



cosc 111
Computer Programming I

Practical Guide to Java Programming

Dr. Firas Moosvi

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)

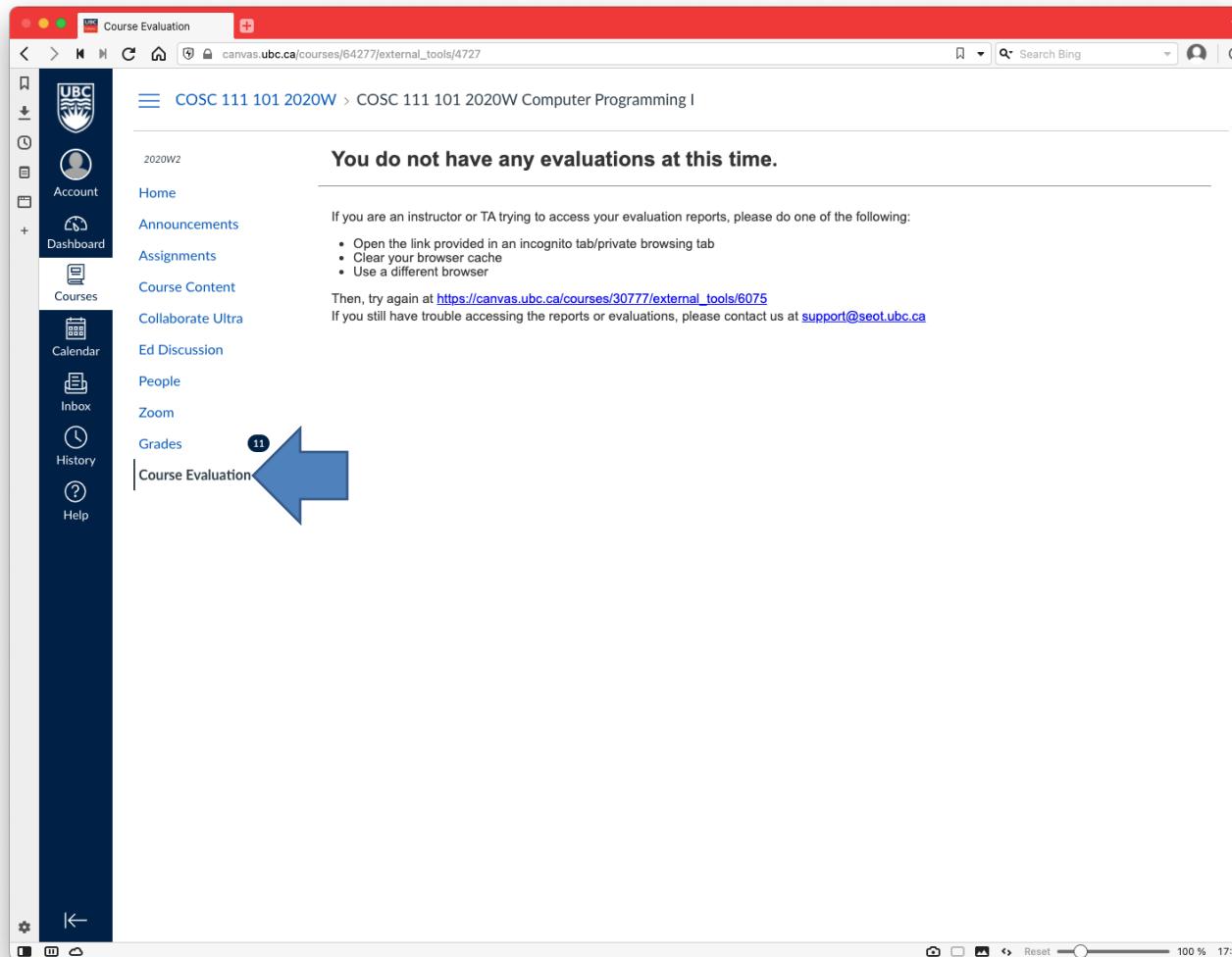
- Despite their flaws, Teaching Evaluations are used to departments to:
 - Make decisions on Tenure and Promotion
 - Decide which courses instructors teach
 - Rate/rank grant applications and awards
- More important to me however, is how you felt about the course content, the structure, and me as an 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()

```
Q2.java > ...
1 import java.util.Scanner;
2
3 public class Q2 {
4     Run | Debug
5     public static void main(String[] args) {
6         double[] numbers = new double[10];
7         // read input
8         Scanner input = new Scanner(System.in);
9         System.out.print("Enter ten numbers: ");
10        for (int i = 0; i <= numbers.length; i++)
11            numbers[i] = input.nextDouble();
12        // invoke method and display output
13        System.out.println("Index of min is " + indexOfMin(numbers));
14        input.close();
15    }
16
17    public static int indexOfMin(double[] list) {
18        double min = list[0];
19        int minIndex = 0;
20        // find min element and its index
21        for (int i = 1; i < list.length; i++)
22            if (min > list[i]) {
23                min = list[i];
24                minIndex = i;
25            }
26    }
27 }
```

indexOfMin()

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).

