**Eclipse Debugging**

**What is Debugging?**

Debugging is the process of running a program **step by step** so that we can carefully check how it works, find out where the problem (bug) is, and then fix it.

When a program does not behave as expected (for example, it gives wrong output, crashes, or stops suddenly), debugging helps us to look inside the program and understand what is really happening.

**Important Terminologies**

1. **Bug**
   * A bug is an error, defect, or mistake in a program.
   * It causes the program to behave in an unexpected way.
   * Example: If you write code to divide two numbers, but forget to handle the case when the denominator is 0, the program will crash. That is a bug.
2. **Debug**
   * The word comes from "De + Bug" which means **removing the bug**.
   * Debugging is the act of identifying, analyzing, and fixing bugs in the code.
3. **Debugger**
   * A **debugger** is a special tool (software/program) that helps us in the debugging process.
   * It allows us to run the program step by step, pause it, check the values of variables, and understand the program’s flow.

**Debugging Tools for Java**

There are two main types of debuggers used in Java programming:

**1. JDB (Java Debugger)**

* It comes with the JDK installation.
* It is available inside the <JAVA\_HOME>\bin folder as jdb.exe.
* JDB is a **CUI tool (Character User Interface)** → it runs in the command line/terminal.
* It is powerful but **not very user-friendly**.
* Because of its complexity, it is **not commonly used in the industry**.

**2. IDE Supplied Debuggers**

* Popular IDEs like **Eclipse, IntelliJ, and IBM RAD** provide their own debugging tools.
* These are **GUI (Graphical User Interface)** based → you can click, drag, and use buttons to debug instead of typing commands.
* They are **very user-friendly** and make the debugging process easy.
* These are the **industry standard** tools for debugging Java code.

**Where Debugging is Required in a Company?**

Debugging is not something you do only when the program crashes. In real company projects, debugging is needed in many situations. Let’s see them one by one:

**1. To Find, Analyze, and Fix Logic Errors**

* Sometimes the program compiles and runs, but the output is **not what we expect**.
* Example: You wanted the program to calculate the average, but it is calculating the sum instead.
* These are called **logic errors**, and debugging helps to trace step by step and see where the mistake happened.

**2. When Unit Testing Fails**

* In companies, developers write **unit tests** (like JUnit tests) to automatically check if each part of the code is working correctly.
* If a unit test fails, it means some part of the logic is wrong.
* In such cases, the developer runs the test case in **debug mode** to watch how the code is executing and fix the problem.

**3. Understanding an Existing Project**

* When a developer **joins a company** or is assigned to a **new/existing project**, they may not understand the project’s complete flow immediately.
* Debugging helps them:
  + Run the project step by step.
  + See which methods are called first.
  + Understand how data is moving across classes.
* This way, debugging is not only for fixing bugs, but also for **learning the flow of the project.**

**4. Debugging Production Issues**

* When a project is running in **production (live environment)** and customers face issues, developers usually cannot directly debug the code on the live server (for safety reasons).
* Instead:
  + They check the **log files** (log messages written by the application).
  + These logs give hints about where the error happened.
  + Then, developers try to **reproduce the same issue** in their local environment.
  + Finally, they use debugging in their IDE (like Eclipse) to trace the bug and fix it.



**Debugging Life Cycle Steps**

**1. Identify the Error**

* First, we need to recognize that there is an error in the system.
* This can come from:
  + User complaints,
  + Failed test cases, or
  + Unexpected program behavior.
* Without identifying the error, we don’t even know what needs fixing.

**2. Find the Error Location**

* After knowing there is a problem, the next step is to **locate where in the code the error is happening**.
* This can be done by:
  + Checking log files,
  + Running the program in debug mode,
  + Narrowing down the method/class where the issue occurs.

**3. Analyze the Error**

* Once the location is known, study the code carefully.
* Ask questions like:
  + Why did this error happen?
  + Is it because of wrong logic?
  + Is it caused by wrong input data?
  + Or is it because of dependency failure (like database connection, API call, etc.)?

**4. Prove the Analysis**

* Before fixing, you need to confirm that your understanding is correct.
* For example:
  + If you think a variable is always becoming null, you can **add breakpoints or log statements** to prove that it is really null during execution.
* This step avoids fixing the wrong thing.

