# Assignment 2 Day 2

## Snippet 1:

```
public class Main {
public void main(String[] args) {
System.out.println("Hello, World!");
}
}
```

### What error do you get when running this code?

Main method is not static in class Main, please define the main method as:
  public static void main(String[] args)

## Explaination:

main method of Main class is a gate of program where programming start.
So the main method should be always public static void.
In above code snippet static keyword was missing.

## Corrected code

```
public class Main {
	public static void main(String[] args) {
		System.out.println("Hello, World!");
	}
}
```

In the above code snippet static key word was missing.

## Snippet 2:

```
public class Main {
static void main(String[] args) {
System.out.println("Hello, World!");
}
}
```

### What happens when you compile and run this code?

```
Main method not found in class Main, please define the main method as:
  public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

## Explaination:

main method of Main class is a gate of program where programming start.
So the main method should be always public static void.
In above code snippet static keyword was missing.

## Corrected code

```
public class Main {
        public static void main(String[] args) {
                System.out.println("Hello, World!");
        }
}
```

In the above code snippet public key word was missing

## Snippet 3:

```
public class Main {
        public static int main(String[] args) {
                System.out.println("Hello, World!");
        return 0;
        }
}
```

**What error do you encounter? Why is void used in the main method?**

Error: Main method must return a value of type void in class Main3, please
define the main method as:
  public static void main(String[] args)

## Explaination:

1. **Incorrect Return Type (int instead of void)**
   - In Java, the main method must have the exact signature:

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

   - The Java Virtual Machine (JVM) looks specifically for public static void main(String[] args). If the return type is not void, the JVM will not recognize it as the entry point.
2. **Main Method Must Not Return a Value**
   - The main method is the starting point of execution and does not need to return any value because the JVM does not expect a return type.
   - Returning int suggests that the method should return a numerical value, but main should simply execute and terminate without returning anything.

## Corrected Code

```
public class Main {
        public static void main(String[] args) {
                System.out.println("Hello, World!");
                return 0;
        }
}
```

## Snippet 4:

```
public class Main {
        public static void main() {
                System.out.println("Hello, World!");
        }
}
```

**What happens when you compile and run this code? Why is String[] args needed?**

## Compilation:

The code will **compile successfully** because the syntax is correct, and Java allows defining multiple methods named main with different parameter lists (method overloading).

## Execution:

However, when you try to **run** the program, you will get an error similar to:

```
Error: Main method not found in class Main4, please define the main method as:
  public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

## Explaination:

1. **JVM Entry Point Requirement:**
- Java requires the main method to have the signature:
  public static void main(String[] args)
- Without this exact signature, the JVM does not recognize it as the starting point.

2. **Command-Line Arguments:**
- String[] args allows passing arguments from the command line when executing the program.

3. **Method Overloading in Java:**
- Java allows overloading methods, so if you define:
  public static void main()
  It is treated as a separate method, not the special main method that the JVM needs.

## Corrected Code

```
public class Main {
        public static void main(String[] args) {
                System.out.println("Hello, World!");
                return 0;
        }
}
```

## Snippet 5:

```
public class Main {
	public static void main(String[] args) {
		System.out.println("Main method with String[] args");
	}
	public static void main(int[] args) {
		System.out.println("Overloaded main method with int[] args");
	}
}
```

## Output

```
Main method with String[] args
```

## Can you have multiple main methods? What do you observe?

Yes, Java allows multiple main methods in a class through method overloading. This means you can define multiple methods with the same name (main) but with different parameter lists.

- What I observe is,

The code compiles successfully because method overloading is allowed in Java. And give output (**Main method with String[] args**).

- This Happens because

The JVM only recognizes the main method with the signature:
**public static void main(String[] args)**

- The overloaded method **public static void main(int[] args)** will not be executed automatically by the JVM because it does not match the expected signature.

