Using the Debugger

About This Page:

Questions Answered: How can I execute my program step by step and examine what it does? What's a good tool for hunting runtime errors?

Prerequisites: This page assumes prior knowledge of topics from Weeks 1 to 4. You can make the most of this page if you read it sometime after reaching Chapter 4.2.

Points Available: None. This material is optional.

Related Modules: Miscellaneous.

Introduction: Tracing Program Execution

Programmers — and students of programming — commonly need to figure out what an existing program does. For instance, they may need to:

- develop an existing program further;
- write documentation about a program someone else wrote;
- locate errors in a buggy program (self-written or otherwise); or
- study an example program provided by a teacher.

In such scenarios, programmers often mentally trace the program's behavior step by step. Tracing is also useful while writing an entirely new program: in order to determine if the code works, the programmer needs to be able to "run it in their mind."

As a quick example, consider the `Experience` class from Chapter 3.4:

```
class Experience(val name: String, val description: String, val price: Double, val rating: Int) {

  def valueForMoney = this.rating / this.price

  def isBetterThan(another: Experience) = this.rating > another.rating

  def chooseBetter(another: Experience) = if (this.isBetterThan(another)) this else another

}
```

Here's a little program that calls some of the class's methods:

```
object ExperienceTest extends App {
  println("Starting the program.")
  val wine1 = new Experience("Il Barco 2001", "okay", 6.69, 5)
  val wine2 = new Experience("Tollo Rosso", "not great", 6.19, 3)
  val better = wine1.chooseBetter(wine2)
  var result = better.name
  println(result)
  result = "Better than wine1? " + better.isBetterThan(wine1)
  println(result)
}
```

Try running `ExperienceTest` mentally step by step. If you do it carefully, you'll notice that there is a whole bunch of things you need to keep track of at each step:

- where in the program you're currently at (which line and where on that line?);
- the class and object definitions that form the program (although you can easily refresh your memory if you see the code);
- each variable's values at the present time (`var`s demand particular attention);
- which data is associated with each object (via its instance variables), which is determined by object-creating commands and subsequent effects on state (if any);
- while running a method: Which object is the active `this` object for the method call? What are the values of each parameter variable during this method call?
- when returning from a method: Where was this particular method call initiated? Where should execution resume after returning?
- the relevant built-in operators, functions, and classes and how they are used; and
- any intermediate results that are formed while evaluating compound expressions.

Not all of those things are directly visible in program code. Since there are a lot of details to remember, it often makes sense to use an auxiliary tool that lightens the programmer's burden. That way, it is easier for the programmer to concentrate on the goals and structure of the program, possible bug locations, or whatever the task calls for.

A simple piece of paper can be a big aid: you can make notes of what happens during a program run and sketch out diagrams of the relevant objects and variables.

Instead of, or in addition to, pen and paper, you can use an auxiliary program that illustrates program execution step by step. In this ebook, you've already seen various programs illustrated as animated diagrams. Such animations aren't always available, though, so what to do?

Debuggers

A **debugger** is a utility program for examining the execution steps of other programs. You can run your program in a debugger and examine its state as you step through it line by line.

Debuggers' most prominent purpose is to help the programmer locate defects in code; hence the name.

The information that a debugger displays is similar to that shown in this ebook's animations. However, debuggers generally aren't quite as graphical and detailed as those animations, since they're usually designed for experienced professionals rather than learners; dealing with large programs calls for a different design. Nevertheless, beginner programmers too can benefit from using a debugger.

Some debuggers are standalone programs; others are integrated in an IDE. IntelliJ, for instance, gives you a debugger.

IntelliJ's Scala Debugger

Learning to handle a debugger fluently takes some effort. Here are some of the things you may want to do:

- Browse the list list of common debugging commands in IntelliJ.
- Do the little practice task further below on this page to try out the commands.
- Experiment on your own: write a small program and examine it in the debugger or explore O1's example programs.
- Find additional material online. For instance, here is `a YouTube video <https://www.youtube.com/watch?v=Fe3yB1pHq8w>` (that briefly covers even more debugger features than you need in O1) and IntelliJ's web site explains how to use the debugger (on some Java example code, but the idea is the same for Scala).
- Ask O1's teaching assistants to help you at the lab sessions.

