UBC OKANAGAN

COSC 111
Computer Programming I

Practical Guide to Java Programming

Dr. Firas Moosvi

Acknowledgement: Slides (provided by Dr. Abdallah Mohammed) mainly rely on the materials prepared by Y. D. Liang for the textbook "Introduction to Java Programming, 10th Ed., Pearson Edu. Inc.".

Information about the Final Exam

- Check SSC for the official date and time of the Final Exam
- There will be some multiple choice questions, but the majority will be coding tasks
- The final exam will be:
  - **Cumulative**.
  - Live (2.5 hours), invigilated, but no proctoring.
  - Open book, open-notes, open-web but no cheating sites like Chegg/Course-Hero/Bartleby etc
  - IDEs are ok
  - On Canvas, hopefully using Gradescope

Announcements

- This week (Week 11) will be the last week of labs in the course!
- Next Monday April 5th is a holiday so lecture is cancelled, use this time to catch up
- Office hours will resume as normal after Easter Monday.
- In Week 13 I will give you a preview of Inheritance and what's to come in future COSC courses, and cover some stuff we skipped from the first week that will make a lot of sense to you now!

Student Evaluations of Teaching (SEoT)

- You may have received an email that student evaluations of teaching is now open for this course.
- Research shows that SEoT are flawed because they are influenced by unconscious and unintentional biases.

Student Evaluations of Teaching (SEoT)

- Despite their flaws, Teaching Evaluations are used to departments to:
  - Make decisions on Tenure and Promotion
  - Decide which courses instructors teach
  - Rate/rank grant applications and awards

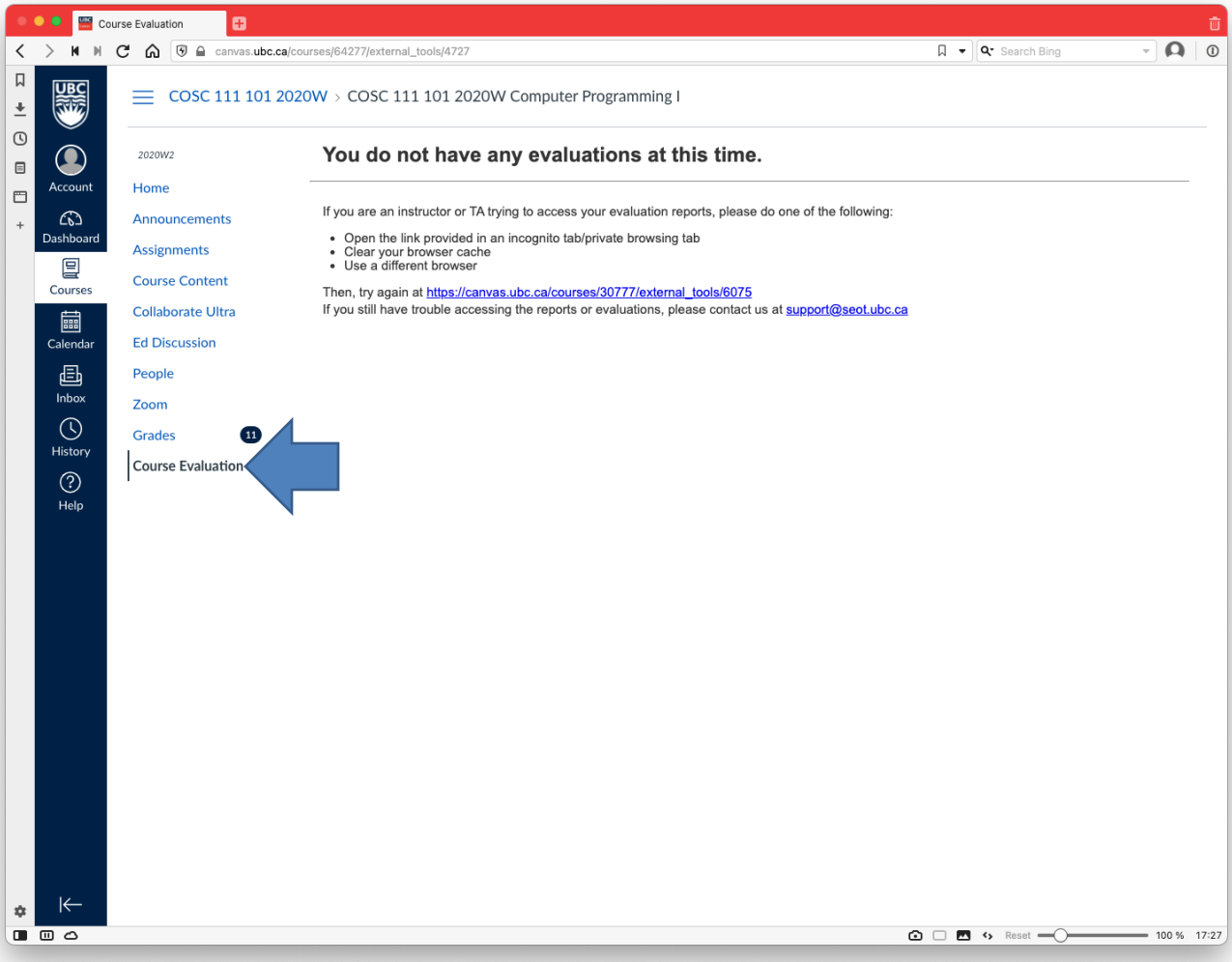
- More important to me however, is how you felt about the course content, the structure, and me as an instructor.

  I want to hear from you!

- My goal is to get at least a 70% response rate on SeOT, the more the merrier!

Student Evaluations of Teaching (SEoT)

- Course Evaluations will be open starting March 31



Conventions for Structuring Java Projects

Simple Java files

If your Java file is very simple and unsophisticated, you can just have a single .java file and keep all your code in one file.

main()

indexOfMin()

Q2.java > ...

```java
import java.util.Scanner;

public class Q2 {
    Run | Debug
    public static void main(String[] args) {
        double[] numbers = new double[10];
        // read input
        Scanner input = new Scanner(System.in);
        System.out.print("Enter ten numbers: ");
        for (int i = 0; i <= numbers.length; i++)
            numbers[i] = input.nextDouble();
        // invoke method and display output
        System.out.println("Index of min is " + indexOfMin(numbers));
        input.close();
    }

    public static int indexOfMin(double[] list) {
        double min = list[0];
        int minIndex = 0;
        // find min element and its index
        for (int i = 1; i < list.length; i++)
            if (min > list[i]) {
                min = list[i];
                minIndex = i;
            }
        return minIndex+2;
    }
}
```

Complex Java Projects

If your Java project starts having multiple methods, and classes, you will need to create a Java Project and put each class in a separate file.

This has several advantages:

- Easier to code (less scrolling)
- Easier to debug (allows you to pinpoint problematic methods)
- Easier to expand or refactor
- Pretty much required from now on!

Easiest thing to do is to have each Java project in its own folder. You can do this on Eclipse (File-> New Project) or in VS Code (

Demo

Debugging

Remember: **Programming Errors**

3 types of errors:

- Syntax Errors
  - Detected by the compiler
  - aka *compilation errors*

```
public class Errors {
    public static   main(String[] args) {
        System.out.println("Welcome to Java");
    }
}
```

- Runtime Errors
  - Causes the program to abort during the runtime.

```
public class Errors {
    public static void main(String[] args) {
        System.out.println(1 / 0);
    }
}
```

Can't divide by zero

- Logic Errors
  - Produces incorrect result during the runtime
  - no error message is shown

```
public class Errors {
    public static void main(String[] args) {
        System.out.println("35 Celsius in Fahrenheit:");
        System.out.println((9 / 5) * 35 + 32);
    }
}
```

Output is incorrect due to wrong formula

Debugging your code

Debugging is the act of finding and correcting errors in a system.

A common **reason for computer errors** is our **lack of precision** in specifying instructions to the computer

As a programmer, you need to know how to debug your code.