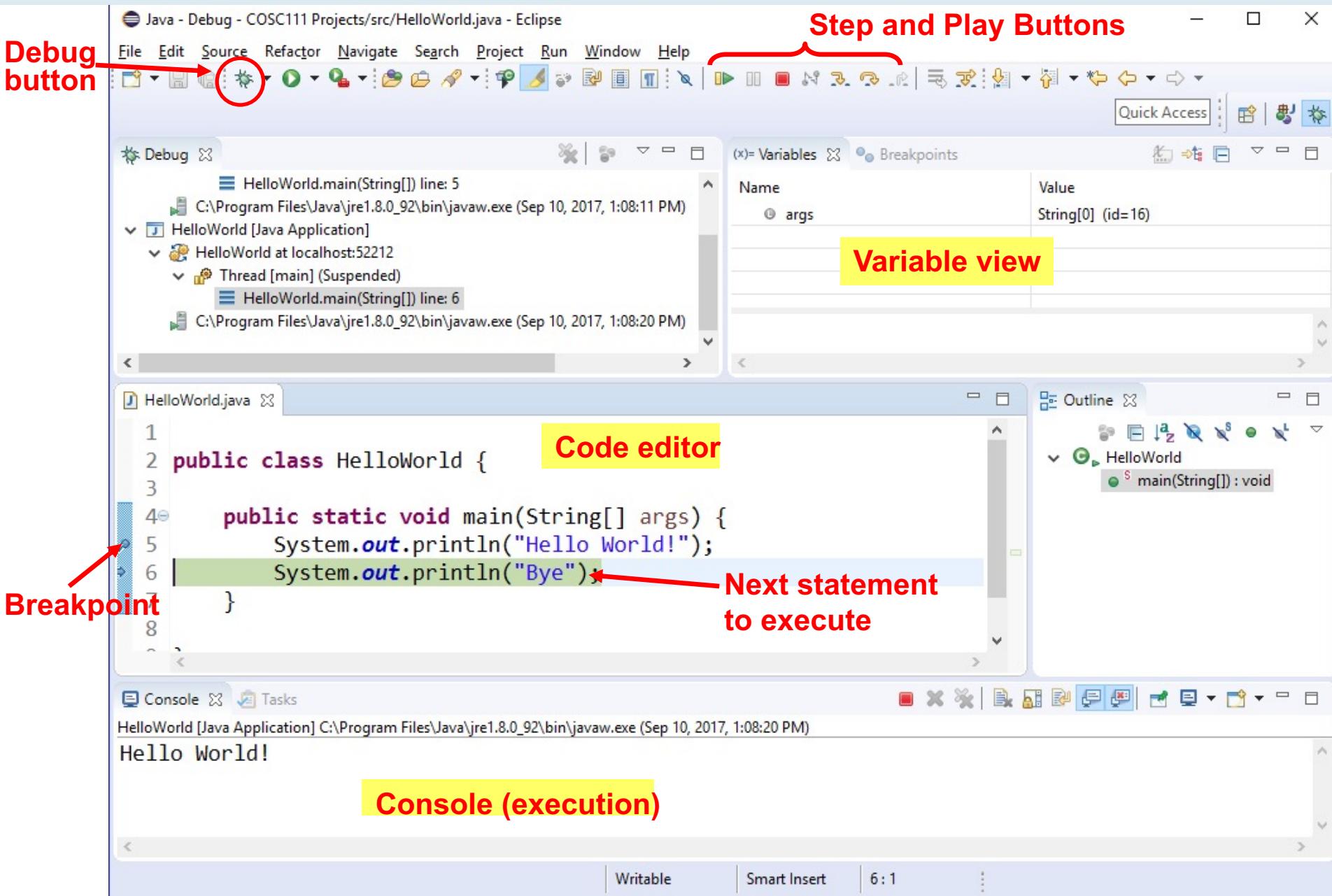
- Syntax errors are identified before compilation.
- Runtime errors are identified while the program is running.

