INDEX

Code Debugging ----------

Code Debugging

What is Debugging

- According to Wikipedia: Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected."
- Bug = problem/ defect/ mistake in the app/ program/ project. (Some abnormality in the application execution).
- Debug = De + Bug (Rectifying the bug by identifying and analyzing the bug).
- The tool or program or software that can be used for debugging is called debugger/ debugging Tool.
- Debugging is the process getting step by step by execution application to find out problems/ bugs and to fix them as needed.
- To debug Java code, we have two tools
    a. JDB (supplied by JDK along with JDK installation)
        - Available in <JAVA_HOME>\bin directory as jdb.exe.
        - It is CUI tool and not so popular (not industry standard).

    b. IDE supplied Debugger
        - Eclipse/ IntelliJ/ RAD and etc. IDES supplied debuggers.
        - These GUI and user-friendly and industry standard.

Note:

- ✓ Unit Testing: Programmer's testing owns his piece of code unit testing. We are using JUnit tool for this.
- ✓ Logging: Keep tracking application flow by generating log messages. E.g. SOPL/ SEPL messages or Log4j messages.

Q. Where debugging is required in the company?

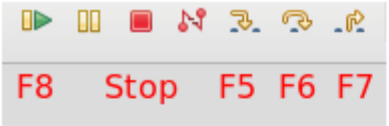
Ans. Debugging is required in the company for the following reasons

- To find, analyze and fix logical problems and errors.
- If unit testing is failed then also, we use debugging to fix the problems (run as JUnit in debugging mode).
- After joining in the company, that to in existing project the programmers perform debugging process to know the code flow of the project.
- If the project is in production the bugs received will be fixed only after seeing log files (log messages) and after performing debugging.
  and etc.

Note: In company environment we use Debug As more and more instead Run as.

Debugging in Eclipse IDE

a. Run the apps in debug as mode
b. Change eclipse mode debug perspective
c. Use various debugging operations/ shortcuts to see the code flow to find out the problem and to fix the problem.



Shortcut	Toolbar	Description
F5 (Step Into)		Steps into the call
F6 (Step Over)		Steps over the call
F7 (Step Return)		Steps out to the caller
F8 (Resume)		Resumes the execution
Ctrl + R (Run to Line)		Run to the line number of the current caret position
Drop to Frame		Rerun a part of your program
Shift + F5 (Use Step Filters)		Skipping the packages for Step into
Ctr + F5 / Ctrl + Alt + Click		Step Into Selection

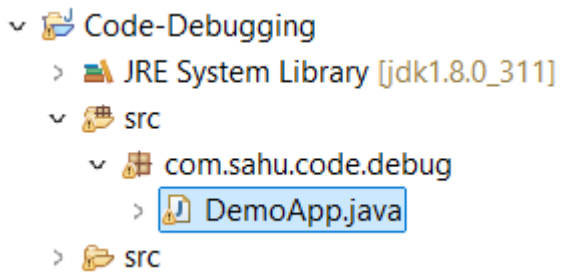
Breakpoint:

- It the point in the app/ project/ code/ program from where debugging process should start. Till the breakpoint code executes normally. From break point control comes to programmer to see the code flow as needed.
- There are two types of break points
  - Method break point: Given at method name or definition of method.
  - Line break point: given inside the method definition where logics are available (popular).

Shortcut	Description
F3 (Step into)	Step into the call (Get into method definition)
F6 (Step Over)	Step over the call (Completed the current line execution without getting into details)
F7 (Step return)	Step out of the caller (Returns the control to caller method from current method definition after executing the code)
F8 (Resume)	Resume the execution (Jumps the control from one break point to next breakpoint in the flow if the current break point is last break point, then it terminates the app by reaching to end of the application)
CTRL + R (Run to line)	Run to the line number of the current cursor position
Shift + F5 (Use Step Filter)	Skipping the packages for step into
CTRL + F5/ CTRL + ALT + CLICK	Step into selection (Transfers the control into select code details/ select code method definition)
CTRL + Shift + B	To toggle breakpoint (to enable or disable method/ line breakpoint)

Note: Eclipse official blog For Debugging concept [Link].

Directory Structure of Code-Debugging:



- Develop the above directory Structure using normal Java Project option and create the package and class.
- Then place the following code with in their respective files.

DemoApp.java

```
package com.sahu.code.debug;

import java.util.Scanner;
```

```java
public class DemoApp {

	private void sayHello(String userName) {
		System.out.println("DemoApp.sayHello()");
		for (int i = 0; i <= 10; i++) {
			System.out.println(userName + " " + i);
		}
		System.out.println("End of sayHello()");
		Scanner scanner = new Scanner(System.in);
		System.out.print("Enter a message: ");
		String message = scanner.next();

		displayMesssage(message);
	}

	private void displayMesssage(String message) {
		System.out.println("DemoApp.displayMesssage() " + message);
		System.out.println("DemoApp.displayMesssage() " + message);
		printNumber(10);
	}

	private void printNumber(int num) {
		System.out.println("DemoApp.printNumber()");
		for (int j = 0; j < num; j++) {
			System.out.println(j);
		}
		System.out.println("End of DemoApp.printNumber()");
	}

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

	private int sub(int a, int b) {
		System.out.println("DemoApp.add()");
		return a - b;
	}

	public static void main(String[] args) {
```

```
            DemoApp demoApp = new DemoApp();
            demoApp.sayHello("Raja");
            demoApp.add(10, 20);
            demoApp.sub(20, 30);
        }

}
```

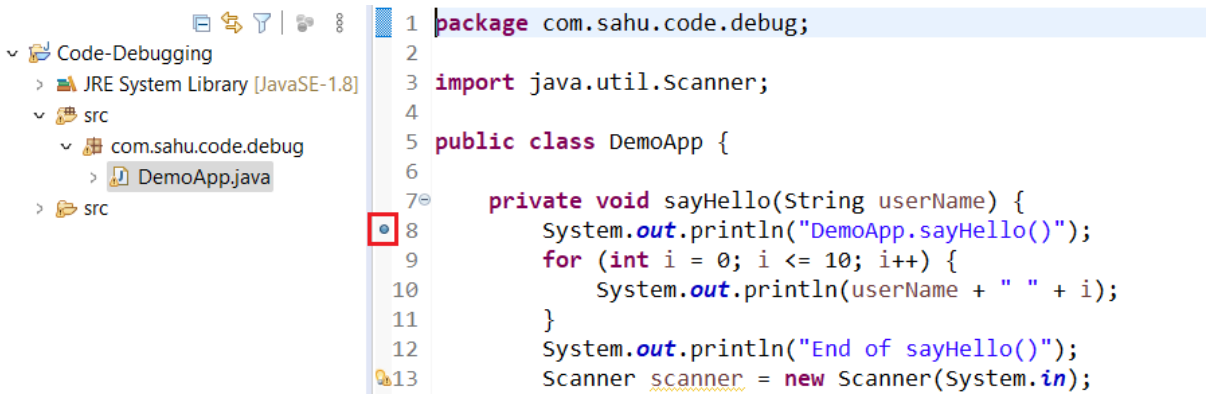
Process of Debugging

Step 1: Ready the above code or your own code in your Eclipse

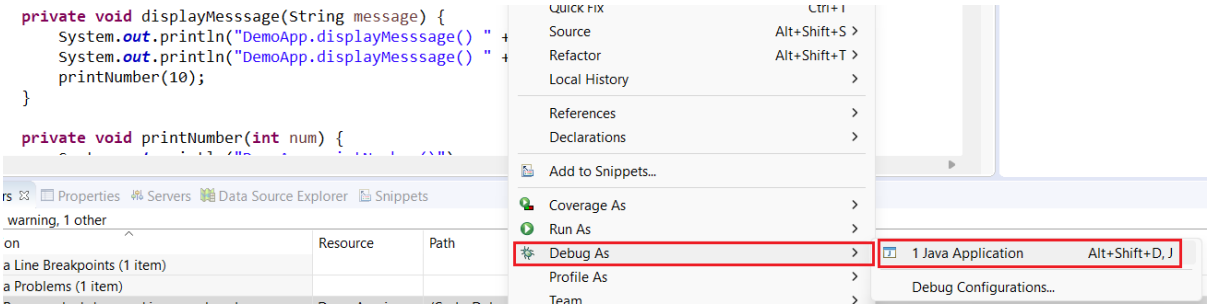
Step 2: Keep a breakpoint at line number 8 (or where you want in your code) for a that before the number there is a white space just once double click there.