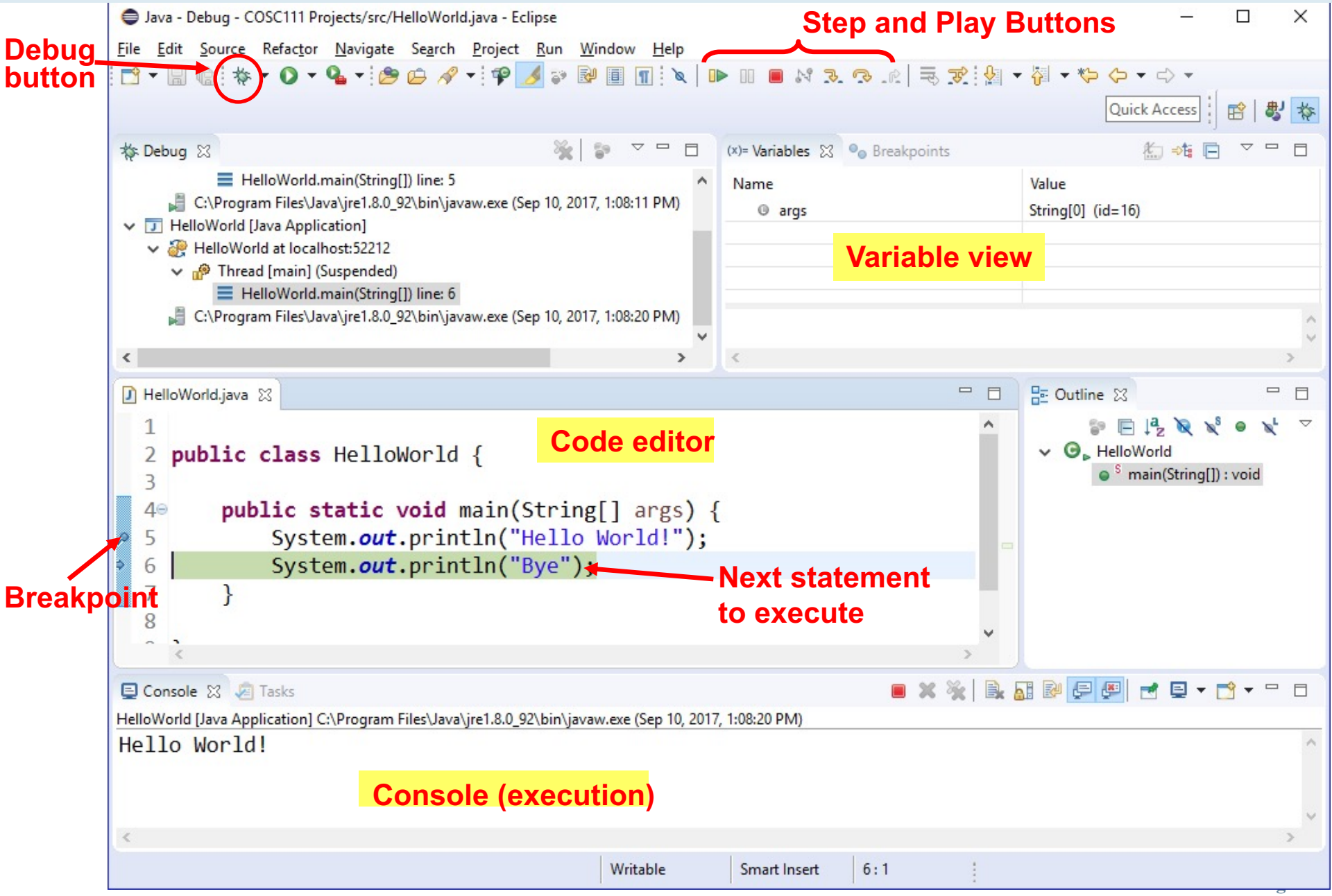
Eclipse provides us with tools to help us identify the source of errors our code.

Both Syntax and Runtime errors are easily found whenever they occur (with the help with the error message that appears on the console).

- Syntax errors are identified before compilation.
- Runtime errors are identified while the program is running.

Logic errors can be located using Eclipse Debugger

Remember: Eclipse: Debugging and Breakpoints



Basic concepts

Debugger: a tool that allows you to run a program interactively while watching how your code runs and how the variables change.

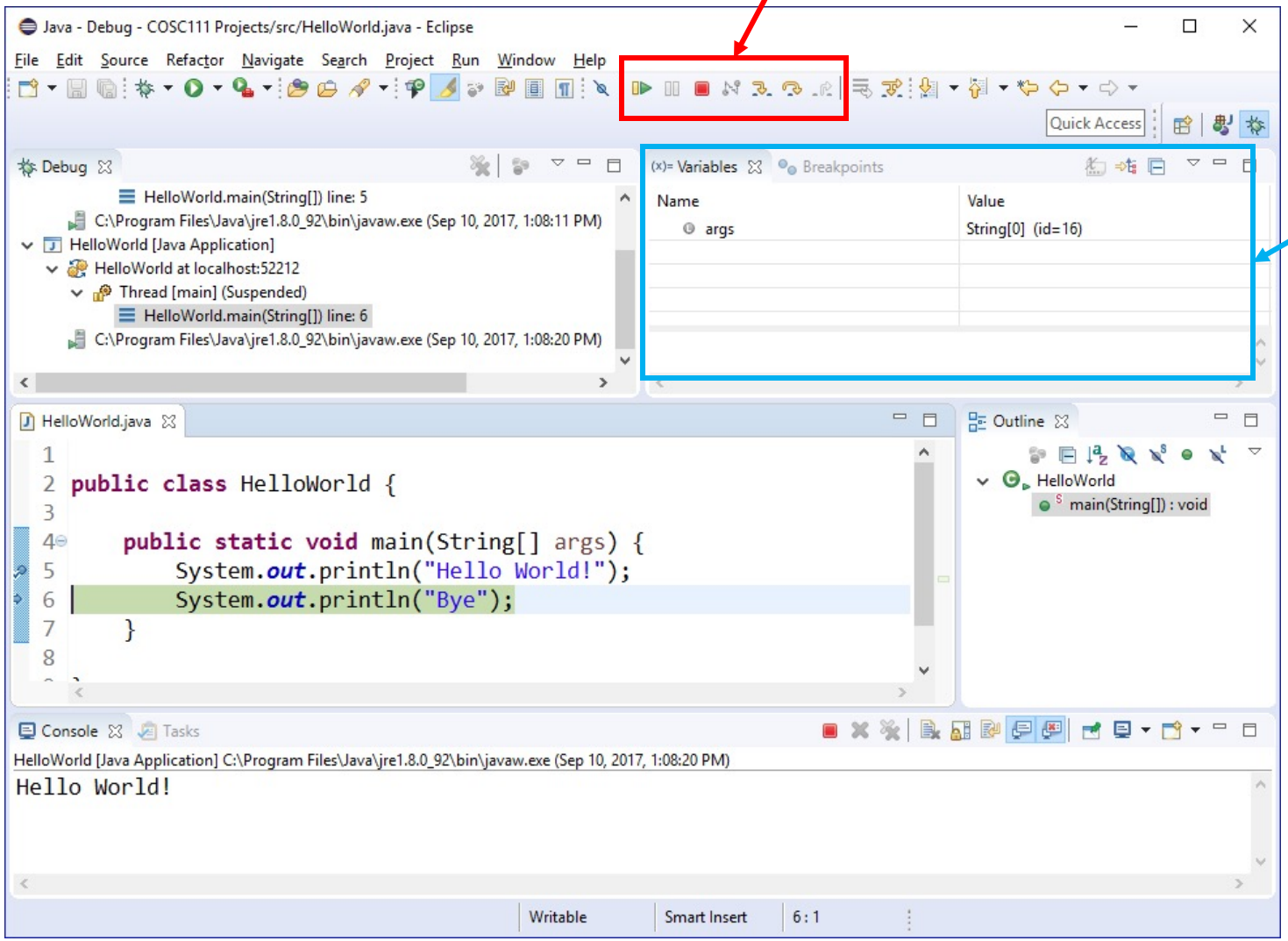
- To start debugging your code, do one of the following:
  - right click on the class file and select *Debug As → Java Application*
  - From the *Run menu*, choose *Debug*.
  - Click the debug button.
- This will open the Debug perspective in Eclipse.
  - You can switch to the default Java perspective by choosing Window menu → Open perspective → Java Browsing.

Breakpoint: a point in your source code where the program execution stops during debugging.

- Once the program stops, you can inspect the variables and run the program in a controlled manner.
- To define a breakpoint in your program, do one of the following:
  - double-click in the left margin of the Java editor in Eclipse.
  - right-click in the left margin and select Toggle Breakpoint.