Logic errors can be located using Eclipse Debugger

Remember: Eclipse: Debugging and Breakpoints

Debug button

Step and Play Buttons



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.

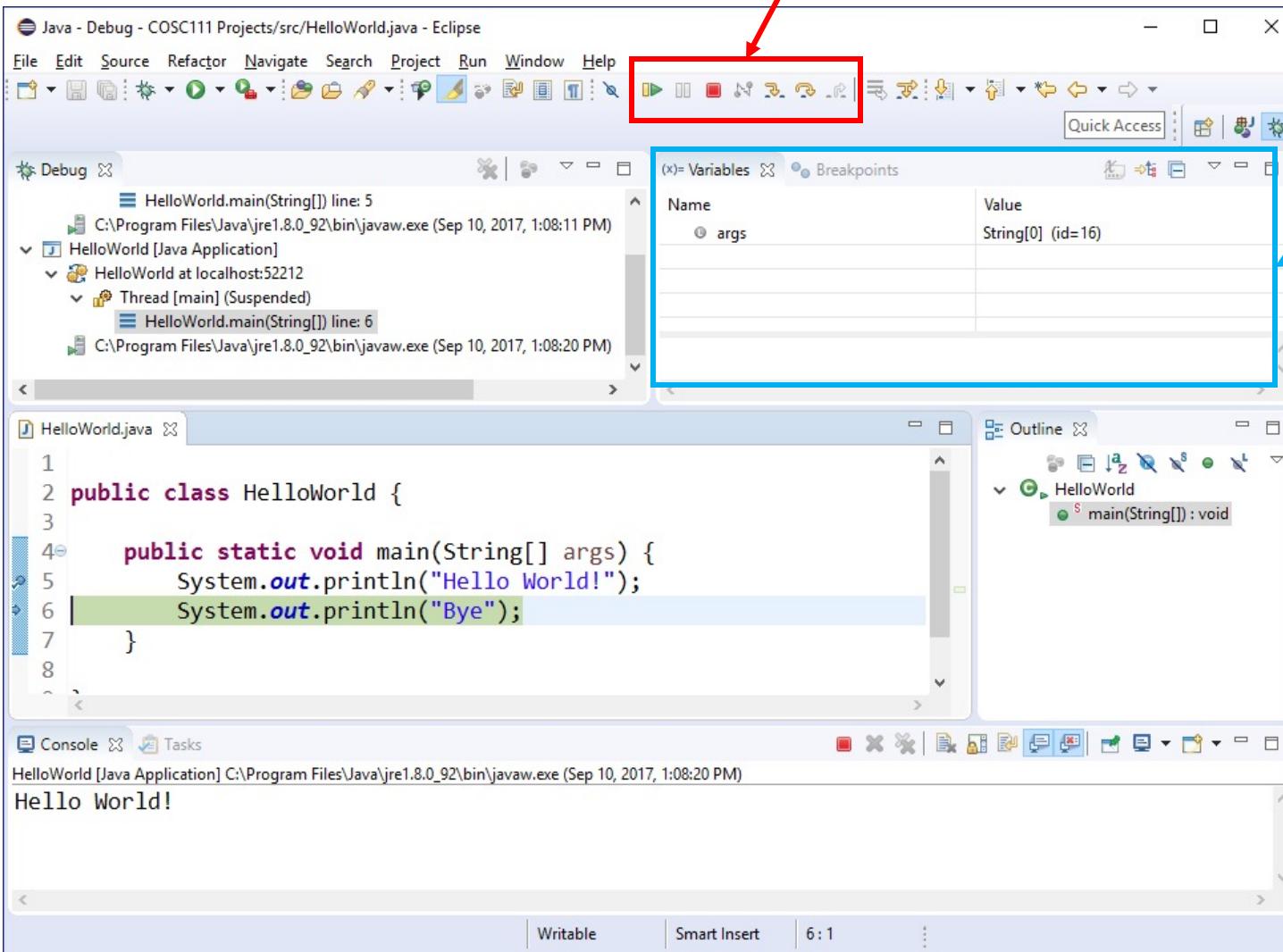
- Once the program stops, you can inspect the variables and run the program in a controlled manner.
- To define a breakpoint in your program, do one of the following:
 - double-click in the left margin of the Java editor in Eclipse.
 - right-click in the left margin and select *Toggle Breakpoint*.