Once double click: breakpoint will come
Again, double click: breakpoint will remove

Or keep the cursor on that line and click on **CTRL + Shift + B** to add breakpoint and again for removing



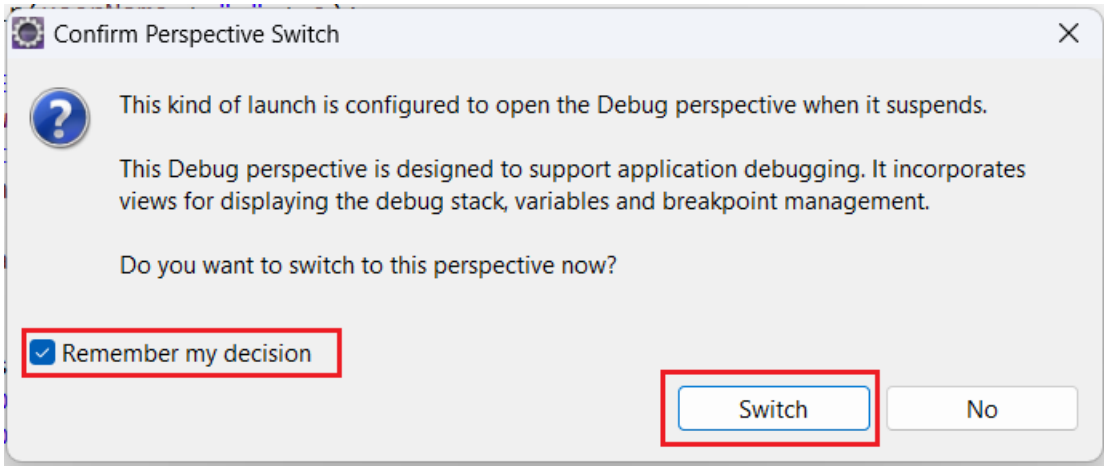
Step 3: Right click on the code or on the file then choose **Debug As** and click on **1 Java** Application option.



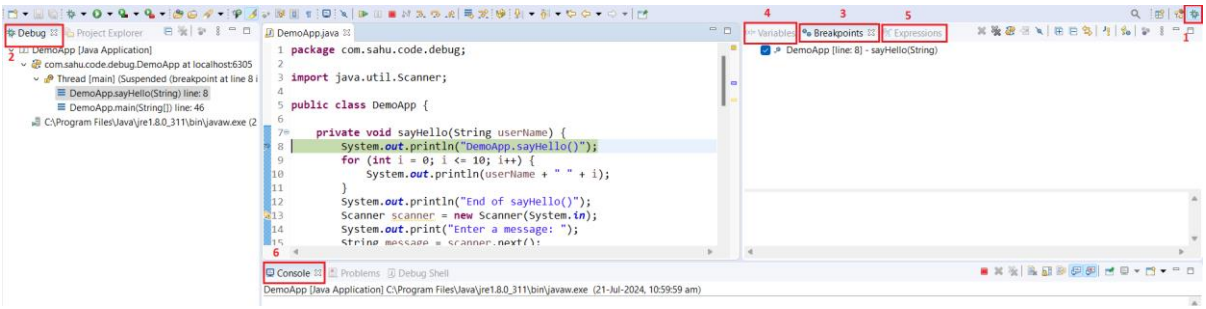
Note:

- ✓ Though it is a **standalone application** that's why we are using **Debug As – Java application**.
- ✓ If we have a **web application** then we have to use **Debug As – Debug on Server**.
- ✓ If we have a **Spring Boot application** (standalone or web) then we have to use to **Debug As – Spring Boot App**.

Step 4: For first time only, it will ask for switch to Debug perspective. So, click on the **Remember my decision** check box then click on **Switch** button.



Note: Now our Eclipse in change the Java perspective to Debug perspective. The debug perspective offers additional views that can be used to troubleshoot an application like Breakpoints, Variables, Debug, Console etc. When a Java program is started in the debug mode, users are prompted to switch to the debug perspective.



1. Debug view: Visualizes call stack and provides operations on that.
2. Breakpoints view: Shows all the breakpoints.
3. Variables view: Shows the declared variables and their values. Press CTRL + Shift + D or CTRL + Shift + I on a selected variable to show its value.

4. Expression view: Shows the expression and their values. Press CTRL + Shift + D or CTRL + Shift + I on a selected expression to show its value. You can also add a permanent watch on an expression/ variable that will then be shown in the Expressions view when debugging is on.
5. Console view: Program output is shown here.

Step 5: To go to next line, we have to use **Step Over** shortcut or **F6**.

```
DemoApp.java
1  package com.sahu.code.debug;
2
3  import java.util.Scanner;
4
5  public class DemoApp {
6
7      private void sayHello(String userName) {
8          System.out.println("DemoApp.sayHello()");
9          for (int i = 0; i <= 10; i++) {
10             System.out.println(userName + " " + i);
11         }
```

Step 6: Click F6 for next line, we can click until it reaches to line number 15, because there is a scanner input require that will block the control until we not enter the value in console.

```
DemoApp.java
10             System.out.println(userName + " " + i);
11         }
12         System.out.println("End of sayHello()");
13         Scanner scanner = new Scanner(System.in);
14         System.out.print("Enter a message: ");
15         String message = scanner.next();
16

Console   Problems   Debug Shell
DemoApp [Java Application] C:\Program Files\Java\jre1.8.0_311\bin\javaw.exe  (21-Jul-2024, 10:59:59 am)
DemoApp.sayHello()
Raja 0
Raja 1
Raja 2
Raja 3
Raja 4
Raja 5
Raja 6
Raja 7
Raja 8
Raja 9
Raja 10
End of sayHello()
Enter a message:
```

Step 7: After enter the value it will go to the next line, if we click on F6 then the entire method will execute directly and give the output, but we want get into

the details then we have to use **Step Into** shortcut or **F5**.

```
DemoApp.java
10                  System.out.println(userName + " " + i);
11              }
12              System.out.println("End of sayHello()");
13              Scanner scanner = new Scanner(System.in);
14              System.out.print("Enter a message: ");
15              String message = scanner.next();
16
17              displayMesssage(message);
18          }
```

Step 8: Now it came inside the display method.

```
DemoApp.java
17              displayMesssage(message);
18          }
19
20          private void displayMesssage(String message) {
21              System.out.println("DemoApp.displayMesssage() " + message);
22              System.out.println("DemoApp.displayMesssage() " + message);
23              printNumber(10);
24          }
```

Step 9: If you click on the F5 again then it will try to get into the details of println() method. If you again click on F5 it goes into deeper and take lots of time to come out. So, for that we have to use **Step Return** shortcut or **F7**.

```
DemoApp.java    StringBuilder.class
103
104     /**
105      * Constructs a string builder initialized to the contents of the
106      * specified string. The initial capacity of the string builder is
107      * {@code 16} plus the length of the string argument.
108      *
109      * @param   str   the initial contents of the buffer.
110      */
111     public StringBuilder(String str) {
112         super(str.length() + 16);
113         append(str);
114     }
```

Step 10: Now you will back to your method and you clicked Step over or F6 for next line but you want to go to the beginning of the method then we have use **Drop to Frame** shortcut.

```
DemoApp.java   StringBuilder.class
17          displayMesssage(message);
18      }
19
20      private void displayMesssage(String message) {
21          System.out.println("DemoApp.displayMesssage() " + message);
22          System.out.println("DemoApp.displayMesssage() " + message);
23          printNumber(10);
24      }
```

You can see the cursor move to the first line.

```
20      private void displayMesssage(String message) {
21          System.out.println("DemoApp.displayMesssage() " + message);
22          System.out.println("DemoApp.displayMesssage() " + message);
23          printNumber(10);
24      }
```

Step 11: Add one more breakpoint in display method first line.

```
20      private void displayMesssage(String message) {
21          System.out.println("DemoApp.displayMesssage() " + message);
22          System.out.println("DemoApp.displayMesssage() " + message);
23          printNumber(10);
24      }
```

Step 12: Suppose from the first endpoint to we want to go to next breakpoint then we have to use **Resume** shortcut or **F8**.

```
DemoApp.java   StringBuilder.class
6
7       private void sayHello(String userName) {
8           System.out.println("DemoApp.sayHello()");
9           for (int i = 0; i <= 10; i++) {
10              System.out.println(userName + " " + i);
11          }
```

Step 13: Though we are having a scanner input in the first method, so for that it will wait until we not give the value once we give the value and hit enter it will direct go to next breakpoint i.e. display method 1st line.

```
20      private void displayMesssage(String message) {
21          System.out.println("DemoApp.displayMesssage() " + message);
22          System.out.println("DemoApp.displayMesssage() " + message);
```