Basic concepts, cont'd

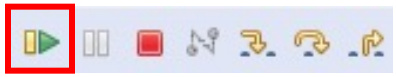
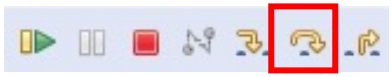
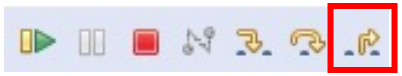
Code execution can be controlled from here



Variables can be monitored here

Code Execution Controls

When your code stops at a breakpoint, you can use the following to control the execution of your program:

- Resume (F8) 
  - Continue execution till the next breakpoint.
- Step over (F6) 
  - Execute the given statement and move to the next one. If the statement contains a method, the debugger will **not** go into each line of the method.
- Step into (F5) 
  - Runs the same as “step over” if the statement does not contain a method. But if it does, the debugger will enter the method and continue debugging there.
- Step out (F7) 
  - get out of a method back to the statement where the method was called.

Try this …

Create a class called Test1 (or any name of your choice) with this code in eclipse and then follow the steps on the following few slides.

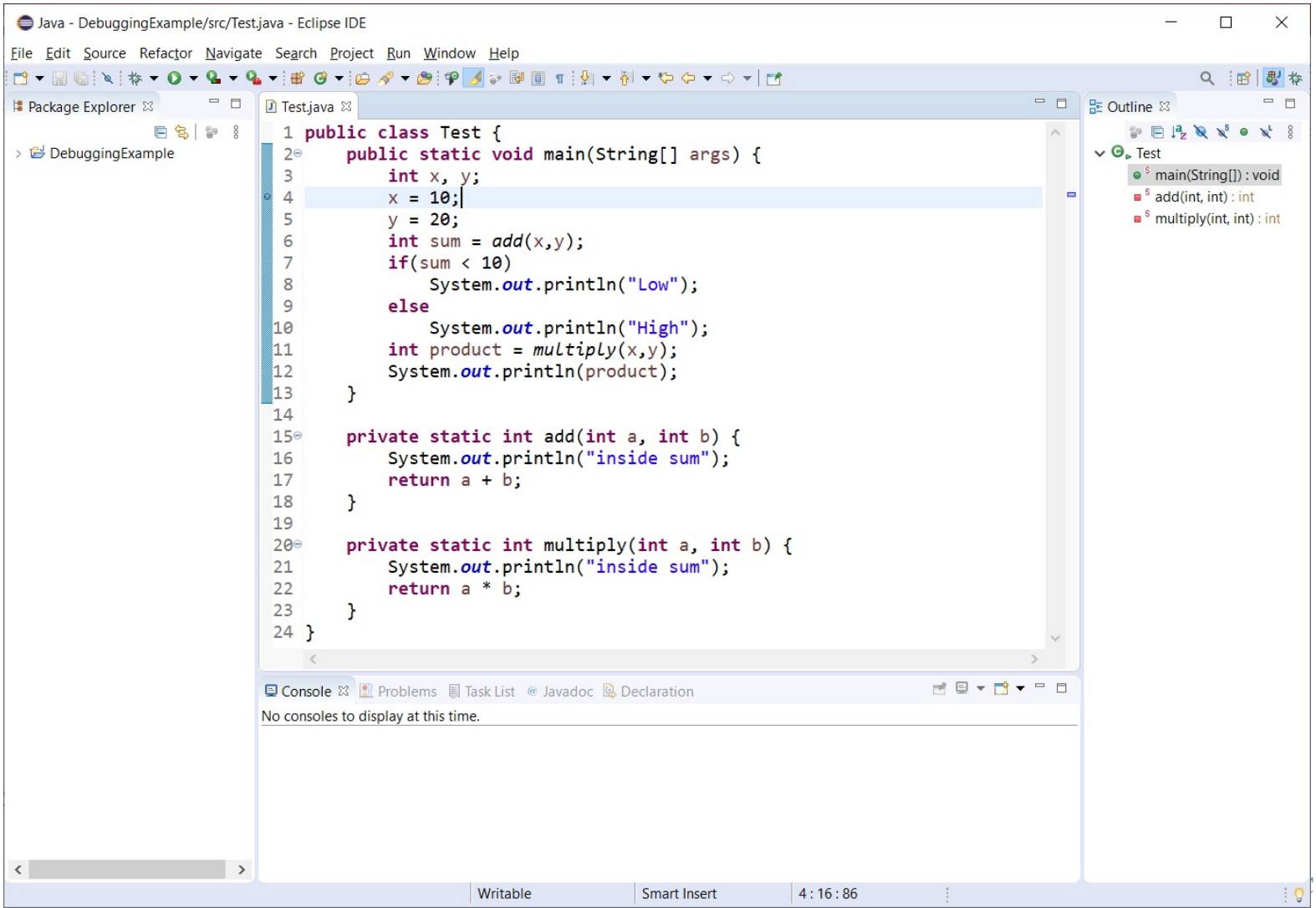
```
public class Test1 {
  public static void main(String[] args) {
     int x, y;
     x = 10;
     y = 20;
     int sum = add(x,y);
     if(sum < 10)
       System.out.println("Low");
     else
       System.out.println("High");
     int product = multiply(x,y);
     System.out.println(product);
  }

  private static int add(int a, int b) {
     System.out.println("inside sum");
     return a + b;
  }

  private static int multiply(int a, int b) {
     System.out.println("inside sum");
     return a * b;
  }
}
```

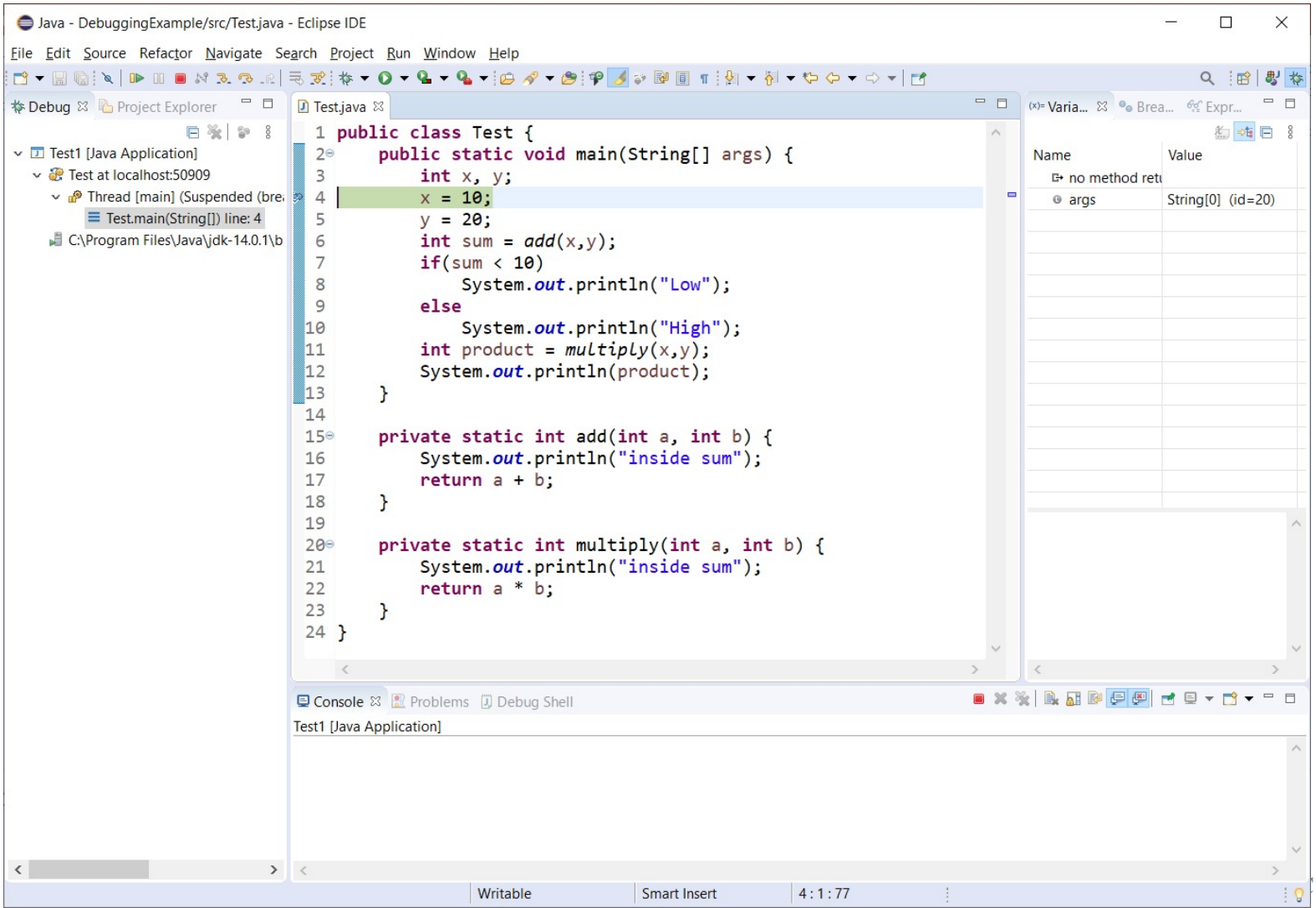
Tutorial

1) Set a breakpoint at statement #4, i.e. at x = 10;