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 $x = 10$;

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** Java - DebuggingExample/src/Test.java - Eclipse IDE
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Standard Eclipse toolbar icons.
- Package Explorer:** Shows a single project named DebuggingExample.
- Test.java Editor:** The main code editor window containing the following Java code:

```
1 public class Test {  
2     public static void main(String[] args) {  
3         int x, y;  
4         x = 10; // Breakpoint is set here.  
5         y = 20;  
6         int sum = add(x,y);  
7         if(sum < 10)  
8             System.out.println("Low");  
9         else  
10            System.out.println("High");  
11         int product = multiply(x,y);  
12         System.out.println(product);  
13     }  
14  
15     private static int add(int a, int b) {  
16         System.out.println("inside sum");  
17         return a + b;  
18     }  
19  
20     private static int multiply(int a, int b) {  
21         System.out.println("inside sum");  
22         return a * b;  
23     }  
24 }
```
- Outline View:** Shows the class structure with methods main, add, and multiply.
- Console View:** Shows the message "No consoles to display at this time."
- Bottom Status Bar:** Writable, Smart Insert, 4:16:86, and a date/time stamp.

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.

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** Java - DebuggingExample/src/Test.java - Eclipse IDE
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Standard Eclipse toolbar with various icons for file operations, search, and run.
- Project Explorer:** Shows a Java Application named "Test1 [Java Application]" with a sub-node "Test at localhost:50909". A "Thread [main] (Suspended (breakpoint at line: 4))" node is expanded, showing the stack trace "Test.main(String[]) line: 4".
- Code Editor:** The "Test.java" editor window displays the following code:

```
1 public class Test {  
2     public static void main(String[] args) {  
3         int x, y;  
4         x = 10;  
5         y = 20;  
6         int sum = add(x,y);  
7         if(sum < 10)  
8             System.out.println("Low");  
9         else  
10            System.out.println("High");  
11         int product = multiply(x,y);  
12         System.out.println(product);  
13     }  
14  
15     private static int add(int a, int b) {  
16         System.out.println("inside sum");  
17         return a + b;  
18     }  
19  
20     private static int multiply(int a, int b) {  
21         System.out.println("inside sum");  
22         return a * b;  
23     }  
24 }
```

The line `x = 10;` is highlighted in green, indicating it has not yet been executed.
- Variables View:** Shows the current variable values:

Name	Value
no method rett	
args	String[0] (id=20)
- Console View:** Shows the output of the application so far.
- Bottom Status Bar:** Writable, Smart Insert, 4:1:77

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.

The screenshot shows the Eclipse IDE interface during a Java debugging session. The central window displays the code for `Test.java`:

```
1 public class Test {  
2     public static void main(String[] args) {  
3         int x, y;  
4         x = 10;  
5         y = 20;  
6         int sum = add(x,y);  
7         if(sum < 10)  
8             System.out.println("Low");  
9         else  
10            System.out.println("High");  
11         int product = multiply(x,y);  
12         System.out.println(product);  
13     }  
14  
15     private static int add(int a, int b) {  
16         System.out.println("inside sum");  
17         return a + b;  
18     }  
19  
20     private static int multiply(int a, int b) {  
21         System.out.println("inside sum");  
22         return a * b;  
23     }  
24 }
```

The line `y = 20;` is highlighted in green, indicating it is the current statement being executed. The `args` variable is listed in the Variables view on the right, with its value set to `String[0] (id=20)`. The variable `x` is also listed with its value set to `10`, which is highlighted with a red box.