Step 14: If we want go from current debug cursor to the mouse cursor by skipping the middle things. For that we have to first click where we want to move then do **Right click** and after that click on **Run to Line** or use **CTRL + R**.

```
26    private void printNumber(int num) {
27        System.out.println("DemoApp.printNumber()");
28        for (int j = 0; j < num; j++) {
29            System.out.println(j);
30        }
31        System.out.println("End of DemoApp.printNumber()")
32    }
33
34    private int add(int a, int b) {
35        System.out.println("DemoApp.add()");
```

debug cursor

mouse cursor

Console | Problems | Debug Shell

```
DemoApp [Java Application] C:\Program Files\Java\jdk1.8.0_311\bin\javaw.exe  (21-Jul-2024,
Raja 2
Raja 3
Raja 4
Raja 5
Raja 6
Raja 7
Raja 8
Raja 9
Raja 10
End of sayHello()
Enter a message: hyd
```

Copy	Ctrl+C
Copy Qualified Name	
Paste	Ctrl+V
Quick Fix	Ctrl+1
Source	Alt+Shift+S >
Refactor	Alt+Shift+T >
Local History	>
References	>
Declarations	>
Add to Snippets...	
All Instances...	Ctrl+Shift+N
Instance Count...	
Force Return	Alt+Shift+F
Watch	
Inspect	Ctrl+Shift+I
Display	Ctrl+Shift+D
Execute	Ctrl+U
Run to Line	Ctrl+R
Step Into Selection	Ctrl+F5

Step 15: From the current position, where other line is selected, we want to transfer the control onto selected code details or selected code method definition then **Right click** and after that click on **Step Into** selection or use **CTRL + F5**.

```
26    private void printNumber(int num) {
27        System.out.println("DemoApp.printNumber()");
28        for (int j = 0; j < num; j++) {
29            System.out.println(j);
30        }
31        System.out.println("End of DemoApp.printNumber()");
32    }
33
34    private int add(int a, int b) {
35        System.out.println("DemoApp.add()");
```

Console | Problems | Debug Shell

```
DemoApp [Java Application] C:\Program Files\Java\jdk1.8.0_311\bin\javaw.exe  (21-Jul-2024, 1:06:29 pm)
Raja 2
Raja 3
Raja 4
Raja 5
Raja 6
Raja 7
Raja 8
Raja 9
Raja 10
End of sayHello()
Enter a message: hyd
DemoApp.displayMesssage() hyd
DemoApp.displayMesssage() hyd
```

Copy	Ctrl+C
Copy Qualified Name	
Paste	Ctrl+V
Quick Fix	Ctrl+1
Source	Alt+Shift+S >
Refactor	Alt+Shift+T >
Surround With	Alt+Shift+Z >
Local History	>
References	>
Declarations	>
Add to Snippets...	
All Instances...	Ctrl+Shift+N
Instance Count...	
Force Return	Alt+Shift+F
Watch	
Inspect	Ctrl+Shift+I
Display	Ctrl+Shift+D
Execute	Ctrl+U
Run to Line	Ctrl+R
Step Into Selection	Ctrl+F5
Coverage As	>

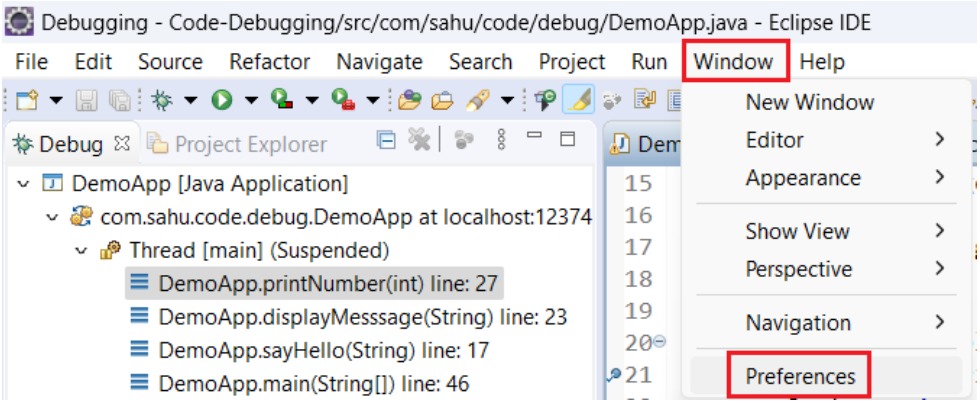
Step 16: After all the operation like Step into, step over, step return, drop to frame and etc. when we will reach to the end of the application it will

terminate automatically.

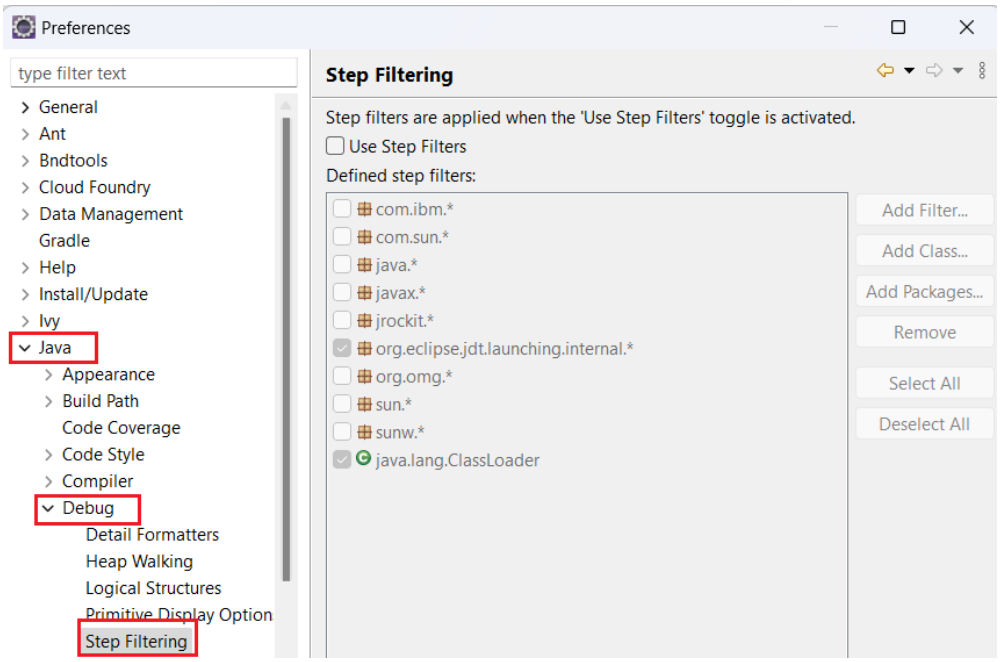
Working with Step Filter

- Allows Certain packages not to participate in debugging especially useful if you are not interested to get into the details of pre-defined classes and their methods like SOPLN, scanner.next() and etc.

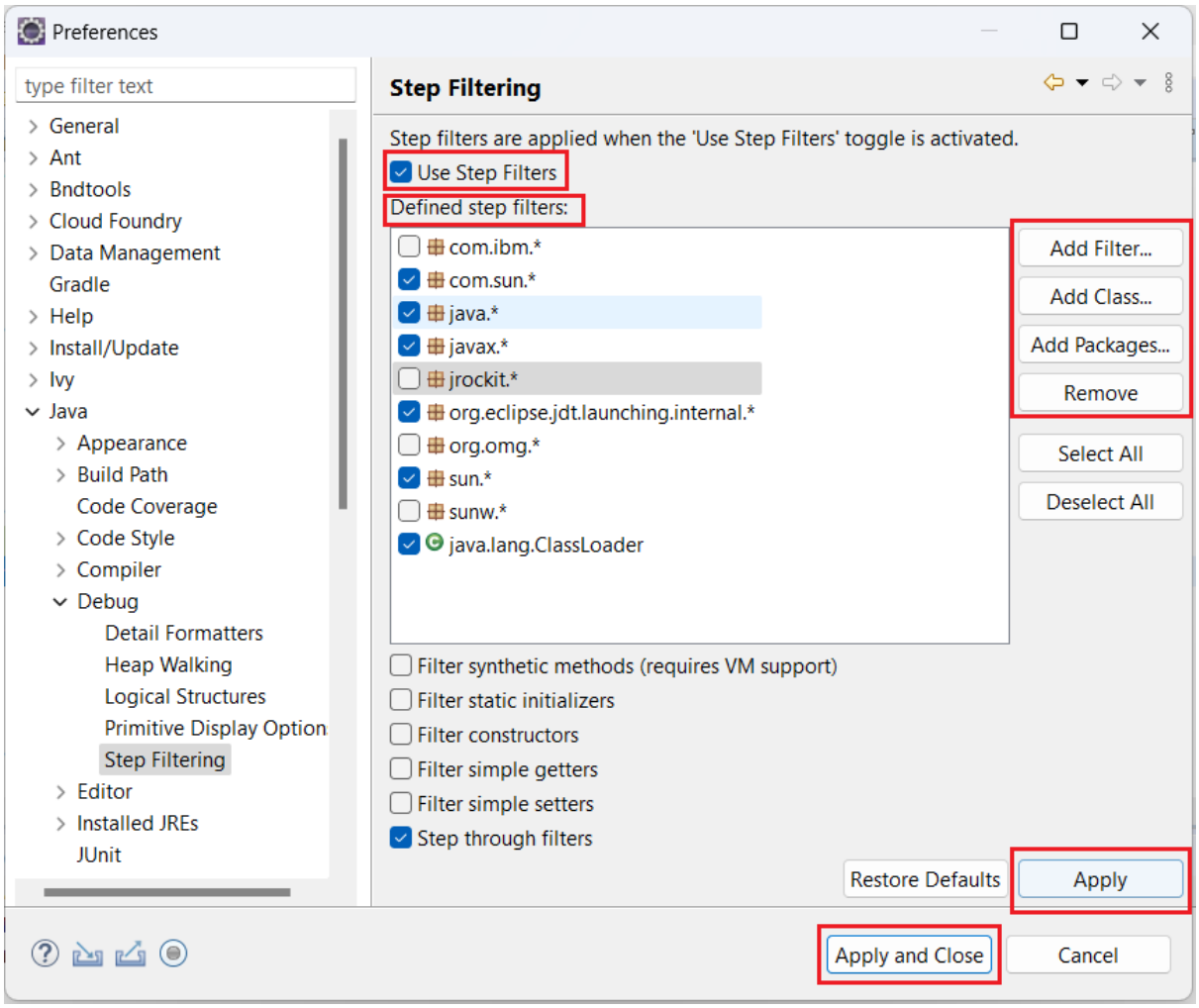
Step 1: Click on the **Window** tab then click on the **Preferences** option