A list of selected debugger commands

> **Setting a breakpoint**
> Before starting a program in the debugger, you'll usually want to set at least one *breakpoint*: a location in the program where execution should pause so that you can examine the program in detail. To do this, start by choosing a line in your program where you want the breakpoint. (The choice is up to you, but placing a breakpoint at the first command within your app object is a good idea to try first.) Put the cursor on that line and select Run → Toggle Breakpoint or press `Ctrl+F8`. Alternatively, you can click the margin between the line and the line number). Doing the same again toggles the breakpoint off. A red dot in the margin indicates that you've got a breakpoint there.
>
> **Launching a program in the debugger**
> Execute your program using the Debug command rather than the usual Run. You'll find it by selecting your app object and looking at its context menu or the Run menu in the menu bar at the top; pressing the bug icon lude in the tool bar also works. This command launches your program and runs it until the (first) breakpoint. (If you haven't set any breakpoints, your entire program will run much like usual.)
>
> **Executing one entire line of code**
> Select Run → Debugging Actions → Step Over or press `F8` (or the corresponding icon in the Debug tab). The computer will then execute the current line in full without displaying any intermediate steps. In particular, if the line contains a function call, the steps of executing the function will not be shown. Do this repeatedly to execute multiple lines.
>
> **Executing (part of) a line in detail**
> Select Run → Debugging Actions → Step Into or press `F7`. This command is similar to Step Over but doesn't necessarily execute the entire line: it displays pauses at certain important events within the line's execution. In particular, if the line contains a function call, Step Into "jumps inside the function" so that you can examine the function's execution steps in detail. Executing a single line of code can thus be split into several consecutive Step Intos. If there are multiple function calls on the same line, IntelliJ will ask you to use the arrow keys to pick which one you're interested in. (It may give you the option of stepping into a library function such as `println`, which is however seldom useful.)
>
> **Examining program state**
> You can browse the values of variables in the Debug tab that shows up at the bottom of the IntelliJ window once you launch the debugger.
>
> **Returning from a method to the call site**
> Select Run → Debugging Actions → Step Out or press `Shift+F8`. The method you were in runs until it returns and your debugger session resumes where that method was called.
>
> **Continuing execution without stepping**
> Select Run → Debugging Actions → Resume or press `F9`. Your program will run until the next breakpoint, or until the end of the entire program run if no more breakpoints are reached.
>
> **Stepping backwards**
> There's no command for this, but you can terminate the current debugger run and start over.
>
> **Stopping a debugging session**
> Stop the program with Run → Terminate or `Ctrl+F2` or use the other commands listed above until you reach the end of the program.

You'll find shortcuts to many of the above commands in the toolbar at the edges of the debugger view at the bottom. For example, you can Stop by pressing the stop button.

Student question: Why can't debuggers go backwards?

There are certain technical challenges to overcome, which is why most support for backwards stepping hasn't been built into most debuggers. But it's not impossible and might become more frequently supported in the future.

A Small Practice Task

Fetch the Miscellaneous project. It contains a class named `o1.excursion.Excursion` and an app object `ExcursionTest`.

Start here

First study the Scaladocs for `Excursion` and skim its program code. Then proceed.

Now drill into the given code; see below for guidance. As you do so, it will turn out that there is a little bug in the `Excursion` class.

The instructions below are only guideline. Do experiment on your own.

Examining `Excursion` in the debugger

Set a breakpoint in `ExcursionTest`, on the line that defines `val testTrip`.

Launch the program in the debugger.