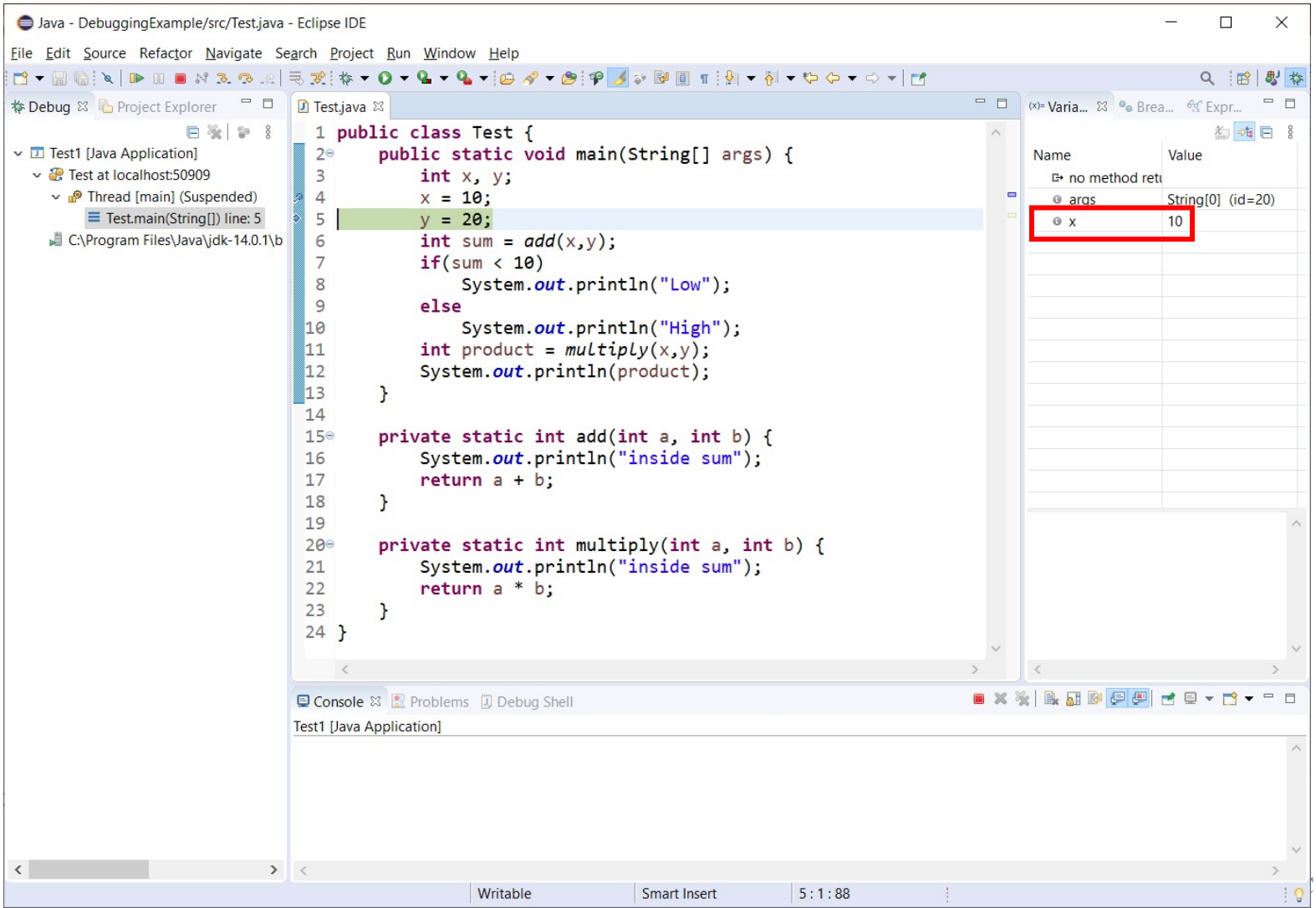
Tutorial, cont'd

2) Start the debugger. Note how the execution is suspended at the breakpoint. The statement highlighted in green has not been executed yet.



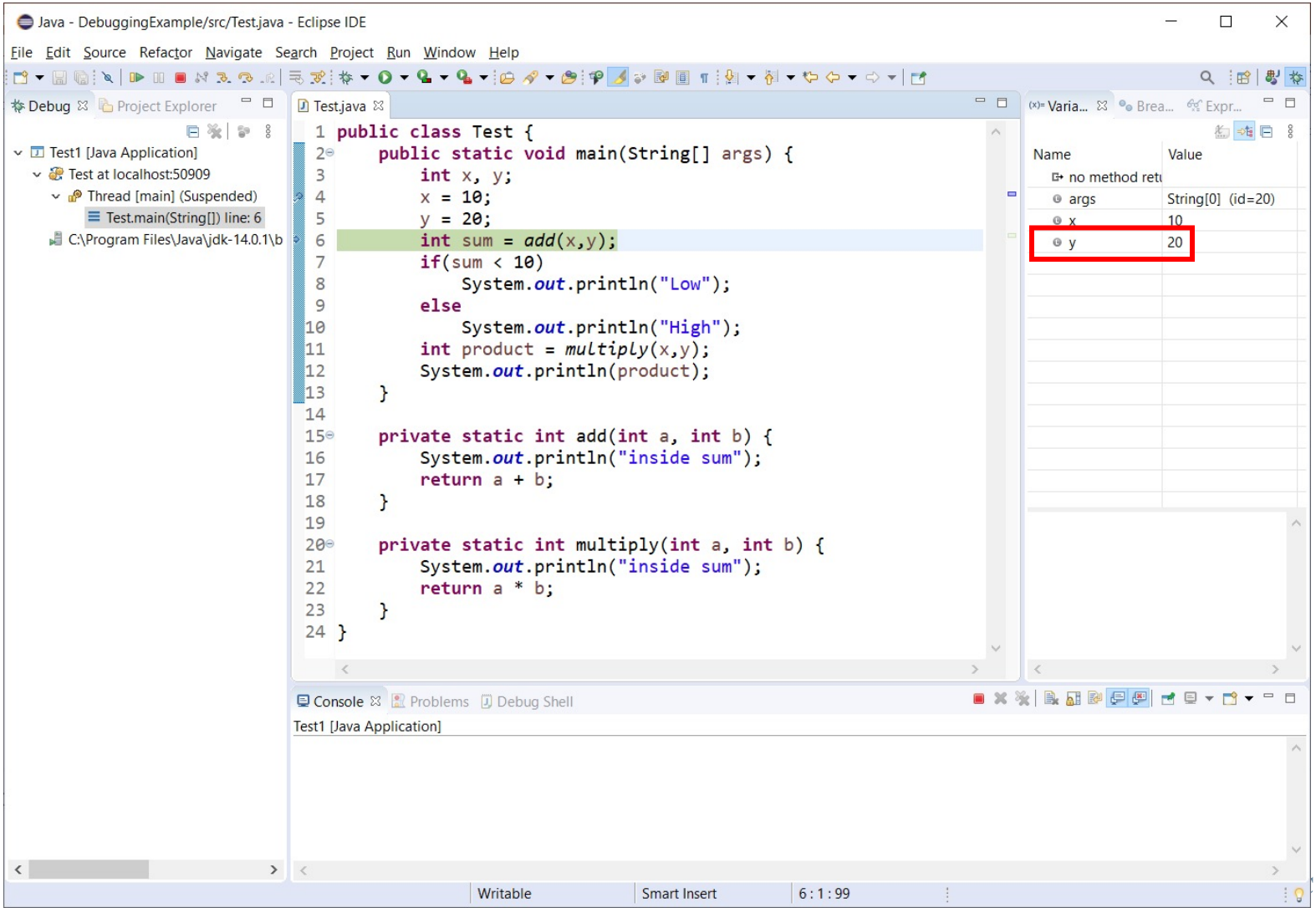
Tutorial, cont'd

3) Proceed by hitting F5 (step into) – this will set x to 10, and move to next statement. Note how x is not added to the variable list on the right.