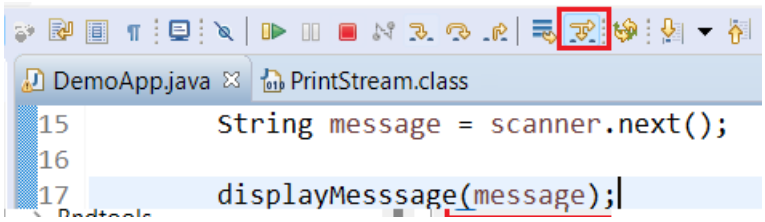
Step 2: The go to **Java** expands it then expand the **Debug** then click on **Step Filtering** option.



Step 3: Check the **User Step Filters** option then choose the packages that you want to filter under **Defined step filters** or by using **Add Filter**, **Add Class**, **Add Packages** options also you can use and after that click on **Apply** button then click on **Apply and Close** button.



Step 4: Then make sure step filter button is in active mode for that we have to use **Step Filter** shortcut or **Shift + F5**.



Step 5: Now Debug the Java application and press F5 button for classes and methods of above packages but it will not step into method definitions.

Note: Make sure you have to add JDK into the project build path otherwise if

we have JRE in build path then the Step into the predefine classes feature will not be working so there is no use of Step Filter option.

Variable & Expression View

Step 1: By keeping mouse over variable or parameter of the debugging mode method definition we can watch value changes that are taking the place in the application execution.

```
26⊖     private void printNumber(int num) {
27          System.out.println("DemoA
28          for (int j = 0; j < num;
29              System.out.println(j)
```

num= 10

Even we can see in Variable view tab

Variables | Expressions | Breakpoints

Name	Value
no method return value	
this	DemoApp (id=16)
num	10
j	0

Step 2: While execute we can change the value and we can see the result for this after change the value we have to click **Step Over** or **F6**.

Variables | Expressions | Breakpoints

Name	Value
println() returned	(No explicit return value)
this	DemoApp (id=16)
num	10
j	9

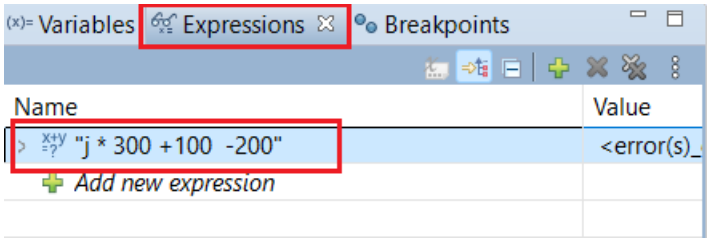
Now you can see after 0 directly it is printing 9.

Console | Problems | Debug Shell

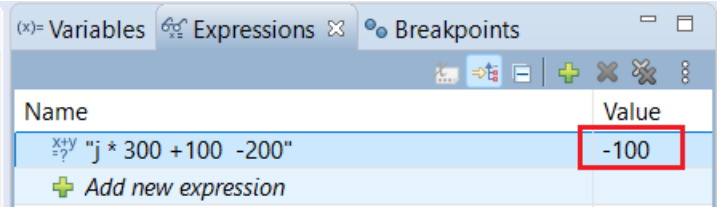
DemoApp [Java Application] C:\Program Files\Java\jdk1.8.0_311\bir

```
DemoApp.printNumber()
0
9
```

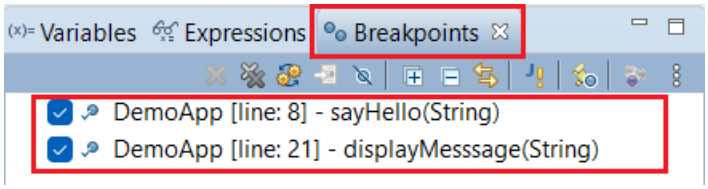
Step 3: In Expressions view tab we can apply the expressions also.



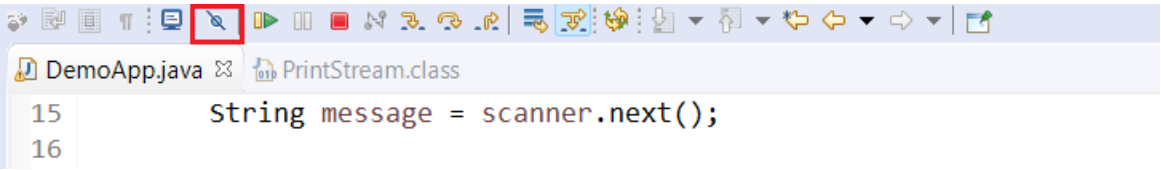
While we are executing according to the "J" value change the expression value will also change.



Step 4: Breakpoint view tab will give list of breakpoints we can enable of disable by clicking on checkbox.



Step 5: If we want to disable all the breakpoints then we have to use **Skill All Breakpoints** shortcut or **CTRL + ALT + B**;



Note:

- ✓ We can use variables window, expression window to see the values changes.
- ✓ Expression window is useful to given current inputs for another formula of business logic and to check the results.
- ✓ Breakpoints window gives list of breakpoints. Show we can enable or disable them as need.
- ✓ We can add new breakpoints in the debugging mode of the app itself.

Standalone layered application components

DS

Client App → Controller class → Service class → DAO class → DB s/w

- We can add break points not only programmer developed user-defined classes and also in pre-defined classes that are internally used by our user-defined classes. like in HttpServlet, GenericServlet, String, System, DispatcherServlet, ClassPathXmlApplicationContext and etc.
- While working maven/ gradle build tools we get built-in java decompiles i.e. there is no need arranging decompiles separately.
- To see source code, we have to use **F3**.
- To open details of current class/ interface/ enum/ etc. including hierarchy we have to use **F4**.
- In the middle of debugging, we can add/ remove new break points dynamically.
- So, we can use all the shortcut and all the feature of debugging here so, we can take a Spring or Spring Boot standalone layered application and we can go ahead.

Debugging on Web application

- Also known as server level debugging.
- Here instead of **Run As – Run on Server** we have to **use Debug As – Debug on Server**.
- For Spring Boot Web application, we have to use **Debug As – Spring Boot App**.
- And rest of thing as like previous all shortcut and all we have use like above.

Debug in IntelliJ IDE

- Let's use the following sample code to demonstrate.

