Lecture 5

Program Flow & Debugging

Learning objectives

- After this lecture, students will be able to:
  - Understand how object-oriented Java programs execute
  - Trace program flow from main() through object interactions
  - Distinguish between object creation, method invocation, and data transfer
  - Identify and resolve common runtime and logical errors
  - Apply systematic debugging strategies to fix code defects
  - Read and interpret stack traces effectively

Lecture outline

- **Program Flow & Object Interaction**
  - Class vs Object fundamentals
  - Program execution model
  - Object interaction patterns
  - Object composition and dependencies
- **Debugging & Error Resolution**
  - Types of programming errors
  - Reading stack traces
  - Common error patterns
  - Debugging strategies and tools

The central problem

I can write classes, but I can’t make them work together

- Common student challenges:
  - Writing multiple classes successfully

    But…
  - Unable to determine which object calls which
  - Misunderstanding execution order
  - Losing track of data flow between objects

☞ **Goal:** Make object-oriented program execution visible and comprehensible

Class vs Object

Class	Object
• A blueprint or template	• An instance created from a class
• Does not execute code	• Actual entity that executes at runtime
• Defines structure and behavior	• Occupies memory
`class Student {`	`Student s1 = new Student();`
`    String name;`	
`    int age;`	
`    void study() { }`	
`}`	

Key Principle: No objects → no program execution

Program entry point

- Every Java application begins at main()

```
public class Main {
    public static void main(String[] args) {
        // Execution starts here
        // From here: create objects, invoke methods
        // Program flow spreads to other classes
    }
}
```

- The main() method is:
  - The mandatory entry point
  - Static (exists without object instantiation)
  - The root of all program execution

Understanding program flow

- **Program flow** is the sequential order of code execution.
- Example execution sequence:
  - JVM invokes main()
  - Objects are instantiated
  - Methods are called on objects
  - Control flows to called methods
  - Execution returns to caller
  - Program terminates when main() completes
- **Critical note**: Java does not execute all classes simultaneously. Execution follows method call chains.

Simple execution example

```
class Student {
  void sayHello() {
    System.out.println("Hello!");
  }
}

public class Main {
  public static void main(String[] args) {
    Student s = new Student();
    s.sayHello();
  }
}
```

Execution trace:

1. `main()` begins
2. `Student` object created in memory
3. `sayHello()` invoked on object `s`
4. `"Hello!"` printed to console
5. Control returns to `main()`
6. Program terminates

Object Interaction Fundamentals

Object Interaction: One object invoking methods on another object

- Real-world analogy:
  - A customer placing an order with a cashier
  - A student asking a question to a teacher
  - A driver requesting service from a mechanic
- In code:
  - Object A holds a reference to Object B
  - Object A calls methods on Object B
  - Data may flow between objects through parameters and return values

Two-object interaction example

```java
public class Teacher { // Teacher.java
    void teach() {
        System.out.println("Teaching OOP concepts...");
    }
}
public class Student { // Student.java
    void attendClass(Teacher t) {
        t.teach();
    }
}
public class Main { // Main.java
    public static void main(String[] args) {
        Teacher teacher = new Teacher();
        Student student = new Student();
        student.attendClass(teacher);
    }
}
```

Note: The `Student` object receives a `Teacher` reference and uses it temporarily.

Tracing program flow

- Execution sequence for previous example:
  1. `main()` begins execution
  2. `Teacher` object instantiated
  3. `Student` object instantiated
  4. `attendClass()` invoked with teacher reference
  5. Inside `attendClass()`:
  6. `teach()` invoked on `teacher` object
  7. `"Teaching OOP concepts..."` printed
  8. Control returns to `attendClass()`
  9. Control returns to `main()`
  10. Program terminates

➢ **Always ask:** Who is calling whom? In what order?