## Corrected Code

```
public class Main5 {
	public static void main(String[] args) {
		System.out.println("Main method with String[] args");
		main(intArray); // Calling overloaded main method
	}
	public static void main(int[] args) {
		System.out.println("Overloaded main method with int[] args");
	}
}
```

## Output

```
Main method with String[] args
Overloaded main method with int[] args
```

## Snippet 6:

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

### What error occurs? Why must variables be declared?

## Compilation:

```
Main6.java:3: error: cannot find symbol
int x = y + 10;
        ^
  symbol:   variable y
  location: class Main6
1 error
```

While compilation give Error because we not define y

**Why Must Variables Be Declared?**

1. Type Safety:

- Java is a strongly typed language, meaning every variable must have a defined data type before use.
- Without declaring y, the compiler does not know if it's an int, double, String, etc.

2. Memory Allocation:

- When a variable is declared, Java allocates memory for it.
- Since y is missing, the program does not know how much memory to allocate.

3. Code Readability & Maintainability:

- Declaring variables makes the code clear and easier to understand.
- Uninitialized or undeclared variables can lead to logical errors and unexpected behavior.

## Corrected Code

```
public class Main {
	public static void main(String[] args) {
		int y = 5;  // Declaring and initializing y
		int x = y + 10;
		System.out.println(x); // Output: 15
	}
}
```

## Snippet 7:

```
public class Main {
        public static void main(String[] args) {
                int x = "Hello";
                System.out.println(x);
        }
}
```

**What compilation error do you see? Why does Java enforce type safety?**

## Compilation

```
Main7.java:3: error: incompatible types: String cannot be converted to int
int x = "Hello";
        ^
1 error
```

**Why Does Java Enforce Type Safety?**

1. Prevents Unexpected Behavior:

- Strong type-checking ensures that operations are performed on compatible data types.
- Without type safety, you might accidentally assign the wrong type, leading to unpredictable errors.

2. Early Error Detection:

- Type mismatches are caught at compile time rather than runtime, making debugging easier.
- This reduces runtime crashes and makes programs more stable.

3. Memory Efficiency:

- Java allocates memory based on data types (int takes 4 bytes, while String is an object with variable length).
- Assigning incompatible types can cause inefficient memory usage and logical errors.

4. Code Readability & Maintainability:

- Explicitly declaring types makes it easier for developers to understand what kind of data a variable holds.
- Helps prevent confusion and reduces debugging time.

## Corrected Code

```
public class Main {
    public static void main(String[] args) {
        String x = "Hello";  //  Correct data type
        System.out.println(x);  // Output: Hello
    }
}
```

## Snippet 8:

```
public class Main {
        public static void main(String[] args) {
                System.out.println("Hello, World!"
        }
}
```

### What syntax errors are present? How do they affect compilation?

## Compilation

```
Main8.java:3: error: ')' expected
System.out.println("Hello, World!"
                                ^
1 error

Main8.java:3: error: ';' expected
System.out.println("Hello, World!"
                                ^
1 error
```

**What Syntax Errors Are Present?**

1. Missing Closing Parenthesis )
   The System.out.println method requires a closing ) after "Hello, World!".

2. Missing Semicolon ; at the End of the Statement
   Every Java statement must end with a semicolon (;).

**How Do These Errors Affect Compilation?**

- Syntax errors prevent the code from compiling. The Java compiler does not allow incomplete or incorrect syntax.
- The program will not run until the errors are fixed.
- The compiler stops at the first detected error, meaning it won't check for further issues until the first one is resolved.