DemoApp.java

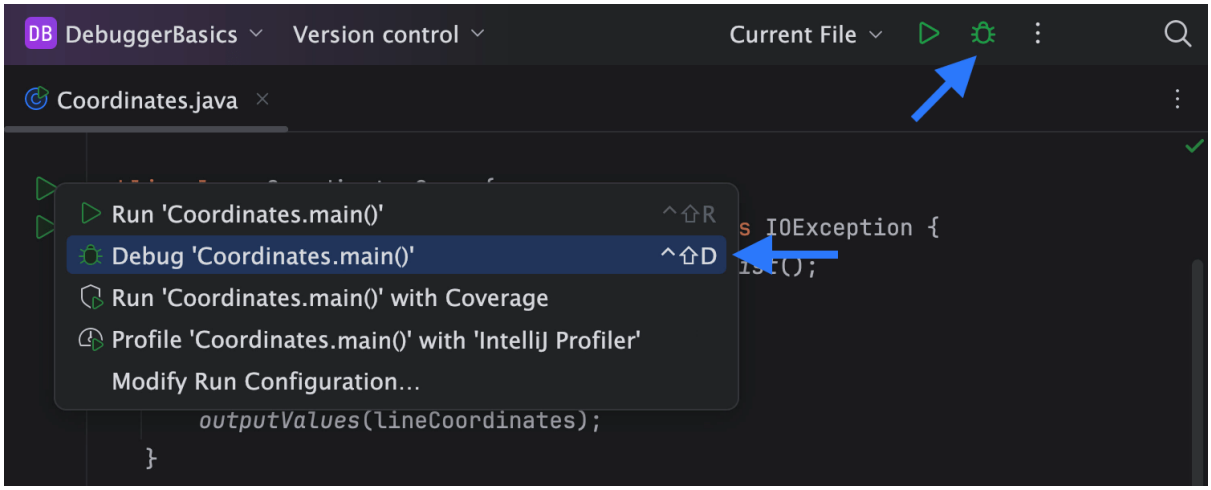
```
package com.jetbrains;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
```

```
public class Coordinates {

  public static void main(String[] args) throws IOException {
     List<Point> lineCoordinates = createCoordinateList();
     outputValues(lineCoordinates);
     Point p = new Point(13, 30);
     removeValue(lineCoordinates, p);
     outputValues(lineCoordinates);
  }

  private static void outputValues(List<Point> lineCoordinates) {
     System.out.println("Output values...");
     for (Point p : lineCoordinates) {
        System.out.println(p);
     }
  }

  private static void removeValue(List lineCoordinates, Point p) {
     lineCoordinates.remove(p);
  }

  private static List createCoordinateList() {
     ArrayList list = new ArrayList<>();
     list.add(new Point(12, 20));
     list.add(new Point(13, 30));
     return list;
  }

}
```

There are a number of ways to start the debugger:

a. Click on the **Run** icon in the gutter area and select the **Debug** option.
b. Invoke context actions on the class or main method by using **Alt + Enter** (Windows/Linux) or ⌥⏎ (macOS) and choose the **Debug** action.
c. Launch from the **Run** menu.
d. Simply press **Shift + F9** (Windows/Linux) or **^(⇧)D** (macOS).

Note: Referred Blog: [Link]

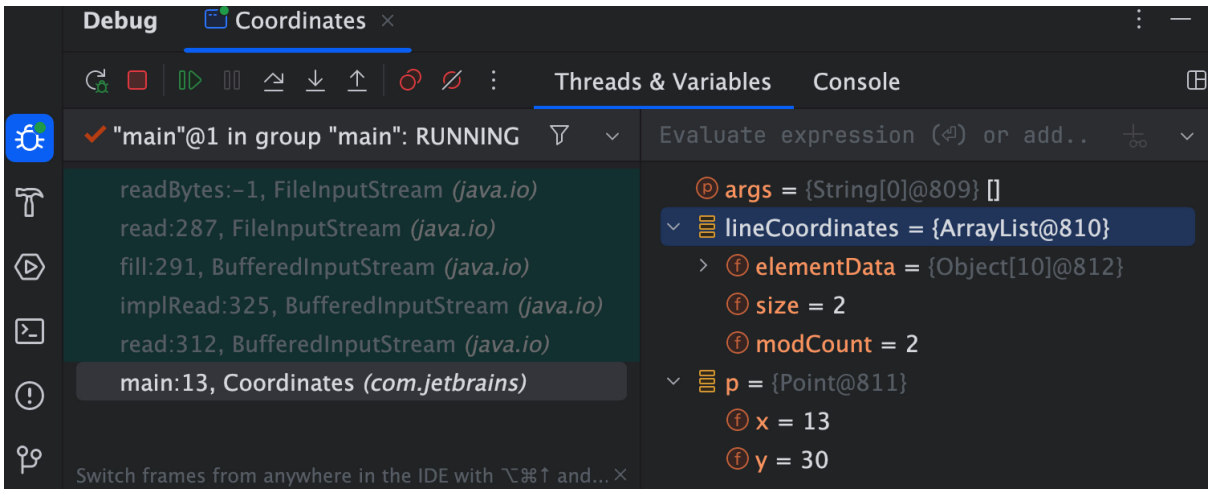


Overview of the sample application

- The code used in this blog is simple. The method **createCoOrdinateList()** creates two instances of the **Point** class and adds them to an **ArrayList**. The Point class has two fields, **x** and **y**, and getter and setter methods. The **outputValues()** method outputs the past list items to the console. The next line of code creates a **Point** instance and the **removeValue()** method tries to remove it from the **lineCoordinates** list.
- When you execute this code, you'll see in the output that even though a **Point** with **x** and **y** values **13** and **30** were added to the list, when another instance with identical values was created to remove it, it was not successful. Let's debug the code.
- To debug your code, you'll need to know the various step actions you can use to move through your code to find the bugs.

Debug Window

- The debug window displays important information when your application suspends execution on a breakpoint, like frames, threads, console window, step action icons, variables pane, and much more.



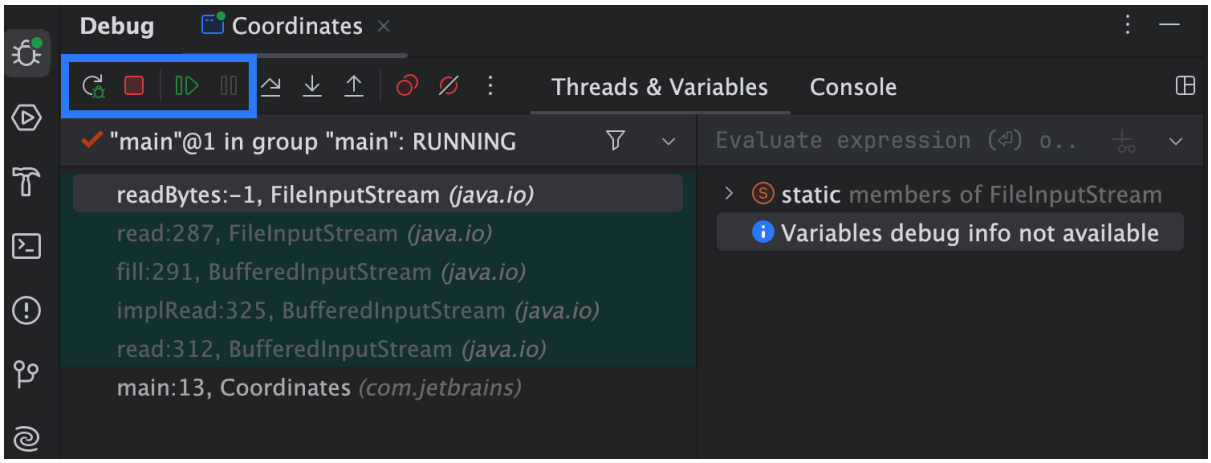
- If you close the **Debug** Window by mistake, you can always reopen it using the shortcut **Alt + 5** (Windows/Linux) or **⌘5** (macOS).

Pause, resume, restart, or stop the debugger

- If your application seems to be unresponsive, you can pause the program to analyze where your code is stuck. Let's modify the main method from the preceding section as follows:

```
public static void main(String[] args) throws IOException {
    List<Point> lineCoordinates = createCoordinateList();
    outputValues(lineCoordinates);
    Point p = new Point(13, 30);
    int y = System.in.read();        // execution pauses here
    removeValue(lineCoordinates, p);
    outputValues(lineCoordinates);
}
```