Relationship patterns: **uses-a** vs **has-a**

- **uses-a** Relationship (Dependency)
  - Object uses another object temporarily
  - Passed as method parameter
  - Temporary association

```
student.attendClass(teacher); // temporary use
```

- **has-a** Relationship (Composition)
  - Object contains another object as a field
  - Persistent association
  - Stronger coupling

```
class Student {
    private Teacher teacher; // ownership
}
```

→ **has-a** creates a stronger dependency than **uses-a**

Composition example (has-a)

The Student object **owns** a Teacher reference throughout its lifetime.

```
class Student {
    private Teacher teacher;

    Student(Teacher teacher) {
        this.teacher = teacher;
    }

    void study() {
        teacher.teach();
    }
}

public class Main {
    public static void main(String[] args) {
        Teacher t = new Teacher();
        Student s = new Student(t);
        s.study();
    }
}
```

Real-world example: eCommerce system

```
class ShoppingCart {
    private Product[] items;
    private int count = 0;

    void addProduct(Product p) {
        items[count++] = p;
    }

    double calculateTotal() {
        double total = 0;
        for (int i = 0; i < count; i++) {
            total += items[i].getPrice();
        }
        return total;
    }
}
```

```
class Product {
    private String name;
    private double price;

    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    double getPrice() {
        return price;
    }
}
```

eCommerce flow analysis

```
public static void main(String[] args) {
    ShoppingCart cart = new ShoppingCart();

    Product laptop = new Product("Laptop", 999.99);
    Product mouse = new Product("Mouse", 29.99);

    cart.addProduct(laptop);
    cart.addProduct(mouse);

    double total = cart.calculateTotal();
    System.out.println("Total: $" + total);
}
```

- **Flow:** `main()` → creates `cart` → creates `products` → adds to `cart` → calculates `total` → prints result.

Pass-by-reference in Java

- Java passes object references by value

```java
void changeName(Student s) {
    s.name = "New Name"; // modifies the object
}

public static void main(String[] args) {
    Student student = new Student();
    student.name = "Alice";
    changeName(student);
    System.out.println(student.name); // prints "New Name"
}
```

- **(*)** Modifying object fields affects the original object.
  - This is a common source of confusion.

Common Mistake #1 - NullPointerException

```java
Student s;
s.sayHello(); // Runtime error
```

- **Why this fails:** The reference variable s is declared but not initialized. It contains `null`.
- **Correct approach:**

```java
Student s = new Student();
s.sayHello(); // this works
```

➢ **Intuition:** Declaration creates a reference variable. The new keyword creates the actual object.

Common Mistake #2 - Misunderstanding Execution Order

```java
class A {
    void callB(B b) {
        b.doSomething();
    }
}

class B {
    void doSomething() {
        System.out.println("Executing B");
    }
}
```

- **Key Point:** Code executes according to method invocation order, not file order or class declaration order.
- Execution follows the call chain, not the physical arrangement of code.

Visualizing method calls

- **Good practice:** Draw execution diagrams

```
Main
├── creates Teacher
├── creates Student
└── calls
Student.attendClass()
    └── calls Teacher.teach()
        └── prints output
```

- **Benefit:** Visual representation clarifies complex object interactions

Understanding program flow checklist

1. Always ask: Who invokes whom?
2. Trace execution from main() systematically
3. Never guess - follow the code step by step
4. When confused - draw a diagram
5. Use print statements to verify assumptions

❖ Understanding program flow is essential for debugging and design

Basic Debugging in Java

Why debugging matters?

- "Programs are written for people to read, and only incidentally for machines to execute." - Abelson & Sussman
- **Key insights:**
  - Every programmer writes defective code
  - Professional programmers excel at debugging, not at writing perfect code initially
  - Debugging is a core survival skill in software development
  - The ability to fix errors determines programming competence

What is debugging?

- Debugging is the systematic process of:
    - Identifying why a program behaves incorrectly
    - Determining where the error occurs
    - Fixing the error methodically
- Debugging is NOT:
    - Random code modifications
    - Complete rewrites without understanding the problem
    - Guessing solutions without evidence
    - Panic or frustration-driven changes

Why code fails?

