JAVA-DEBUGGING

Introduction to Debugging in Java

Definition:
Debugging is the **process of identifying, analyzing, and fixing errors or bugs** in a program to ensure it behaves as intended.

- It helps developers **understand program flow** and **resolve issues efficiently.**

Why Debugging is Needed

- To **find and fix logical, runtime, and compilation errors**
- Understand **program execution flow**
- Improve **code quality and reliability**
- Essential for **interview preparation and real-world coding**

Types of Errors

Type	Description
Syntax Error	Mistakes in code grammar; caught at compile-time
Runtime Error	Occurs during execution (e.g., NullPointerException)
Logical Error	Code runs but produces **wrong output**

Common Debugging Techniques in Java

A. Print Statements

```
System.out.println("Variable x: " + x);
```

- Quick check of variable values or flow

B. Using IDE Debugger

- Set **breakpoints**
- Step through code: **Step Over**, **Step Into**, **Step Out**
- Inspect **variables and stack traces**

C. Logging

```
import java.util.logging.*;

Logger logger = Logger.getLogger("MyLogger");
logger.info("Current value of x: " + x);
```

- Useful for **large applications**
- Can control **log levels**: INFO, WARNING, SEVERE

D. Exception Handling

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    e.printStackTrace();
}
```

- Catch **runtime errors** and analyze stack trace

Real-Life Analogy

- Debugging → Detecting **why your car won't start**
- Print statements → Checking **fuel, battery, engine**
- IDE Debugger → Using **diagnostic tools**
- Exception handling → Catching issues **before breakdown**

Best Practices

- Write **readable and modular code** → easier to debug
- Use **IDE tools** instead of only print statements

- Handle **exceptions properly**
- Test **edge cases**
- Learn to **read stack traces efficiently**

Advantages

- Reduces **development time**
- Improves **code quality**
- Helps understand **complex code flows**
- Essential for **interview problem-solving**

Common Interview Questions (Cognizant Level)

Q1. What is debugging?

A. Process of finding, analyzing, and fixing errors in code.

Q2. Difference between debugging and testing?

- Debugging → Fixes problems in code
- Testing → Verifies if code works correctly

Q3. What tools are used for debugging in Java?

A. IDE debugger (Eclipse, IntelliJ), logging, print statements

Q4. How do you debug a NullPointerException?

A. Check variable initialization and trace where null is being used

Q5. What are breakpoints?

A. Points in code where execution **pauses** for inspection

One-Line Summary (Quick Revision)

Debugging is the systematic process of identifying and fixing errors in code using techniques like breakpoints, logging, and exception handling.

Getting Started with IntelliJ IDEA Debugger

Definition:

IntelliJ IDEA Debugger is a **built-in tool** that allows developers to **inspect, pause, and control program execution** to find and fix bugs efficiently.

- It helps in **understanding program flow** and **tracking variable states.**

Why Use IntelliJ Debugger

- Step through code **line by line**
- Inspect **variables and objects** in real-time
- Analyze **method calls and stack traces**
- Detect **logical and runtime errors** without modifying code

---

Key Features

Feature	Purpose
Breakpoints	Pause execution at a specific line
Step Over	Execute current line; move to next line (without entering methods)
Step Into	Enter into a method call to debug inside
Step Out	Exit current method and return to caller
Resume Program	Continue execution until next breakpoint or end
Watches	Track specific variables or expressions
Evaluate Expression	Test expressions at runtime
Frames/Call Stack	See method execution sequence

How to Start Debugging in IntelliJ IDEA

Step 1: Set Breakpoints

- Click the **left gutter** next to a line number → red dot appears
- Execution will **pause** at this line

Step 2: Run Debug

- Right-click main class → **Debug 'ClassName'**
- Program starts and **pauses at breakpoints**

Step 3: Navigate Execution

- **Step Over (F8)** → Move to next line
- **Step Into (F7)** → Enter method
- **Step Out (Shift+F8)** → Exit method

Step 4: Inspect Variables

- Hover over variable to see value
- Use **Variables panel** to monitor all variables
- Add **Watches** to track custom expressions

Step 5: Resume / Stop

- **Resume Program (F9)** → Continue execution
- **Stop Debugger** → Terminate program

Real-Life Analogy

- **Debugger** → Detective investigating a case
- **Breakpoints** → Place to pause and inspect clues
- **Step Into** → Enter a suspect's room to inspect
- **Variables** → Evidence showing the state at that moment

Example: Debugging Simple Java Program