Tutorial, cont'd

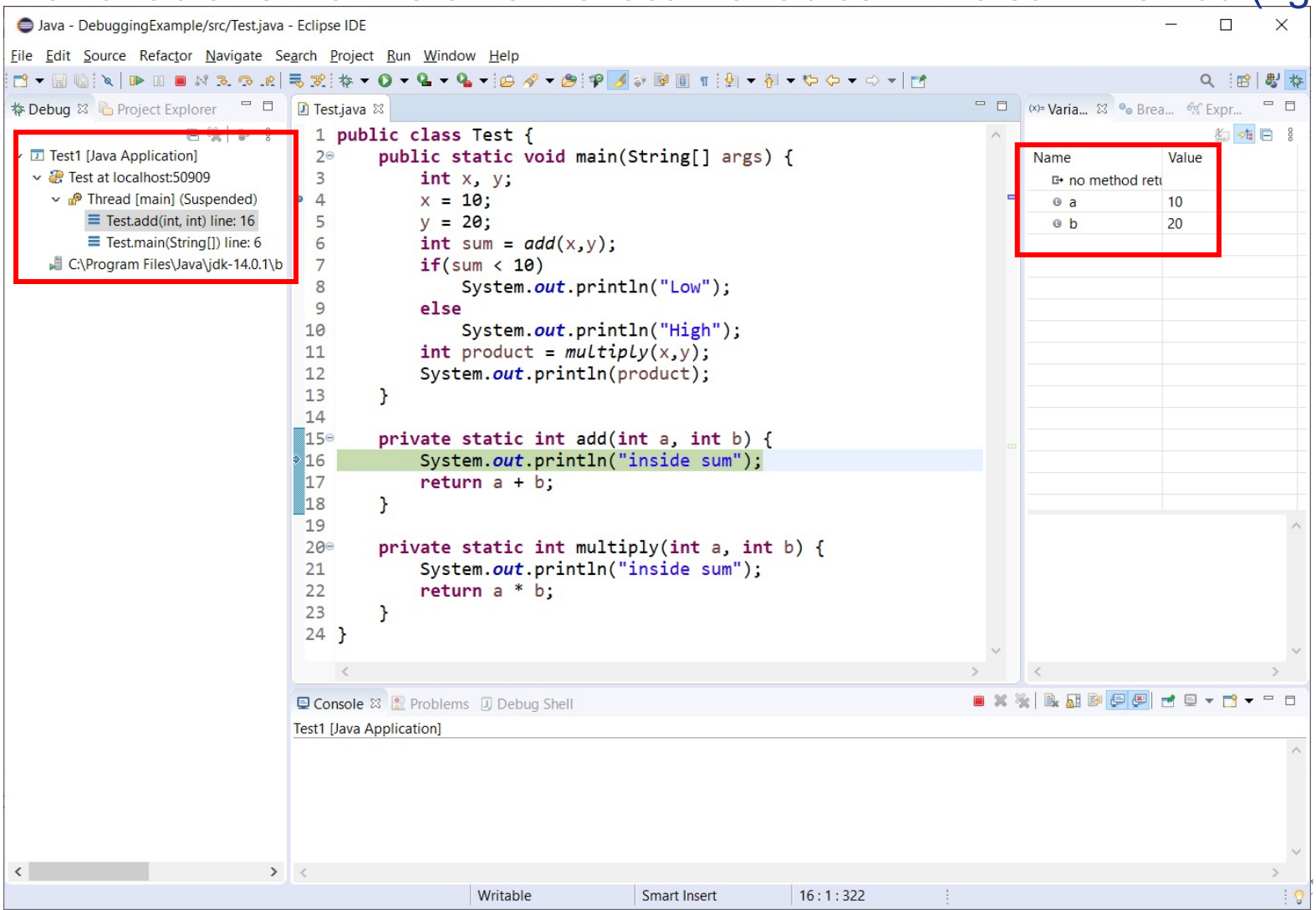
4) Press F5 (step into) again – this will set y to 20 and add it to the variable list, then move to next statement.



Tutorial, cont'd

5) Press F5 (step into) again – this will take you into the sum method. Note:

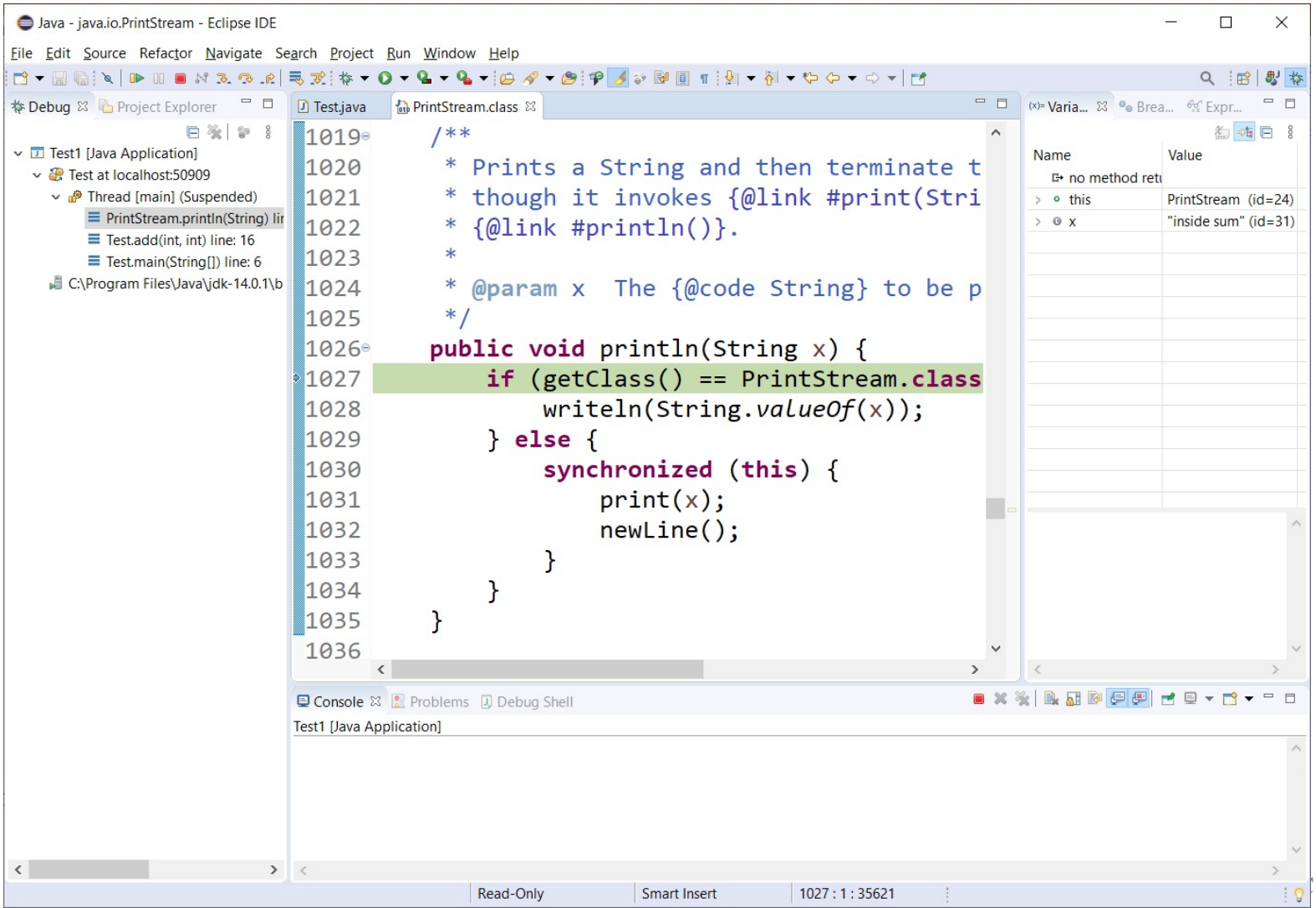
- The call stack now refers to the sum method (on the left)
- The variable list now refer to the local variables in the sum method (right)