**5. Cover Lateral Damage**

* While debugging, we must check if the bug has **side effects** in other parts of the system.
* Example: Fixing one calculation in billing should not break the invoice generation.
* This step ensures that the fix will not damage other functionalities.

**6. Fix and Validate**

* Finally, apply the fix in the code.
* Run the program again in debug mode and test to confirm that:
  + The bug is solved.
  + No new bugs were introduced.
* Once validated, the fix is complete.

✅ **In short:**

1. Identify → Find what the error is.
2. Locate → Find where the error is in the code.
3. Analyze → Understand why the error happened.
4. Prove → Confirm your analysis with evidence.
5. Cover Damage → Check that fixing won’t break other features.
6. Fix & Validate → Apply fix and re-test.

**🔎 Debugging in Eclipse (Detailed Explanation)**

Debugging is like **watching your code run in slow motion**.  
Instead of the program running from start to finish instantly, Eclipse lets you:

* Pause execution at specific points (breakpoints),
* Watch what variables contain at each step,
* Control how you move forward in the program,
* And check if the logic is working as expected.

**🟢 Steps to Start Debugging in Eclipse**

1. **Put a Breakpoint**
   * Right-click the left margin of the editor (or double-click).
   * A blue dot will appear → this is your breakpoint.
2. **Run in Debug Mode**
   * Right-click your class → Debug As → Java Application.
   * Eclipse will pause execution at the first breakpoint it encounters.
3. **Switch to Debug Perspective**
   * Eclipse shows you:
     + **Editor** → where your code is.
     + **Debug Panel** → buttons for Step Into, Step Over, etc.
     + **Variables View** → live values of variables.
     + **Breakpoints View** → list of all breakpoints.

**🟡 Breakpoints in Detail**

Breakpoints are the heart of debugging. Without them, the program just runs normally.

1. **Method Breakpoint**
   * Placed at the method definition (line where method starts).
   * Program pauses **whenever the method is called**.
   * Good when:
     + You want to debug a full method.
     + You don’t know exactly which line has the problem, but you know the method is involved.
2. **Line Breakpoint**
   * Placed at a **specific line inside a method**.
   * Program pauses exactly on that line.
   * Good when:
     + You suspect a specific calculation or condition is wrong.
     + Example: checking a variable value in a loop.

👉 You can also disable breakpoints without deleting them (right-click → disable). This is helpful when you want to keep them for later.

**🔵 Main Debugging Operations (Step Control)**

When the program pauses at a breakpoint, you get control. These are the key actions:

1. **Step Into (F5)**
   * If the current line calls another method, Step Into will take you **inside that method**.
   * Good when you want to see internal logic.
   * Example: You’re on calculateTotal(), pressing F5 will open that method.
2. **Step Over (F6)**
   * Executes the current line but does **not go inside method calls**.
   * It just gives the result of the method and moves to the next line.
   * Good when you **don’t care about internal method details**.
3. **Step Return (F7)**
   * Finishes execution of the current method and returns to the caller.
   * Good when you accidentally stepped into a method you didn’t want to see.
4. **Resume (F8)**
   * Continues program execution until the **next breakpoint**.
   * If no more breakpoints, program runs till end.
5. **Terminate (Ctrl + F2)**
   * Stops debugging immediately.
   * Good if you realize you’re stuck in an infinite loop or you don’t want to continue.
6. **Ctrl + R** → Run to Line
   * You place the cursor on any line in the editor.
   * Press Ctrl+R.
   * The program will run normally until it reaches that line → then it pauses there (like a temporary breakpoint).
   * Useful when you don’t want to step line by line, but quickly jump to a specific line.

**7. Ctrl + F5 → Step Into Selection**

* Instead of stepping into every method call, you can directly jump inside the **method where your cursor is placed**.
* Example:
* calculateTotal();
* generateInvoice(); // cursor here + Ctrl+F5 → will step directly into generateInvoice()
* logResult();
* Very useful to **skip unwanted methods** and go only into the method you care about.