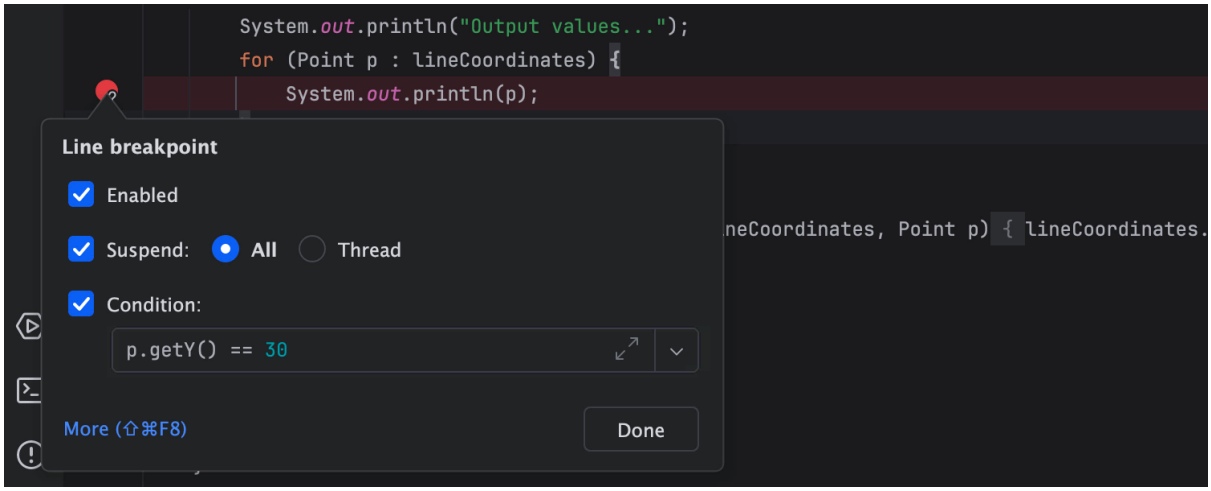
- Execute your application in debug mode. If you don't input a value, your application would seem to become unresponsive.
- In the Debug window, click on **Pause Program** and the editor window will show the class and method your application is currently executing – or blocked on. In this example, you can see that the code is blocked for user input, showing the relevant class and method in the editor. You can also view the call stack. By clicking on the method calls in the call stack, you can view the corresponding class and method in the editor window.
- You can resume program execution by clicking on **Resume Program** or by using the shortcut **F9**. To restart the program in debug mode, select **Rerun**. You can stop debugging your program at any time by using the **Stop** icon.



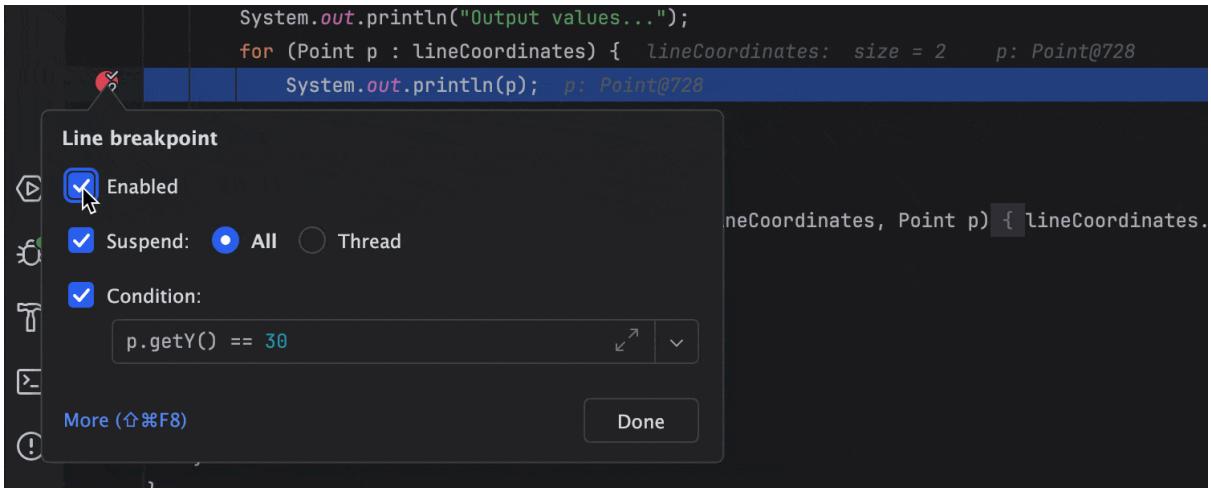
Note: Notice that I didn't set any breakpoints in this case.

Breakpoints

- A breakpoint will stop the execution of your program, so that you can analyze the state of your code.
- To set a breakpoint on a line of code, click in the gutter area or use the shortcut **CTRL + F8** (Windows/L inux) or **⌘F8** (macOS). If you don't want to stop execution every time it reaches a breakpoint, you can define a condition for the breakpoint. For example, let's add a breakpoint in the method **outputValues()**, on the line of code that outputs the value of variable **p** and define a condition to stop code execution when the field **y** of reference variable **p** is equal to **30**.



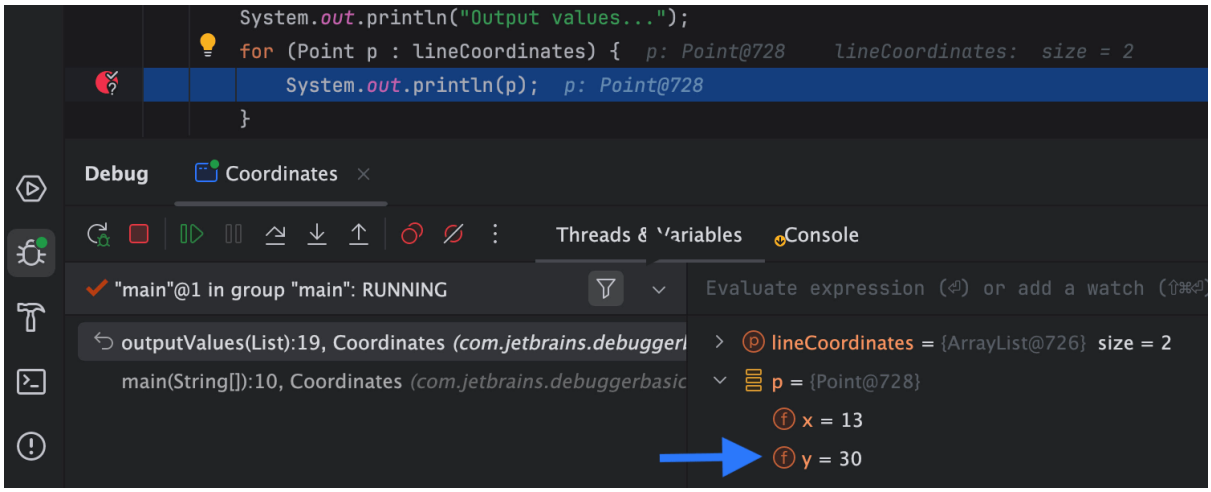
- You can also drag-drop the breakpoint in the gutter and move it to another line of code



By default, clicking a breakpoint icon in the gutter will delete it (you can

modify the default behavior in **Settings/ Preferences**). But if you've defined conditions or other parameters for a breakpoint, you might prefer it to be disabled, rather than deleted, when you click on it. You can do this by right-clicking the breakpoint icon and uncheck **Enabled**. A tick indicates that there is information for this line of code.

- To check how the breakpoint and its conditions work, execute the sample code included in this blog in debug mode. You'll see that this program will pause when the value of field **y** for variable **p** is **30**.



- There's much more to breakpoints. You can right-click on the breakpoint icon in the gutter and click on More. In the dialog that opens, you can modify a breakpoint so that it doesn't suspend the program execution and instead logs an expression when it is reached. Let's log the value of the x and y fields of the Point class and rerun our code. Now the code execution doesn't stop at the breakpoint – instead it logs the expression we defined to the console.

Step Actions

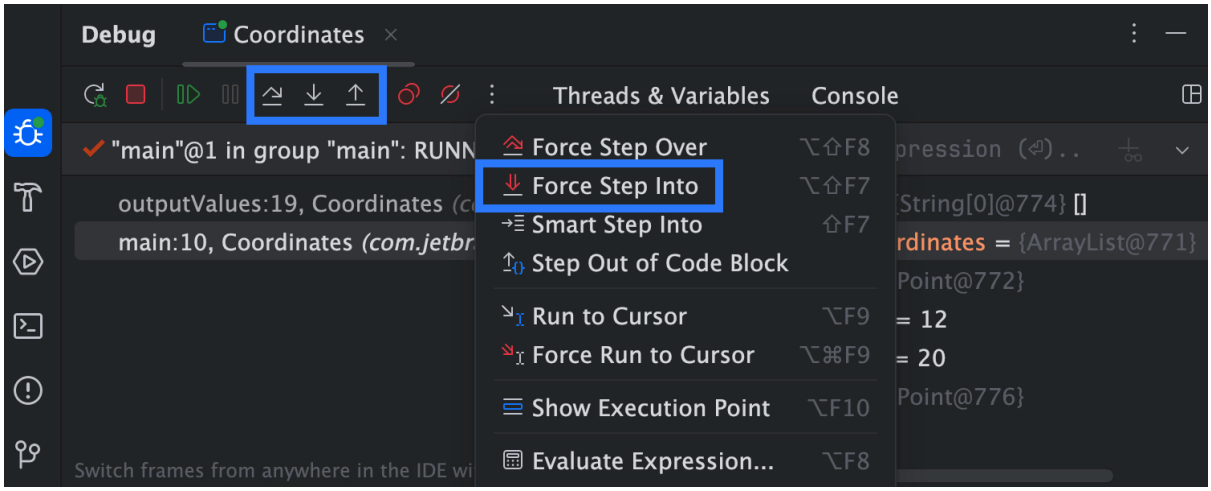
- There are different ways to navigate your code in debug mode. For example, you might prefer to execute a line of code without bothering about the details of the methods being called. Or you might prefer to see which lines of code execute when you call another method from your application, libraries, or APIs. You can do this through the various step actions.
- Set a breakpoint before you start the application in the debug mode. The various step actions are:
    - **Step Over (F8):** lets you execute a line of code and move on to the next line. As you step through the code, you'll see the value of the

variables next to the code in the editor window. These values are also visible in the Variables pane in the Debug window.