Tutorial, cont'd

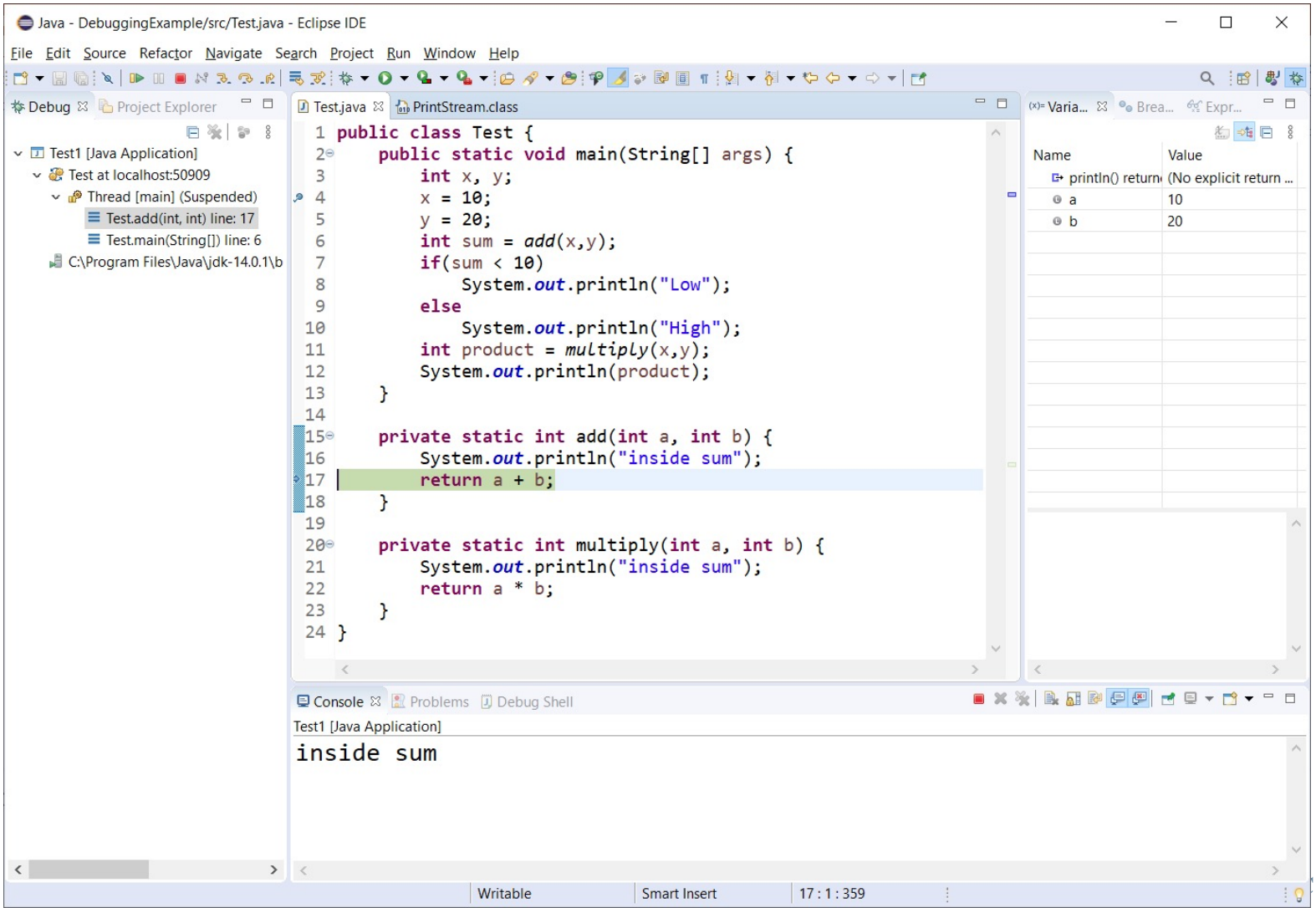
6) Press F5 (step into) again – note how we went into the println() method.

- We should have pressed on F6 (step over) to run the println() method without going into it. However, we will fix this in the next step.



Tutorial, cont'd

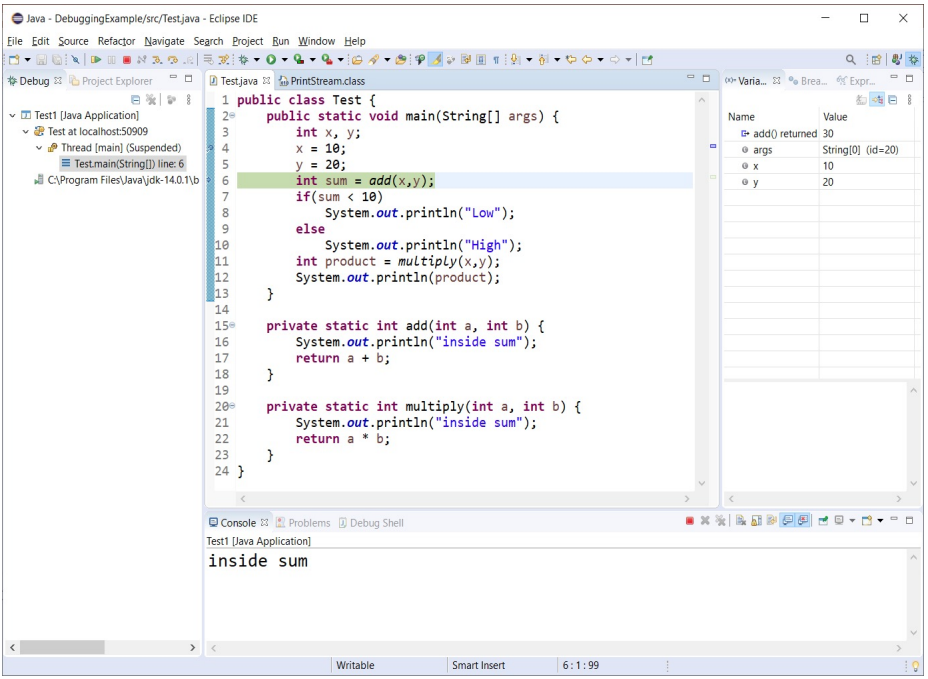
7) Press F7 (step out) to finish the execution of the println() method and get out of it back to your program.