```
public class DebugExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;
        int c = divide(a, b);
        System.out.println("Result: " + c);
    }

    public static int divide(int x, int y) {
        return x / y; // Set breakpoint here
    }
}
```

Debug Steps:

1. Set breakpoint at return x / y;
2. Run in Debug mode
3. Hover over x and y → Inspect values
4. Detect y = 0 → Prevent runtime exception

Advantages

- Easy to **trace program flow**
- Reduces **trial-and-error with print statements**
- Helps debug **complex code and multithreaded apps**
- Supports **real-time variable inspection and expression evaluation**

Best Practices

- Use **breakpoints selectively**
- Use **conditional breakpoints** for specific cases
- Monitor **loops carefully** to avoid long pauses
- Combine with **logging** for large applications

Common Interview Questions (Cognizant Level)

Q1. What is a breakpoint?

A. A marker that **pauses program execution** for inspection.

Q2. Difference between Step Over and Step Into?

- Step Over → Execute current line without entering method
- Step Into → Enter into method to debug line by line

Q3. How to watch a variable in IntelliJ?

A. Right-click variable → **Add to Watches**, or use Variables panel

Q4. Can you debug multithreaded programs in IntelliJ?

A. Yes, IntelliJ shows **thread list and call stacks**

Q5. How to evaluate expressions at runtime?

A. Use **Evaluate Expression tool (Alt+F8)**

One-Line Summary (Quick Revision)

IntelliJ IDEA Debugger lets you pause, step through, and inspect program execution to identify and fix bugs efficiently.

Basic Debugging Techniques in Java (IntelliJ IDEA)

Definition:

Debugging techniques are methods used to **inspect, trace, and control program execution** to identify and fix errors.

Key techniques include:

- Setting Breakpoints
- Step Over
- Step Into
- Step Out

Why They Are Needed

- To **pause execution** at critical points
- **Analyze variable values and program flow**
- Quickly **identify logic and runtime errors**
- Essential for **complex programs and interviews**

Setting Breakpoints

- **Breakpoint** → Marker that pauses program at a line for inspection
- **How to set:** Click the **left gutter** next to the line number → red dot appears
- **Conditional Breakpoint:** Pause **only if a condition is true**

Example:

```
int x = 10;
int y = 0;
int z = x / y; // Set breakpoint here
```

- Execution pauses here, allowing inspection of x and y

Step Over (F8)

- **Moves to the next line of code** in the same method
- **Does not enter into method calls**
- Useful for **skipping internal logic of functions** you trust

Example:

```
int result = add(5, 3); // Step Over moves to next line without entering add()
System.out.println(result);
```

Step Into (F7)

- **Enters into the method call** to debug line by line inside the method
- Useful when you want to **understand method logic**

Example:

```
int result = add(5, 3); // Step Into enters add() method
System.out.println(result);
```

Step Out (Shift + F8)

- **Exits the current method** and returns to the caller
- Useful when you are **done inspecting a method** and want to continue in the main flow

---

Real-Life Analogy

- **Breakpoint** → Pause at a checkpoint
- **Step Over** → Skip a room in a house tour
- **Step Into** → Enter the room to inspect details
- **Step Out** → Leave the room and continue the tour

Advantages

- Visualize **program execution**
- Inspect **variables and object states**
- Efficiently **detect logical and runtime errors**
- Reduces reliance on **print statements**

Best Practices

- Set breakpoints on **critical lines** only
- Use **conditional breakpoints** for loops or large iterations
- Combine Step Over, Into, and Out to **navigate efficiently**
- Inspect **variable states and stack traces** at each breakpoint

---

Common Interview Questions (Cognizant Level)

Q1. What is a breakpoint?

A. A line where program execution **pauses for inspection.**

Q2. Difference between Step Over and Step Into?

- Step Over → Next line without entering method
- Step Into → Enter method to debug inside

Q3. What is Step Out?

A. Exit current method and return to caller

Q4. Why are conditional breakpoints used?

A. To **pause only when specific conditions** are met

Q5. Which keys are used in IntelliJ IDEA?

- Step Over → F8
- Step Into → F7
- Step Out → Shift + F8

One-Line Summary (Quick Revision)

Basic debugging techniques—breakpoints, Step Over, Step Into, Step Out—allow controlled execution and inspection of Java programs for efficient error detection.

Advanced Debugging Techniques in Java (IntelliJ IDEA)

Definition:
Advanced debugging techniques help developers **inspect, analyze, and control program behavior more deeply** than basic breakpoints.
Key techniques include:
- Watch Variables
- Debug Console

Why Advanced Debugging is Needed
- Debug **complex applications** with many variables
- Track **changes in variable values dynamically**
- Execute **expressions on the fly** without changing code
- Useful for **Cognizant interviews and real-world projects**

Watch Variables
- **Watch** allows you to **monitor specific variables or expressions** while debugging
- Updates automatically whenever program execution passes a breakpoint

How to Use Watch Variables
1. Start debugging your program
2. In the **Variables panel**, right-click → **Add to Watches**
3. Enter a **variable or expression**
4. IntelliJ updates the value dynamically

Example:
```
int a = 5;
int b = 10;
int sum = a + b; // Add 'sum' to Watch
```
- Watch panel will show sum = 15 and update if variables change

Use Cases:
- Track loop counters
- Monitor computed values
- Debug dynamic object states

Debug Console
- **Debug Console** allows you to **interact with the running program** during debugging
- Execute **Java expressions, method calls, or assignments**
- Helps **test fixes or calculations** without stopping the program

How to Use Debug Console
1. Pause program at a breakpoint
2. Open **Debug Console tab**
3. Type expressions:

```
sum = sum + 5;
System.out.println(sum);
```

4. Values update in **variables panel and console output**

Use Cases:
- Test **alternative logic**
- Inspect objects and lists
- Dynamically modify variables during execution

Real-Life Analogy
- **Watch Variables** → Monitor a patient's vitals in ICU
- **Debug Console** → Talk to the patient, give instructions, and see immediate results

---

Advantages

- Provides **real-time insights** into program state
- Helps debug **complex expressions and objects**
- Reduces need for **temporary code changes or print statements**
- Makes debugging **faster and precise**

Best Practices

- Add only **relevant variables or expressions** to watch
- Use console for **safe evaluation**; avoid modifying critical program logic
- Combine **watch, breakpoints, and step commands** for maximum efficiency
- Use **conditional watches** in loops or large data structures

---

Common Interview Questions (Cognizant Level)

Q1. What is a watch variable in IntelliJ IDEA?

A. A variable or expression monitored dynamically during debugging.

Q2. How is the debug console used?

A. Execute Java expressions or modify variables while program is paused.

Q3. Difference between watch variable and console evaluation?

- Watch → Automatically updates with program execution
- Console → Manually executes expressions or modifies values

Q4. Can you modify variables in watch or console?

A. Yes, console allows dynamic changes; watch only monitors value

Q5. Why use advanced debugging over print statements?

A. Provides **dynamic inspection, precision, and saves development time**

One-Line Summary (Quick Revision)

Advanced debugging in IntelliJ IDEA—using watch variables and debug console—lets you dynamically monitor and manipulate program state for faster, precise error detection.