- **Step Into (F7):** will take you to the first line of code in a method defined in the same class, or another class in the application.

- **Force Step Into:** lets you debug methods defined in the APIs or libraries. If the source code of the API or library is not available, IntelliJ IDEA decompiles and debugs it for you.

- **Step Out:** you can skip stepping through code in a method line by line and return to the calling method. The called method executes, but without stepping through each line of code in it.



- Let's use all the preceding actions to debug the **Coordinates** class. We'll start by stepping over the lines of code in the **main()** method, stepping into the **removeValues()** method, and force-stepping into the **remove()** method of the **ArrayList** class and the **equals()** method to check how the values of the **lineCoordinates** list are being compared with the value of reference variable **p**, so that a matching value can be removed from the list.

- In the sample application, we discovered that the **'bug'** is caused by the way the **equals()** method compares values. It returns true only if the references match, not if their corresponding field values match.

- Let's fix this bug by overriding the **equals()** method in the **Point** class. To do so, call the **Generate** menu (**Alt + Insert** on Windows/ Linux or **⌘N** on macOS) and select **equals()** and **hashCode()**.

```
Coordinates.java    Point.java
public class Point {
    private int x;
    private int y;

    public Point(int
        this.x = x;
        this.y = y;
    }

    public int getX(
        return x
```

Generate
- Constructor
- Getter
- Setter
- Getter and Setter
- equals() and hashCode()
- toString()
- Override Methods... ^O
- Test...

- Now, let's rerun the code and check whether it is working as expected. Start the application and view the result.

```
public class Coordinates {
    public static void main(String[] args) throws IOException {
```

Run Coordinates — Before

```
Output values...
com.jetbrains.Point@30f39991
com.jetbrains.Point@452b3a41
Output values...
com.jetbrains.Point@30f39991
com.jetbrains.Point@452b3a41
Process finished with exit code 0
```

Run Coordinates — After

```
Output values...
com.jetbrains.Point@549
com.jetbrains.Point@572
Output values...
com.jetbrains.Point@549

Process finished with exit code 0
```

- Everything looks good now. We managed to find a bug and fix it too!

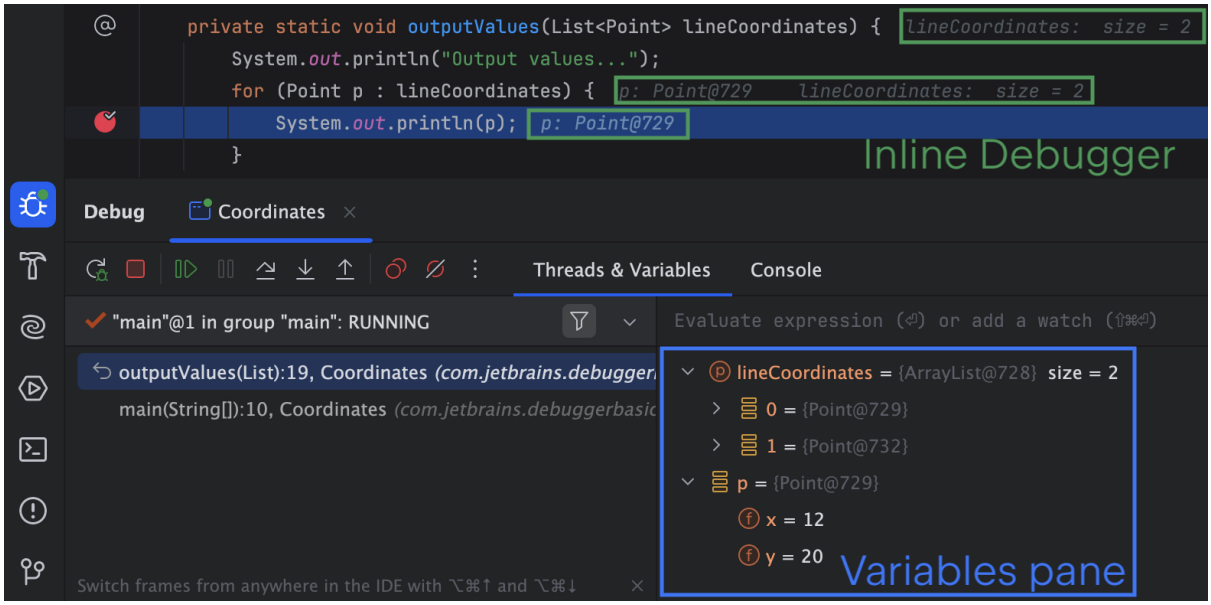
Variables Pane

- The inline debugger is very helpful since it shows the value of the variables in the editor as you step through the code. However, the Variables pane shows a lot more details and includes all fields of variables, including private fields. Clicking on stacks will show us the variables that are relevant to that stack.

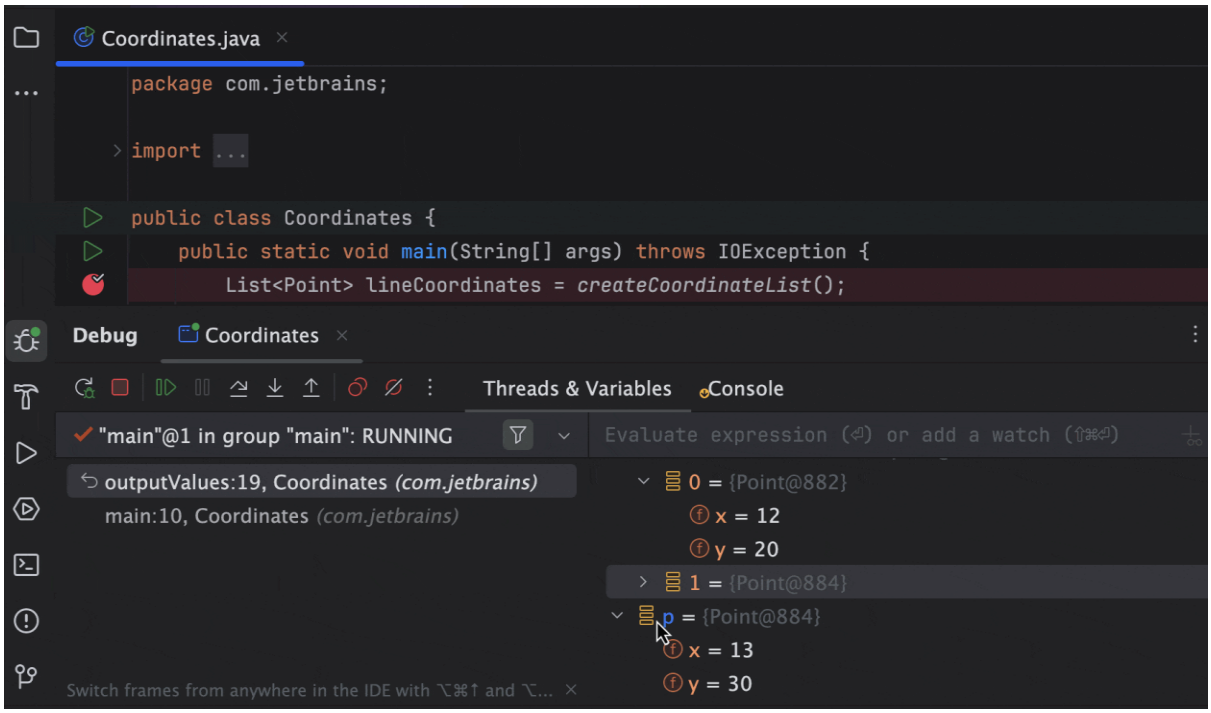
```
private static void outputValues(List<Point> lineCoordinates) {   lineCoordinates:  size = 2
    System.out.println("Output values...");
    for (Point p : lineCoordinates) {   p: Point@729   lineCoordinates:  size = 2
        System.out.println(p);   p: Point@729
    }
```

