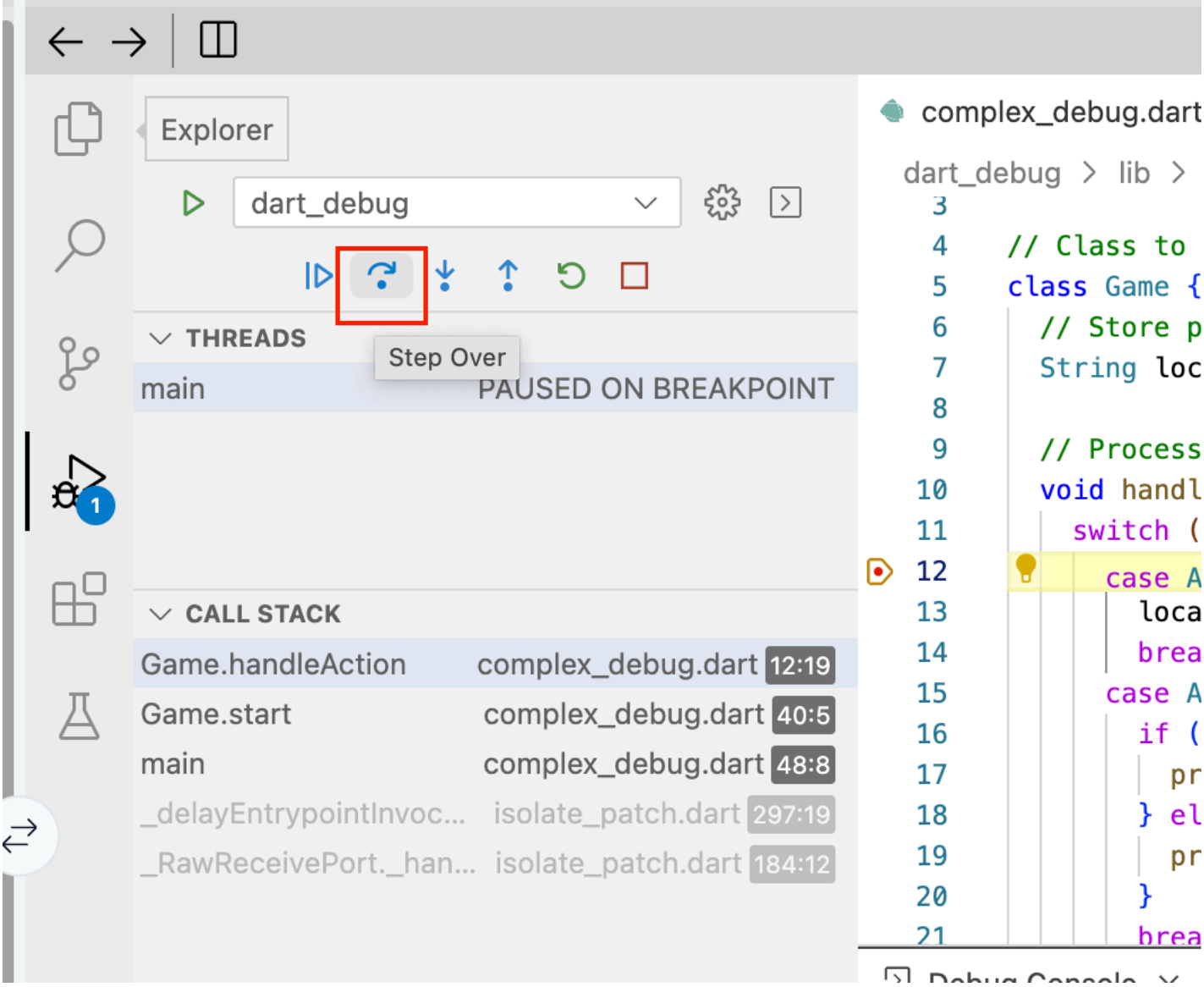
1. When execution is paused at the breakpoint, use the debugger to inspect the value of the `location` variable. You can also use the watch section to add specific variable to watch.



2. If you run the debug config again, you will see the value of the `location` variable is tracked when the breakpoint is triggered.



3. Once you hit a breakpoint, you can use the toolbar to move one step at a time by using the `step over` button. You can also dive into functions and methods by using the `step into` and `step out` buttons.



Step 8: Fix the code

Alright, now that you know what the problem is, let's go ahead and make the fix. The breakpoint told you that the `location` variable was incorrectly set to `forrest`, while the if statement is looking for `forest`. Go ahead and fix the error by changing the `location = 'forrest'` to `location='forest'`.

complex_debug.dart

dart_debug > lib > complex_debug.dart > Game > handleAction

```
5   class Game {
6     // Store player's current location
7     String location = 'start';
8
9     // Process actions taken by the player
10    void handleAction(Action action) {
11      switch (action) {
12        case Action.move:
13          location = 'forest'; // Intentional typo in location na
14          break;
15        case Action.take:
16          if (location == 'forest') {
17            print('You found a hidden treasure!');
18          } else {
19            print('There is nothing to take here.');
20          }
21          break;
22        case Action.open:
23          if (location == 'cabin') {
```

Debug Console

All

If you run the program now, you will see the correct output, and the user will find the treasure!

```
$ dart lib/complex_debug.dart
Game started. You are at the start.
You are at forest
You found a hidden treasure!
```

Conclusion and next steps

Here are some focused next steps that you can take to expand on your debugging skills after completing the lab:

- Investigate the use of conditional breakpoints that only trigger under specific conditions, such as certain values of variables or specific function calls, to better manage complex debugging scenarios.
- Practice using the watch panel to monitor changes in key variables or expressions over time, which can be especially helpful in loops or during repetitive function calls.
- Use the debugging tools to step through asynchronous code, understand how Dart handles async operations, and inspect the state of futures and streams.
- Since Dart can compile to JavaScript, practice debugging the application in different browsers to understand how cross-browser issues can be diagnosed and resolved.

Congratulations on completing this lab! You have learned how to set up and use debugging tools in Visual Studio Code to find and fix errors in a Dart application. This skill is crucial for developing robust and efficient applications.

Author(s)

Skills Network