Execution pauses at the breakpoint. Step through the program at your own pace. Explore:

- First, get to know the Step Over command (`F8`).
- Notice how the program's output appears step by step in the Console, which shows up during debugging as part of the Debug tab.
- Try to understand each execution step as it happens.
- Observe the values in the Debug tab's Variables section.
- Try the Step Into command (`F7`), too.
  - If at any point you end up seeing some strange code in a object named `Predef`, you've probably done a Step Into Scala's `println` function, which is probably not what you want to do. Not to worry. You can either Step Out (`Shift+F8`) or even start over; no sweat.
- Notice the list of frames on the call stack in the Debug tab. Notice how it changes as method calls begin and end. (The top part of the stack's description is where the interesting stuff happens.)
- Sooner or later, as you work your way through the program with Step Into and Step Over, you should notice something happen:

Did the call stack and the variables vanish in the Debug tab? Don't worry; that's to be expected:

- The program run stopped. Browsing the Console, you'll see that the culprit is an `IndexOutOfBoundsException` error.
- Under the error's name, you can see that the error arose while executing `lastParticipant` in the `Excursion` class, which has used the `ArrayBuffer` class. (`ArrayBuffer` is a library class that implements Scala's buffer collections.)
- The highlighted line of code "threw" an error. This error made the program crash.

You may already have noticed — and you can confirm from the stack trace — that the error occurred while running the `lastParticipant` method call that was initiated on line 26 in the code of `ExcursionTest`.

Use the debugger to study this error further. (Even if you already identified the bug, you can do this to practice.) See below for a suggested workflow.

Exploring the buggy method

Relaunch the debugger. Execution again pauses at the breakpoint that you set earlier.

Add another breakpoint on line 26.

Select Resume (`F9`). The program runs until the second breakpoint that you just set. The highlighted line, which produces the program crash, hasn't been executed yet.

Select Step Into (`F7`) and choose to step into `lastParticipants` (not `println`). You end up inside that method.

The next two lines call `numberOfInterested` and `numberOfParticipants`, but let's ignore the internal behavior of those methods here. Step Over the two lines that call those methods (`F8` twice). Executing those lines (the `if` line and the `val` line) do not yet crash the program.

Check the state of the `Excursion` object in the Debug tab. (In the Variables section, notice how `this` refers to that object while we're running `lastParticipant` on it.) Among other things, you should see the buffer that `interestedStudents` refers to and, within that buffer, the names of four people and the buffer's current size (4).

Make note of the local variable `numberOfLast` as well.

Recall: the error we got was `IndexOutOfBoundsException`, which means that an index wasn't within the appropriate range, given the buffer's current size. Can you spot the mistake in `lastParticipant`'s code? Study the code and explore further in the debugger as necessary.

If you execute the next line (that begins with `Some`), the program crashes again.

Points training 0 / 0 | My submissions 0 | Deadline Thursday, 1 April 2021, 12:00

You cannot submit this assignment

This course has been archived (Tuesday, 31 August 2021, 23:59).

Show anyway

Question 1

Adding which two characters will fix the error? (You might also want to add two space characters, but they don't count for this question.)

Submit

Placing breakpoints

In that example, we placed the breakpoints in the app object. Another alternative would have been to set a breakpoint directly into the `lastParticipant` method. Had we done so, the debugger would pause the program each time that method is invoked.

Similarly, if your program has a graphical user interface, you can place a breakpoint in one of the GUI's event-handler methods or one of the model's methods that is invoked by those event handlers. The debugger will then stop as you use the GUI and cause that method to be reached.

It's also possible to define a conditional breakpoint that triggers only under specific circumstances. Try right-clicking one of the red dots that mark breakpoints and explore. You can also try setting a breakpoint that interrupts the program whenever a runtime error occurs: Run → View Breakpoints → Java Exception Breakpoints.

An unfortunate limitation (in IntelliJ's Scala debugger)

The `ExcursionTest` program was structured like this:

```
object ExcursionTest extends App {

  runFactoryScenario()

  def runFactoryScenario() = {
    // Call various methods of Excursion here (in helper
    // function) and return here in between calls.
  }

}
```

That is, the `App` object had a function which was called once from within the object's body.

Nothing wrong with that. However, it's good to be aware that it's not a coindidence that we structured this example so.