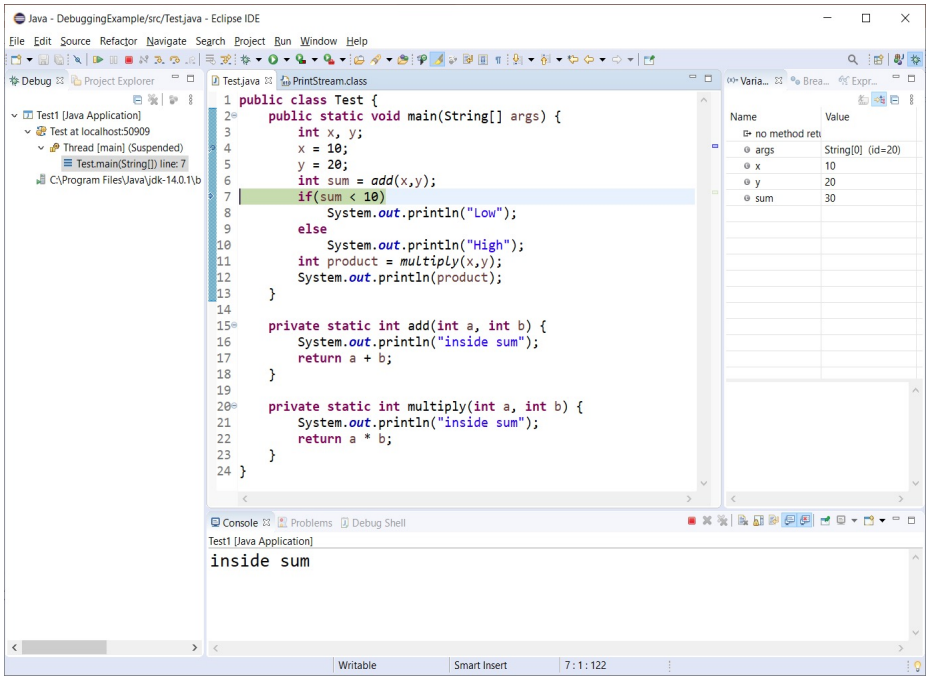


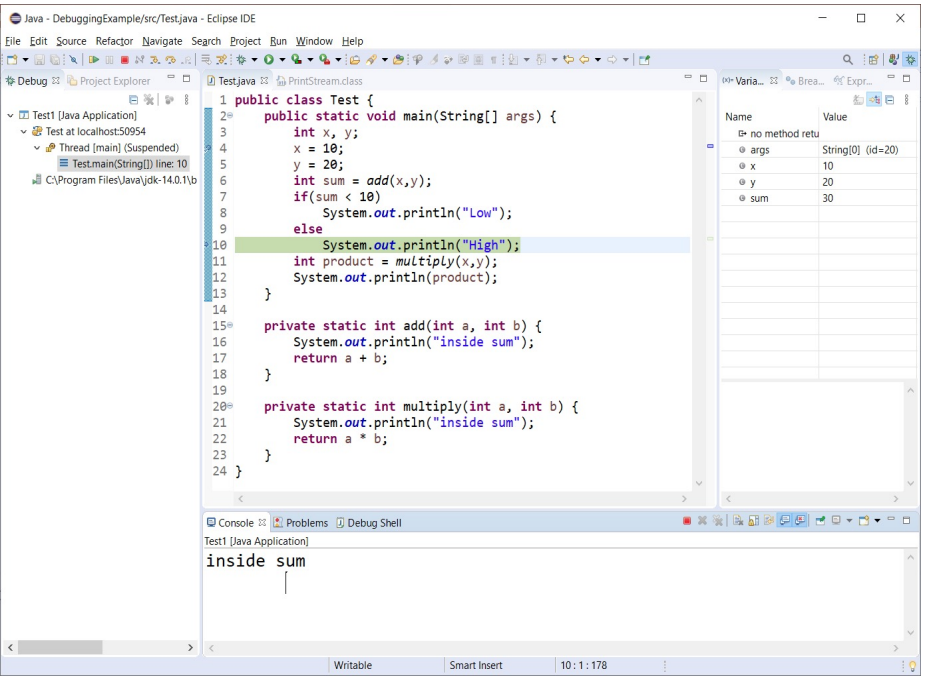
Tutorial, cont'd

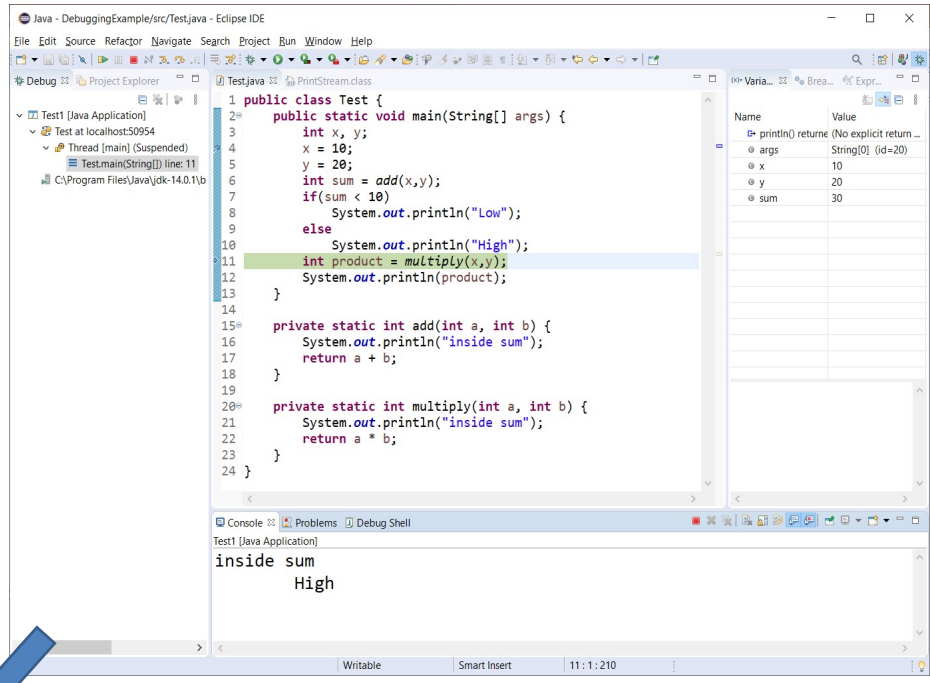
8) Press F6 (step over) a few more times and note:

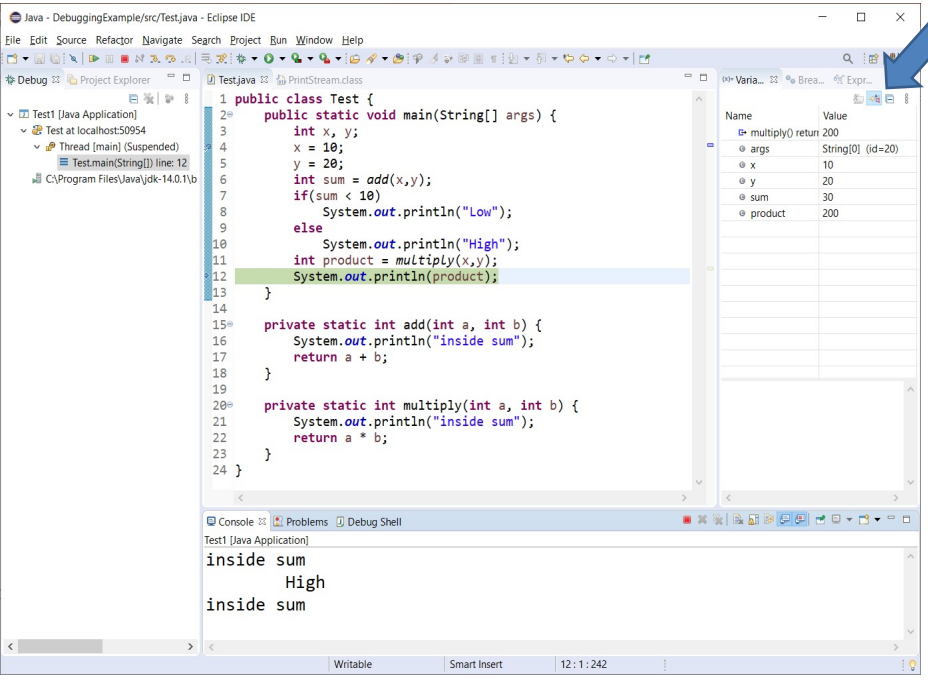
- how the variables change after you execute every statement.
- Which statements run or skipped (e.g. in the if-statement)
- How we don't get into any methods (neither println() nor multiply()).