## 🟣 ****Why Debugging is Powerful in Eclipse****

* **See Variable Values Live**
  + In the Variables View, you can see values of local variables at each step.
  + You can even **change variable values during debugging** (right-click → Change Value).
* **Conditional Breakpoints**
  + You can set a breakpoint that only activates when a condition is true.
  + Example: In a loop, pause only when i == 50.
* **Watch Expressions**
  + You can create watch expressions for variables or calculations to monitor their values continuously.
* **Multi-thread Debugging**
  + If your project uses multiple threads, you can pause one thread while letting others run.

**⚙️ Working with Step Filters in Eclipse Debugging**

When you debug in Eclipse, sometimes you don’t want to step into certain classes or methods (like Java’s internal library code).

👉 Example:  
If you press **F5 (Step Into)** on this line:

System.out.println("Hello Debugging");

By default, Eclipse would take you **inside the println() method**, which belongs to Java’s standard library.  
But usually, as developers, we don’t care how println() works internally — we just want to debug *our own code*.

This is where **Step Filters** help.

**🔎 What are Step Filters?**

* **Step Filter = A rule to exclude specific packages, classes, or methods from debugging.**
* When enabled, Eclipse will **skip over those packages** when you press F5.
* This keeps your debugging focused only on your project code, not unnecessary library details.

**🛠️ Steps to Use Step Filters in Eclipse**

**Step 1: Add Packages to Exclude**

1. Go to:  
   Window → Preferences → Java → Debug → Step Filtering
2. Add the packages/classes you don’t want to step into.
   * Example: java.\*, javax.\*, org.eclipse.\*
   * This means you will not enter standard Java library or Eclipse-related code while debugging.

**Step 2: Activate Step Filters**

* In the same preferences window, make sure **“Use Step Filters”** option is checked.
* Or, while debugging, you can enable/disable it from the **Debug View toolbar** (a small filter icon).

**Step 3: Debug Normally**

* Now when you debug:
  + Press **F5 (Step Into)** on excluded packages like System.out.println() → Eclipse will **skip it**.
  + It will only step into **your project’s code**.

**✅ Advantages of Step Filters**

1. Saves time — you don’t waste effort going inside predefined library code.
2. Keeps debugging focused only on your own classes/methods.
3. Helps especially in large projects that use many external libraries (Spring, Hibernate, etc.).

💡 **Example Setup:**

* Add these common filters:
  + java.\*
  + javax.\*
  + org.\*
  + com.sun.\*
* With this setup, your debugging will stay clean and focused.

**🖥️ Watching Values and Managing Breakpoints in Eclipse Debugging**

When we debug in Eclipse, it’s not only about stopping at breakpoints — it’s also about **seeing how the data changes at runtime**. Eclipse provides multiple windows and views to help us.

**🔎 1. Watching Variable Values**

* While in Debug Mode, you can **hover the mouse** over a variable or parameter.
* Eclipse will immediately show its **current value in a tooltip**.
* This is the quickest way to check values without opening other windows.

👉 Example:

int total = a + b; // hover over 'a', 'b', or 'total' to see their live values

**📊 2. Variables Window**

* The **Variables View** shows all local variables of the current method in a structured way.
* You can expand objects to see their internal fields.
* It also updates live as you move step by step.
* Bonus: You can even **change variable values manually** during debugging (right-click → *Change Value*).
  + Useful for testing how the program behaves with different values without restarting.

**🧮 3. Expressions Window**

* The **Expressions View** allows you to type custom formulas/expressions and watch their values.
* Example:
  + You have variables price and quantity.
  + In Expressions Window, you can add price \* quantity.
  + Eclipse will automatically evaluate it as you step through.
* This is very useful for checking **business logic calculations** without writing extra code.

**🟥 4. Breakpoints Window**

* The **Breakpoints View** lists all breakpoints you have set in the project.
* From here, you can:
  + Enable/disable breakpoints (without deleting them).
  + Delete unwanted breakpoints.
  + Add **conditional breakpoints** (pause only if a condition is true, like i == 100).
  + Manage breakpoints across multiple files easily.

👉 You can even add/remove breakpoints **while the program is already in Debug Mode**.

**✅ In Summary**

1. **Hover Mouse** → Quick check of variable value.
2. **Variables Window** → See and edit current variable values.
3. **Expressions Window** → Add custom formulas and watch their results.
4. **Breakpoints Window** → Manage all breakpoints (enable, disable, add conditions).