- Common root causes:
  - Incorrect assumptions about language behavior
  - Missing edge cases (empty input, null values, boundary conditions)
  - Misunderstanding of API or library behavior
  - Typographical errors in variable names or syntax
  - Copy-paste errors creating subtle bugs
  - Incomplete testing missing failure scenarios
- Important: Bugs are a normal part of development, not a sign of incompetence

Categories of errors

1. Compile-time errors (Syntax errors)
    - Detected by compiler before execution, prevents program from running
    - E.g. missing semicolon, type mismatch
2. Runtime errors (Exceptions)
    - Occur during execution, make program crashes
    - E.g. `NullPointerException`, `ArrayIndexOutOfBoundsException`
3. Logical errors (Semantic errors)
    - Program runs without crashing
    - Produces incorrect results
    - Hardest to detect and fix

Runtime error example

```java
public class Test {
    public static void main(String[] args) {
        int x = 10;
        int y = 0;
        System.out.println(x / y);
    }
}
```

- Result:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:5)
```

The program compiles successfully but crashes at runtime when attempting division by zero.

Understanding stack traces

- Example stack trace:

```
Exception in thread "main"
java.lang.NullPointerException
    at Student.printName(Student.java:15)
    at Classroom.displayStudents(Classroom.java:23)
    at Main.main(Main.java:6)
```

- Exception type: NullPointerException
- File and line: Student.java, line 15
- Call chain: Read from bottom to top
  - main() called displayStudents()
  - displayStudents() called printName()
  - printName() threw the exception

How to read stack traces

- Step to read stack traces:
  1. Read the exception type
     e.g., `NullPointerException`
  2. Identify the first occurrence in your code
  3. Navigate to the specific file and line number
  4. Examine variables and expressions on that line
  5. Trace backwards through the call chain if needed
- **Tip:** The top of the stack trace shows where the error occurred, not necessarily the root cause.

Debugging `NullPointerException`

- Happens when attempting to use an object reference that points to **`null`**
- Example:

```
String name = null;
System.out.println(name.length()); // Exception
```

- Common causes:
  - Uninitialized object references
  - Methods returning null unexpectedly
  - Forgetting to instantiate with new
  - Improper error handling

Infinite Loops

- Example:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    // Missing: i++
}
```

- **Consequence:** Program never terminates. Loop condition remains true indefinitely.
- **Symptoms:**
  - Program hangs
  - High CPU usage
  - No response to input

Preventing Infinite Loops

- Correct code:

```java
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;  // loop counter updated
}
```

- Prevention strategies:
  - Ensure loop variables are modified
  - Verify termination conditions can be satisfied
  - Use for-loops when iteration count is known
  - Set maximum iteration limits in critical code

Array Index Errors

```java
int[] numbers = new int[5];
for (int i = 0; i <= 5; i++) { // Off-by-one error
    numbers[i] = i * 10;
}
```

- Exception:

```
ArrayIndexOutOfBoundsException:
Index 5 out of bounds for length 5
```

- **Reason:** valid indices are 0-4. Attempting to access index 5 causes a runtime exception.
- Fix:

```java
for (int i = 0; i < 5; i++)
```

Debugging Strategy: Print Logging

- The most fundamental debugging technique for beginners.
- **Benefit:** Simple, universally available, helps trace execution flow and inspect variable values

```java
void processStudents(Student[] students) {
    System.out.println("Processing " + students.length + " students");

    for (int i = 0; i < students.length; i++) {
        System.out.println("Index: " + i);
        System.out.println("Student: " + students[i].getName());
        students[i].calculateGrade();
    }

    System.out.println("Processing complete");
}
```

Debugging Strategy: Divide and Conquer

- **Principle:** Isolate the defective component
- Poor approach:
  - Debug a 300-line program as a whole
  - Try to fix everything simultaneously
- Effective approach:
  - Test individual methods in isolation
  - Verify small components first
  - Gradually integrate tested components
  - Use unit testing to validate individual pieces
- **Example:** If a shopping cart calculation is wrong, test `calculateSubtotal()`, `calculateTax()`, and `calculateTotal()` separately

Summary – Program Flow

- Objects are runtime instances; classes are blueprints
- Program execution always begins at main()
- Objects interact through method calls
- Program flow follows the call chain, not file order
- Composition (has-a) creates persistent relationships
- Dependency (uses-a) creates temporary relationships
- Understanding object interaction is essential for OOP mastery

Summary – Debugging

- Three error types: compile-time, runtime, logical
- Stack traces reveal error location and call chain
- Common errors: NullPointerException, infinite loops, array bounds
- Essential strategies: print logging, divide-and-conquer
- Effective debugging requires methodical thinking, not random changes
- Debugging is a learnable skill that determines programming success