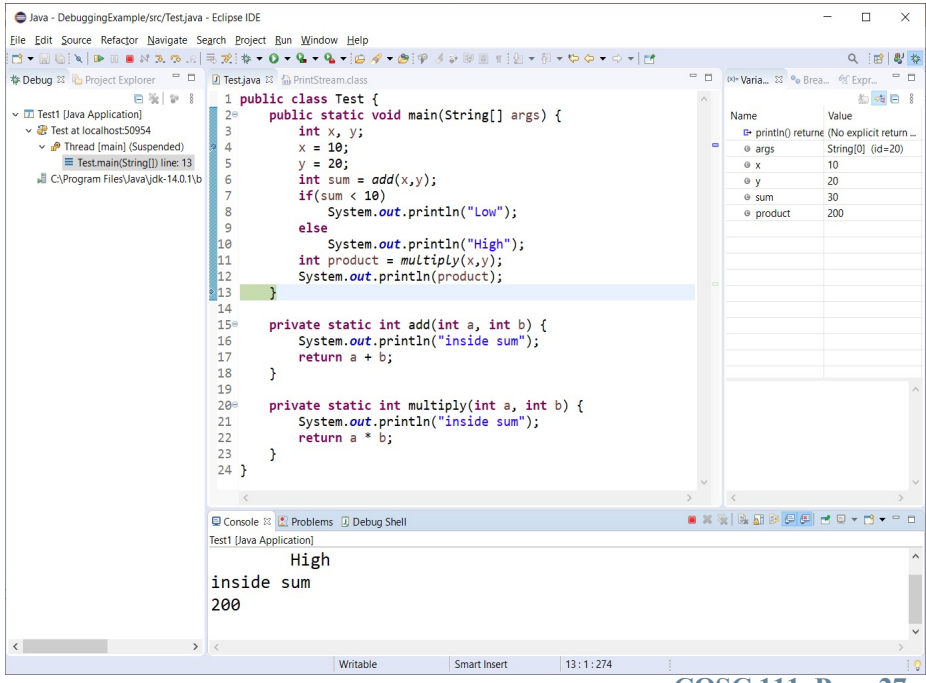




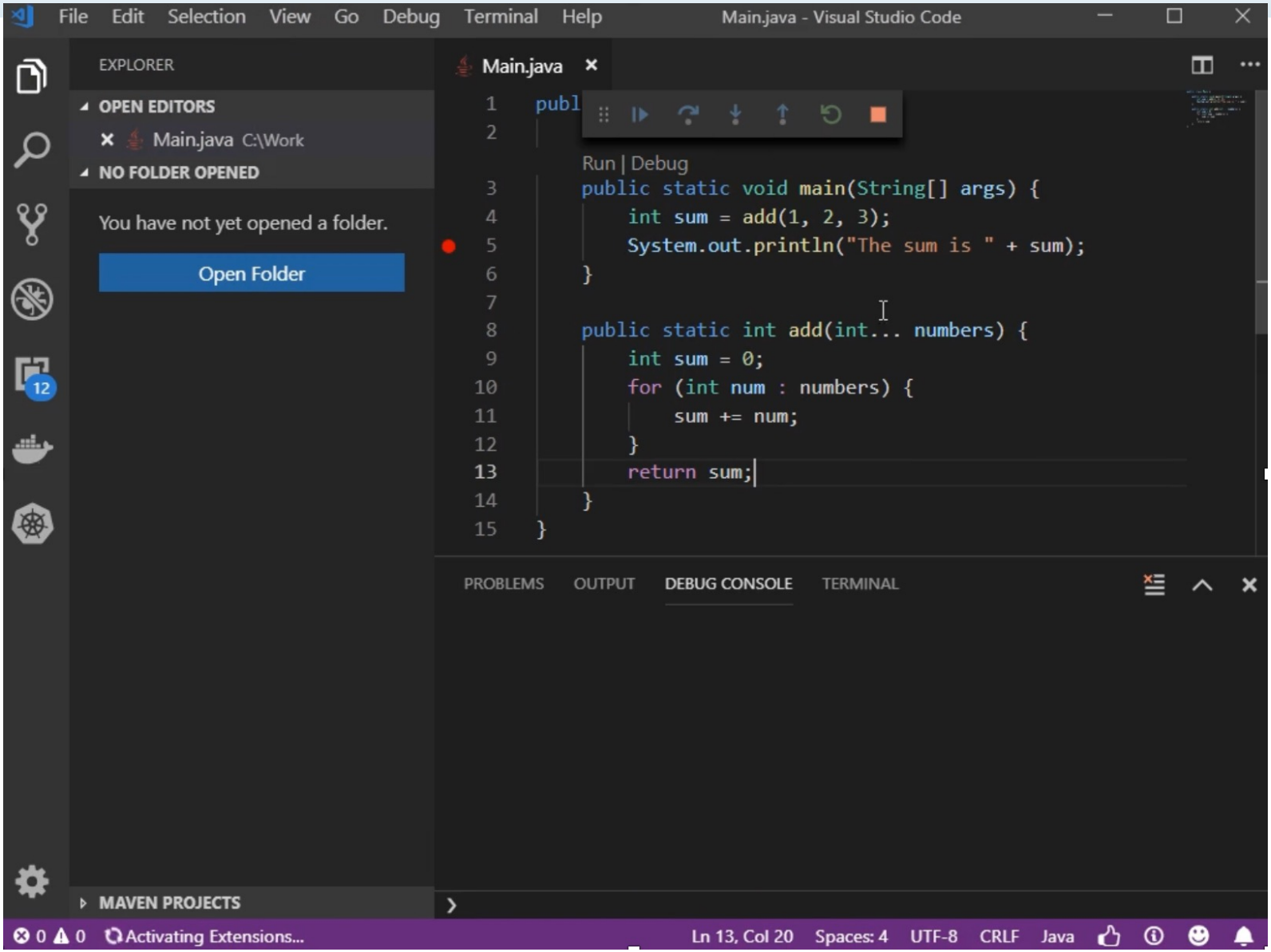

Java - DebuggingExample/src/Test.java - Eclipse IDE
public class Test {
public static void main(String[] args) {
int x, y;
x = 10;
y = 20;
int sum = add(x,y);
if(sum < 10)
System.out.println("Low");
else
System.out.println("High");
int product = multiply(x,y);
System.out.println(product);
}
private static int add(int a, int b) {
System.out.println("inside sum");
return a + b;
}
private static int multiply(int a, int b) {
System.out.println("inside sum");
return a * b;
}
}
inside sum


Java - DebuggingExample/src/Test.java - Eclipse IDE
inside sum
High


Java - DebuggingExample/src/Test.java - Eclipse IDE
inside sum
High
inside sum

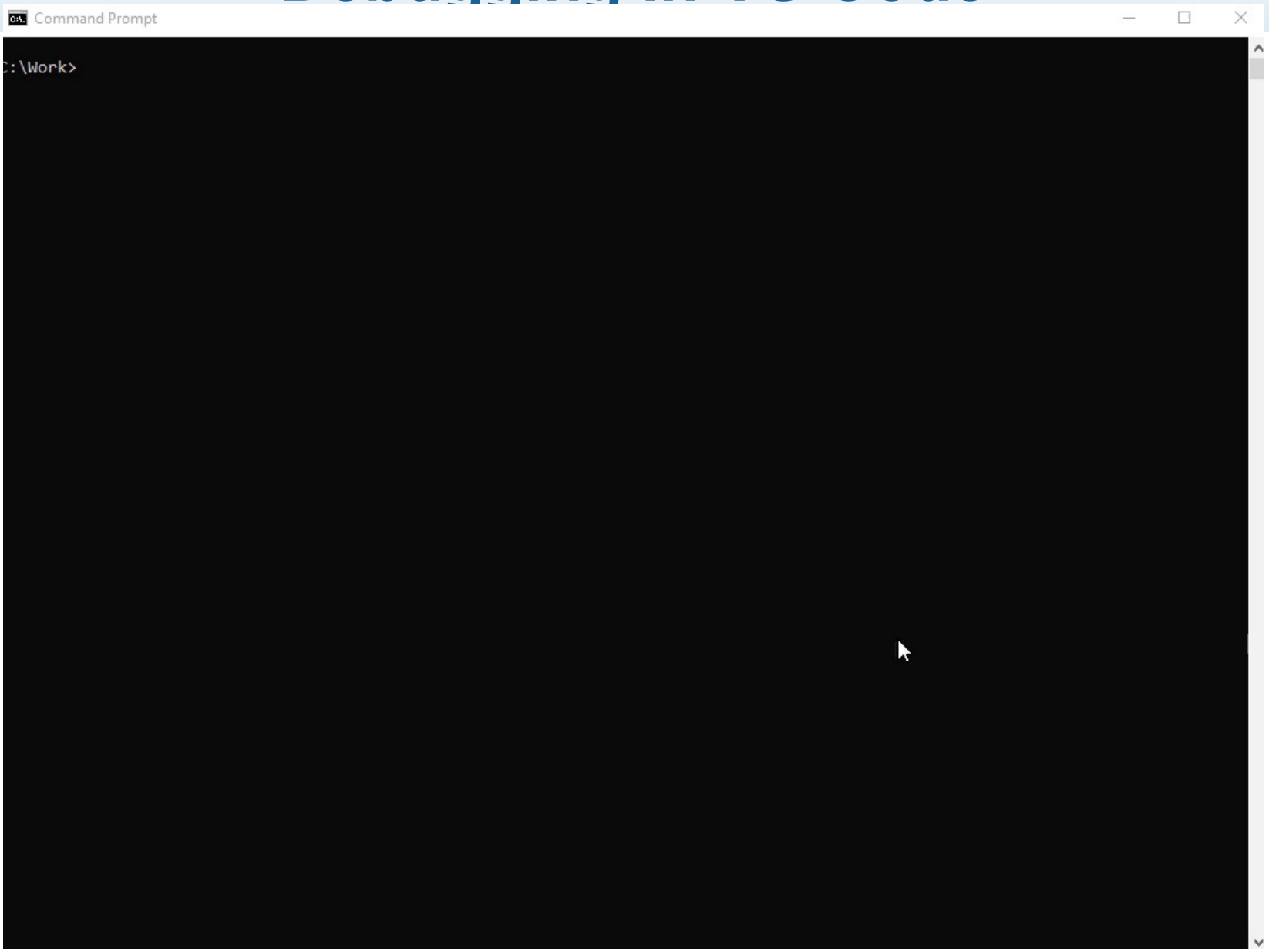

Java - DebuggingExample/src/Test.java - Eclipse IDE
High
inside sum
200

Debugging in VS Code



Source: VS Code Docs

Debugging in VS Code



Source: VS Code Docs