The reason is a current (2020) limitation in IntelliJ's Scala debugger: if you want to debug *an app object that makes multiple method calls directly from the object's body*, the debugger won't show it nicely. That is, this will not behave nicely in the debugger:

```
object ExcursionTest extends App {

  // Call various methods of Excursion here (directly
  // in app object body) and return here in between calls.

}
```

Fortunately, you can easily circumvent the problem: add a helper function like `runFactoryScenario` above and call the methods from there.

We can expect this limitation to go away in future versions of IntelliJ and Scala, but for now it is what it is.

When you're debugging some other kind of program, with no need to return to the body of the `App` object from method calls, there's no need to worry about this bit.

On Debuggers and Debugging

Most of O1's programming assignments don't specifically exhort you to use the debugger. Nevertheless, it's a good idea to gradually make this tool part of your arsenal. When you run into bugs, remember that the debugger is available to use.

As you've already seen, even though the tool is called a "debugger", it doesn't magically fix any errors. It doesn't "even" locate the errors automatically. That still requires work from you the programmer. However, just the fact that the debugger helps you to trace program runs is sometimes a great time-saver.

Speaking of how humans are needed for debugging, here's a 49-second video of Steve Jobs advertising an IDE feature:

Summary of Key Points

- A programmer often needs to mentally run programs step by step.
- There is software that can help the programmer reason about programs. These tools are especially useful when the program is unfamiliar, buggy, and/or complex.
- A debugger is an auxiliary program that lets you examine the intermediate stages of a program run and the state of the program at those stages.
- IntelliJ has a debugger built in. You may find it useful in O1 and elsewhere.
- Links to the glossary: debugger, breakpoint.

Feedback

Not submitted | My submissions 0

You cannot submit this assignment

This course has been archived (Tuesday, 31 August 2021, 23:59).

Show anyway

Submit

Credits

Thousands of students have given feedback that has contributed to this ebook's design. Thank you!

The ebook's chapters, programming assignments, and weekly bulletins have been written in Finnish and translated into English by Juha Sorva.

The appendices (glossary, Scala reference, FAQ, etc.) are by Juha Sorva unless otherwise specified on the page.

The automatic assessment of the assignments has been developed by: (in alphabetical order) Riku Autio, Nikolas Drosdek, Joonatan Honkamaa, Jaakko Kantojärvi, Niklas Kröger, Teemu Lehtinen, Strasdosky Otewa, Timi Seppälä, Teemu Sirkiä, and Aleksi Vartiainen.

The illustrations at the top of each chapter, and the similar drawings elsewhere in the ebook, are the work of Christina Lassheikki.

The animations that detail the execution Scala programs have been designed by Juha Sorva and Teemu Sirkiä. Teemu Sirkiä and Riku Autio did the technical implementation, relying on Teemu's Jsvee and Kelmu toolkits.

The other diagrams and interactive presentations in the ebook are by Juha Sorva.

The O1Library software has been developed by Aleksi Lukkarinen and Juha Sorva. Several of its key components are built upon Aleksi's SMCL library.

The pedagogy of using O1Library for simple graphical programming (such as `Pic`) is inspired by the textbooks *How to Design Programs* by Flatt, Felleisen, Findler, and Krishnamurthi and *Picturing Programs* by Stephen Bloch.

The course platform A+ was originally created at Aalto's LeTech research group as a student project. The open-source project is now shepherded by the Computer Science department's edu-tech team and hosted by the department's IT services. Markku Riekkinen is the current lead developer; dozens of Aalto students and others have also contributed.

The A+ Courses plugin, which supports A+ and O1 in IntelliJ IDEA, is another open-source project. It was created by Nikolai Denissov, Olli Kiljunen, and Nikolas Drosdek with input from Juha Sorva, Otto Seppälä, Arto Hellas, and others.

For O1's current teaching staff, please see Chapter 1.1.

Additional credits for this page

Our thanks to the O1 student who recommended the Jobs video.

a drop of ink