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.

```
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;
    }

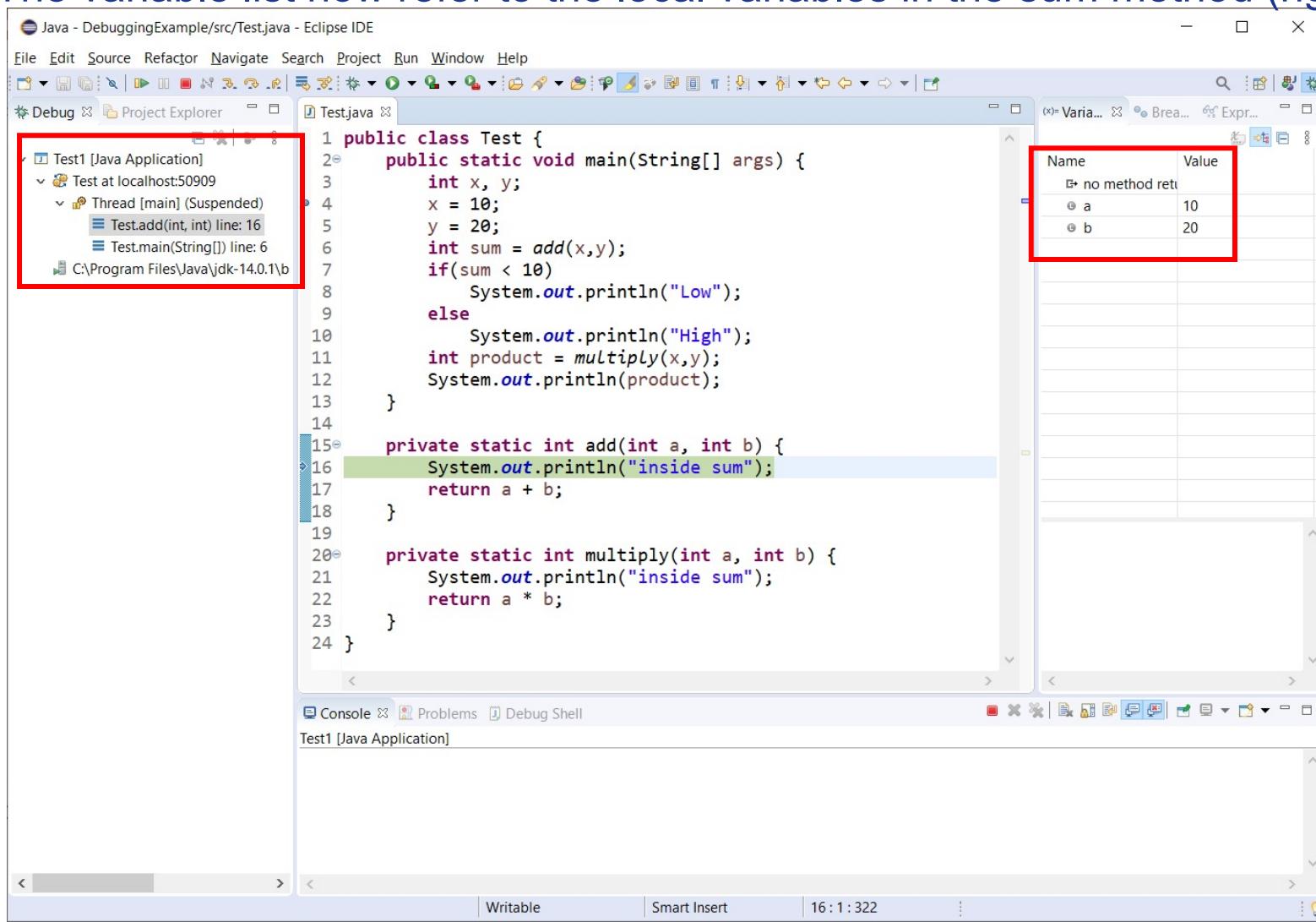
    private static int multiply(int a, int b) {
        System.out.println("inside sum");
        return a * b;
    }
}
```

The screenshot shows the Eclipse IDE interface during a Java debugging session. The left pane displays the Project Explorer with a Java application named 'Test1'. The center pane shows the source code of 'Test.java' with the main method. Line 6 contains the assignment `int sum = add(x,y);`. The variable `y` is highlighted in green. The right pane is the 'Variables' view, which lists the current values of variables: `args` is a `String[0] (id=20)`, `x` is `10`, and `y` is `20`. The value `20` for `y` is highlighted with a red box. The bottom pane shows the 'Console' tab with the message 'Test1 [Java Application]'.

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.

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** Java - java.io.PrintStream - Eclipse IDE
- Menu Bar:** File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help
- Toolbar:** Standard Eclipse toolbar icons.
- Project Explorer:** Shows a Java Application named "Test1" with a thread named "Test at localhost:50909" suspended at the `PrintStream.println()` line.
- Code Editor:** Displays the `PrintStream.println()` method from the `PrintStream.class`. The line `if (getClass() == PrintStream.class)` is highlighted in green, indicating it is currently being executed.
- Variables View:** Shows the current variable values:

Name	Value
no method rett	
this	PrintStream (id=24)
x	"inside sum" (id=31)
- Console View:** Shows the message "Test1 [Java Application]".
- Bottom Status Bar:** Read-Only, Smart Insert, 1027 : 1 : 35621.

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.