Inline Debugger

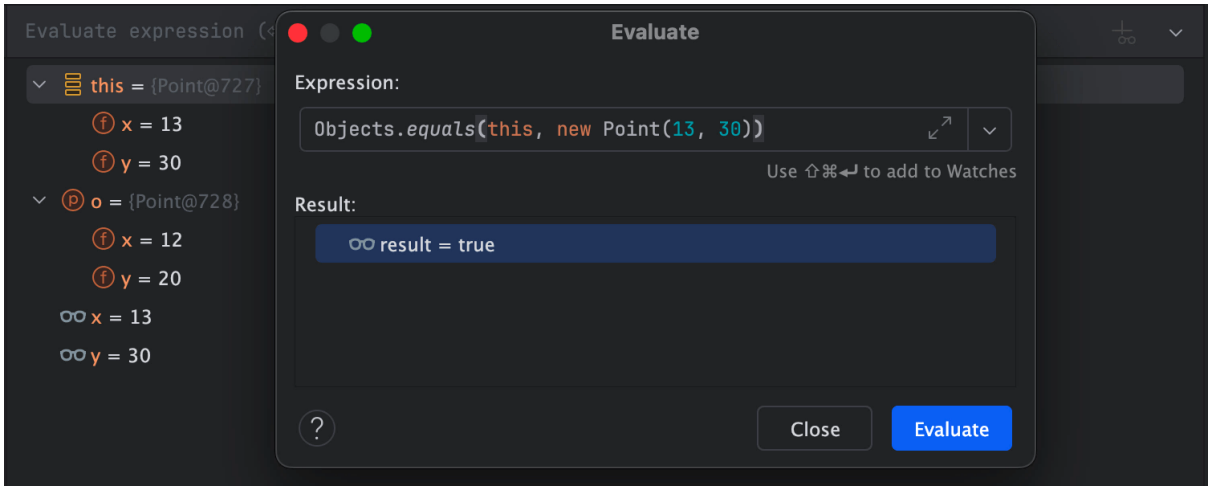
Debug Coordinates



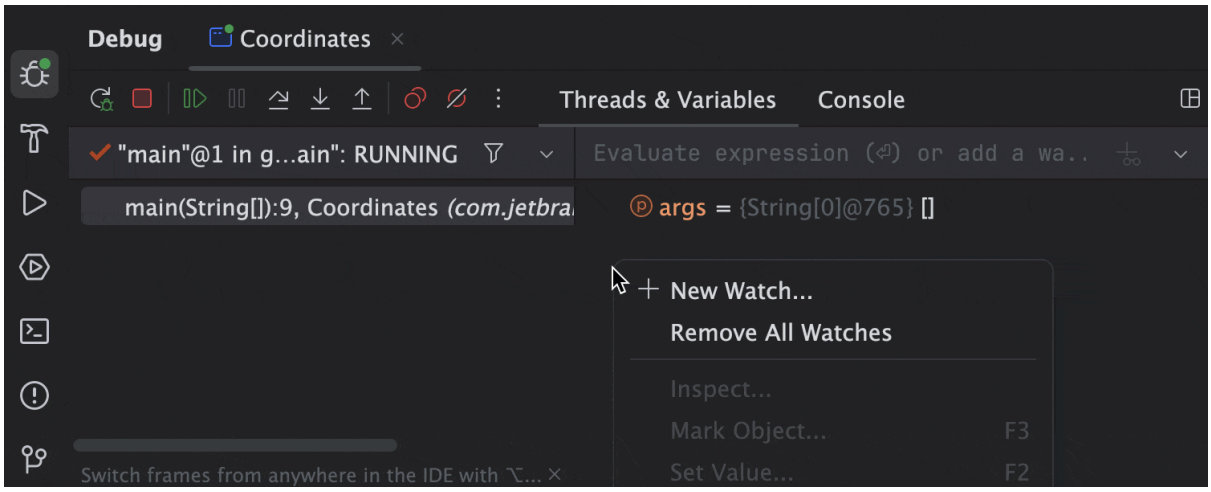
- You can right-click on a variable and select **Jump to Source** (**F4** on Windows/Linux or ⌘↓ on macOS) to view where it was declared to understand your code better. By selecting the option **Jump to Type Source** (**Shift + F4** on Windows/Linux or ⇧ on macOS), you can also view the definition of non-primitive variables.



- In a call stack, you might want to evaluate an expression to verify your assumptions. For example, I can evaluate the value of the **this** variable, or other valid expressions, like **this** is double equal to an instance of the **Point** class, or **this** is **.equals()** to an instance of the **Point** class.

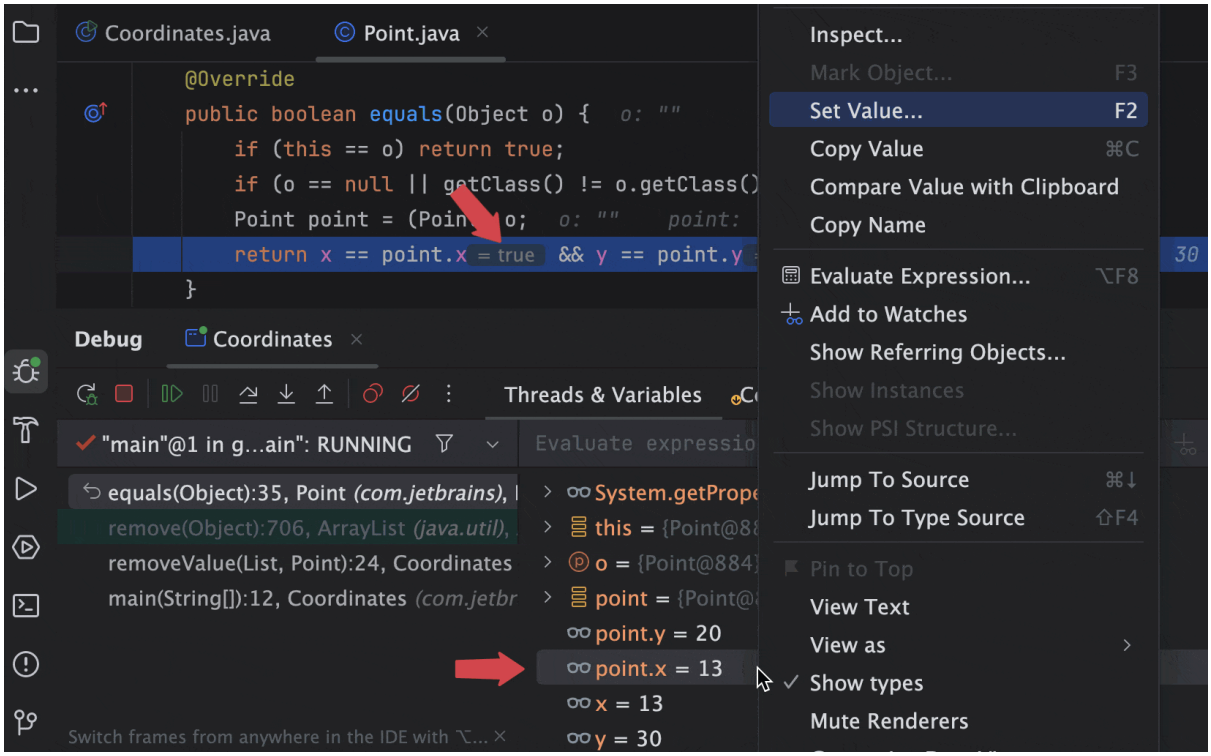


- You can create a variable whose value is accessible in all the call stacks by adding a new watch. Say, **System.getProperty**, and use the name of your OS. You can create watches to view the value of certain variables in all the call stacks. Right-click inside the **Variables Pane** and select **New Watch**.



Modify code behaviour

- Did you know you can change the behavior of your code without changing its source? And this applies to the code defined by another API or framework too.
- In this code execution, the **x** and **y** fields of **Point** instances being compared are equal, and this **equals()** method is about to return true. We can change the value of a variable by right-clicking it in the variable pane and selecting **Set Value (F2)**. When we do this the behavior of the code changes. With the modified value, the **equals()** method returns **false** and this value won't be removed from the **ArrayList**.



Summary

- The debugger is a powerful and versatile tool that executes programs in a controlled environment. With a debugger, you can see the inner state of an application, find bugs, understand code, and do many other things.
- See also:
  - [Debugger Upskill: Basic and Advanced Stepping]
  - [Debugger Upskill: Variables, Evaluate Expression, and Watches]

Note:

- ✓ The IntelliJ entire debugging concept has referred from [Link].
- ✓ IntelliJ official Link to learn Debugging concept [Link].

Debugging in JavaScript Chrome DevTools

- It is really hard to explain on the thing so for that refer the following links.
  - Source Penal overview [Link]
  - Debug Javascript [Link]

Note: From the first link you can get all other links and you can follow and read.

---------- The END ----------