The screenshot shows the Eclipse IDE interface during a Java debugging session. The title bar reads "Java - DebuggingExample/src/Test.java - Eclipse IDE". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar has various icons for file operations like Open, Save, Cut, Copy, Paste, Find, and Run. The left sidebar shows the "Project Explorer" with a tree view of "Test1 [Java Application]" containing "Test at localhost:50909" and "Thread [main] (Suspended)". The main editor window displays the code for "Test.java":

```
1 public class Test {
2     public static void main(String[] args) {
3         int x, y;
4         x = 10;
5         y = 20;
6         int sum = add(x,y);
7         if(sum < 10)
8             System.out.println("Low");
9         else
10            System.out.println("High");
11            int product = multiply(x,y);
12            System.out.println(product);
13        }
14
15    private static int add(int a, int b) {
16        System.out.println("inside sum");
17        return a + b;
18    }
19
20    private static int multiply(int a, int b) {
21        System.out.println("inside sum");
22        return a * b;
23    }
24 }
```

The line `return a + b;` is highlighted with a green background. The bottom-left corner of the editor shows the status bar with "Writable" and "Smart Insert". The bottom-right corner shows the status bar with "17 : 1 : 359". To the right of the editor, the "Variables" view shows:

Name	Value
println() return	(No explicit return ...)
a	10
b	20

The "Console" view at the bottom shows the output: "inside sum".

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()).

The image shows two side-by-side screenshots of the Eclipse IDE interface, illustrating the state of a Java application during debugging. A large blue arrow points from the left window to the right window, indicating the progression of the program's execution.

Left Window (Initial State):

- Project Explorer:** Shows the project structure with 'Test1 [Java Application]' and its source files.
- Code Editor:** Displays the code for `PrintStream.class`. The variable `sum` is highlighted in green at line 6. The code includes `add` and `multiply` methods and prints "inside sum" to the console.
- Variables View:** Shows the current values of variables: `args` (String[0] id=20), `x` (10), and `y` (20).
- Console View:** Shows the output: `inside sum`.

Right Window (After Step Over):

- Project Explorer:** Same as the left window.
- Code Editor:** The code has progressed to line 7, where the `if` condition is checked. The variable `sum` is highlighted in green at line 6.
- Variables View:** Shows updated variable values: `args` (String[0] id=20), `x` (10), `y` (20), and `sum` (30).
- Console View:** Shows the output: `inside sum`.

Java - DebuggingExample/src/Test.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Project Explorer

Test1 [Java Application]

Test at localhost:50954

Thread [main] (Suspended)

Test.main(String[]) line: 10

C:\Program Files\Java\jdk-14.0.1\bin

```

1 public class Test {
2     public static void main(String[] args) {
3         int x, y;
4         x = 10;
5         y = 20;
6         int sum = add(x,y);
7         if(sum < 10)
8             System.out.println("Low");
9         else
10            System.out.println("High");
11         int product = multiply(x,y);
12         System.out.println(product);
13     }
14
15     private static int add(int a, int b) {
16         System.out.println("inside sum");
17         return a + b;
18     }
19
20     private static int multiply(int a, int b) {
21         System.out.println("inside sum");
22         return a * b;
23     }
24 }
```

Console Problems Debug Shell

Test1 [Java Application]

inside sum

Writable Smart Insert 10:1:178

Java - DebuggingExample/src/Test.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Project Explorer

Test1 [Java Application]

Test at localhost:50954

Thread [main] (Suspended)

Test.main(String[]) line: 11

C:\Program Files\Java\jdk-14.0.1\bin

```

1 public class Test {
2     public static void main(String[] args) {
3         int x, y;
4         x = 10;
5         y = 20;
6         int sum = add(x,y);
7         if(sum < 10)
8             System.out.println("Low");
9         else
10            System.out.println("High");
11         int product = multiply(x,y);
12         System.out.println(product);
13     }
14
15     private static int add(int a, int b) {
16         System.out.println("inside sum");
17         return a + b;
18     }
19
20     private static int multiply(int a, int b) {
21         System.out.println("inside sum");
22         return a * b;
23     }
24 }
```

Console Problems Debug Shell

Test1 [Java Application]

inside sum

High

Writable Smart Insert 11:1:210

Java - DebuggingExample/src/Test.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Project Explorer

Test1 [Java Application]

Test at localhost:50954

Thread [main] (Suspended)

Test.main(String[]) line: 12

C:\Program Files\Java\jdk-14.0.1\bin

```

1 public class Test {
2     public static void main(String[] args) {
3         int x, y;
4         x = 10;
5         y = 20;
6         int sum = add(x,y);
7         if(sum < 10)
8             System.out.println("Low");
9         else
10            System.out.println("High");
11         int product = multiply(x,y);
12         System.out.println(product);
13     }
14
15     private static int add(int a, int b) {
16         System.out.println("inside sum");
17         return a + b;
18     }
19
20     private static int multiply(int a, int b) {
21         System.out.println("inside sum");
22         return a * b;
23     }
24 }
```

Console Problems Debug Shell

Test1 [Java Application]

inside sum

High

Writable Smart Insert 12:1:242

Java - DebuggingExample/src/Test.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug Project Explorer

Test1 [Java Application]

Test at localhost:50954

Thread [main] (Suspended)

Test.main(String[]) line: 13

C:\Program Files\Java\jdk-14.0.1\bin

```

1 public class Test {
2     public static void main(String[] args) {
3         int x, y;
4         x = 10;
5         y = 20;
6         int sum = add(x,y);
7         if(sum < 10)
8             System.out.println("Low");
9         else
10            System.out.println("High");
11         int product = multiply(x,y);
12         System.out.println(product);
13     }
14
15     private static int add(int a, int b) {
16         System.out.println("inside sum");
17         return a + b;
18     }
19
20     private static int multiply(int a, int b) {
21         System.out.println("inside sum");
22         return a * b;
23     }
24 }
```

Console Problems Debug Shell

Test1 [Java Application]

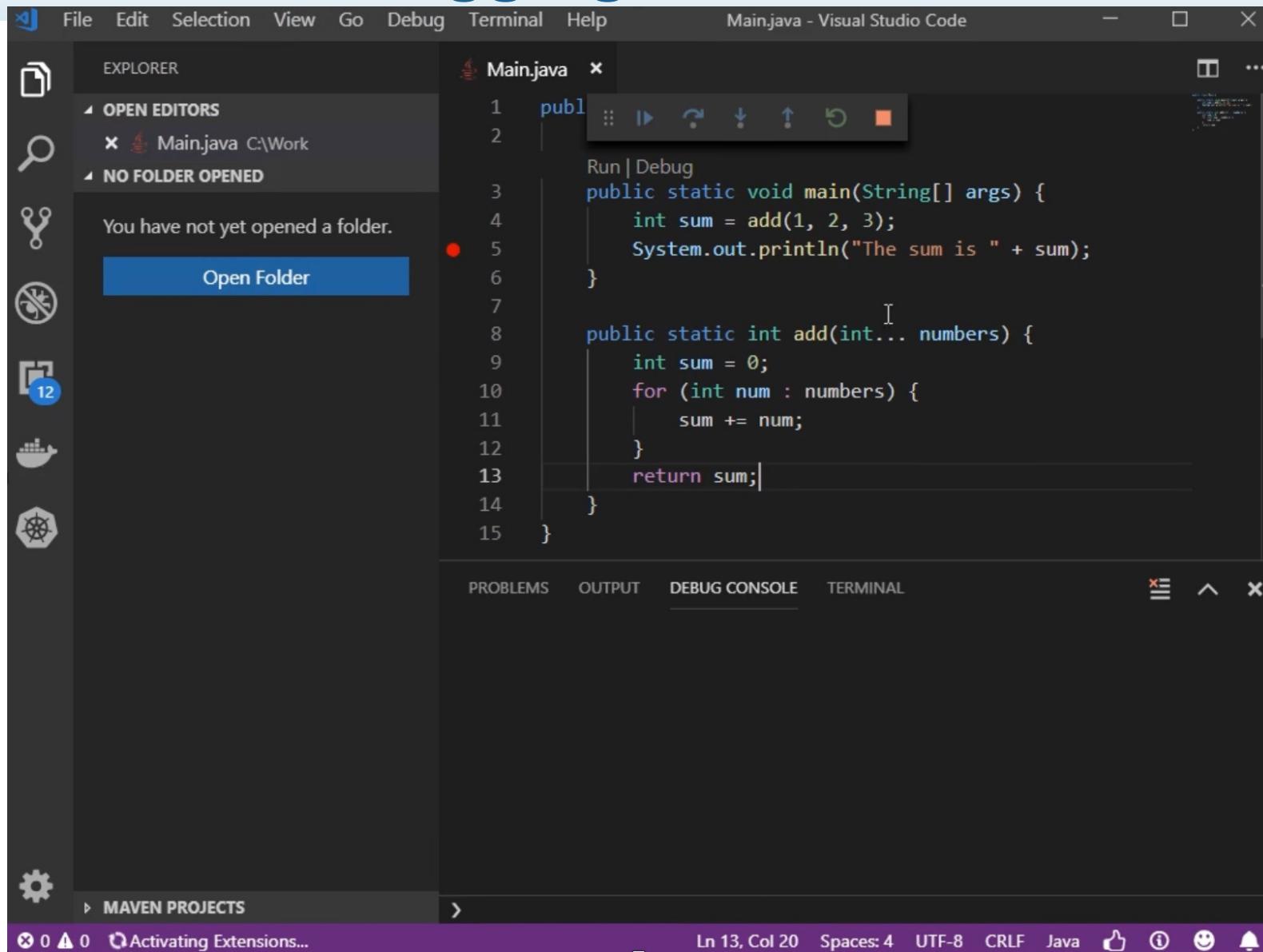
High

inside sum

200

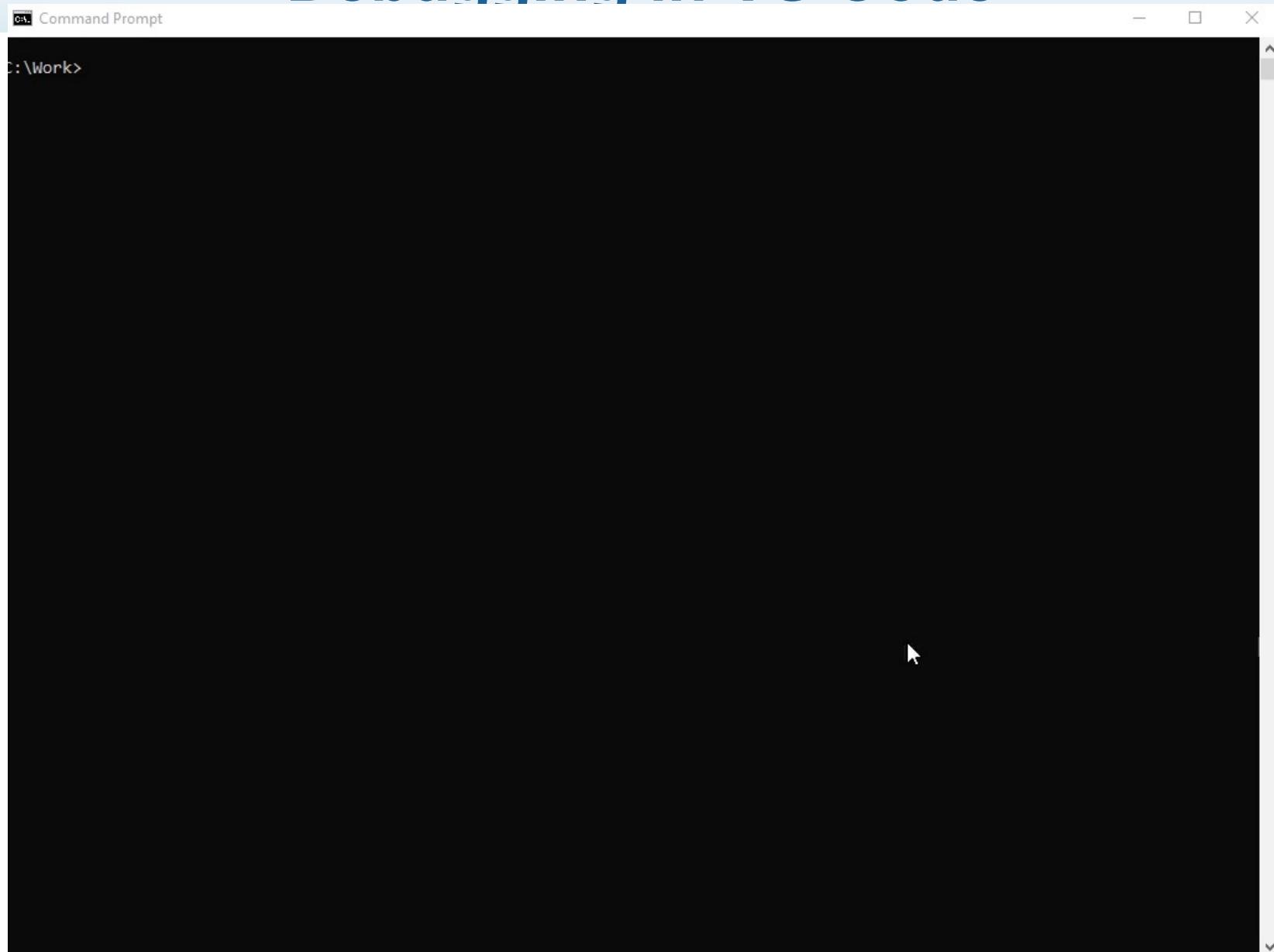
Writable Smart Insert 13:1:274

Debugging in VS Code



[Source](#): VS Code Docs

Debugging in VS Code



[Source](#): VS Code Docs