## Corrected Code

```
public class Main {
   public static void main(String[] args) {
      System.out.println("Hello, World!");  // Correct syntax
   }
}
```

## Snippet 9:

```
public class Main {
	public static void main(String[] args) {
		int class = 10;
		System.out.println(class);
	}
}
```

**What error occurs? Why can't reserved keywords be used as identifiers?**

## Compilation

```
E:\CDAC FEB 25\Java Logic Building\Day2\Assignment2>javac Main9.java
Main9.java:3: error: not a statement
		int class = 10;
		^
Main9.java:3: error: ';' expected
		int class = 10;
		  ^
Main9.java:3: error: <identifier> expected
		int class = 10;
			^
Main9.java:4: error: <identifier> expected
		System.out.println(class);
				   ^
Main9.java:4: error: illegal start of type
		System.out.println(class);
				    ^
Main9.java:4: error: <identifier> expected
		System.out.println(class);
					^
Main9.java:6: error: reached end of file while parsing
}
 ^
7 errors
```

**Why Does This Error Occur?**

- **class** is a reserved keyword in Java.
- Reserved keywords cannot be used as variable names, method names, or class names because they have special meanings in the Java language.
- In this case, **class** is used to define a Java class, so using it as a variable name creates confusion for the compiler.

**Why Can't Reserved Keywords Be Used as Identifiers?**

1. Avoids Ambiguity:

- Java keywords have predefined meanings (e.g., class defines a class, int defines an integer type).
- If allowed as variable names, the compiler would be unable to distinguish between a keyword's intended purpose and its use as an identifier.

2. Ensures Code Readability & Maintainability:

- Keywords are widely recognized across all Java programs.
- Allowing them as identifiers would make the code confusing and harder to understand.

3. Prevents Syntax Errors:

- Using reserved words in unexpected ways could lead to conflicts in the Java compiler, causing parsing errors.

## Corrected Code

```
public class Main {
    public static void main(String[] args) {
        int classNum = 10;  // Valid variable name
        System.out.println(classNum);
    }
}
```

---

## Snippet 10:

```
public class Main {
	public void display() {
		System.out.println("No parameters");
	}
	public void display(int num) {
		System.out.println("With parameter: " + num);
	}
	public static void main(String[] args) {
		display();
		display(5);
	}
}
```

**What happens when you compile and run this code? Is method overloading allowed?**

```
Main10.java:1: error: class Main is public, should be declared in a file named Main.java
public class Main {
       ^
Main10.java:9: error: non-static method display() cannot be referenced from a static context
display();
^
Main10.java:10: error: non-static method display(int) cannot be referenced from a static
context
display(5);
^
3 errors
```

**Why Do These Errors Occur?**

1. Non-static Methods Cannot Be Called from a Static Context

- The display() and display(int num) methods are not static.
- The main method is static, meaning it belongs to the class rather than an instance of the class.
- To call non-static methods inside a static method, you need to create an instance of the class.

**Is Method Overloading Allowed?**

1. Yes, method overloading is allowed in Java.
2. Method overloading occurs when multiple methods have the same name but different parameters (type, number, or both).
3. In this case:

- display() has no parameters.
- display(int num) has one integer parameter. The correct method is selected based on the arguments passed during the function call (this is known as compile-time polymorphism).

## Corrected Code

```
public class Main {
    public void display() {
        System.out.println("No parameters");
    }

    public void display(int num) {
        System.out.println("With parameter: " + num);
    }

    public static void main(String[] args) {
        Main obj = new Main();  // Create an instance of Main
        obj.display();          // Call instance method
        obj.display(5);         // Call overloaded method
    }
}
```

## Output

```
No parameters
With parameter: 5
```

# Snippet 11:

```
public class Main {
	public static void main(String[] args) {
		int[] arr = {1, 2, 3};
		System.out.println(arr[5]);
	}
}
```

## What runtime exception do you encounter? Why does it occur?

**Comilation of above code snipet was sucessful.**

**While execution we get error.**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of
bounds for length 3
	at Main11.main(Main11.java:4)
```

**Why Does This Error Occur?**

1. The array arr is declared and initialized as:
   int[] arr = {1, 2, 3};

This means:

```
Index:  0   1   2
Values: 1   2   3
```

- The valid indices for this array are 0, 1, and 2 (since array indices in Java start from 0).
- arr[5] tries to access index 5, which is out of bounds.

In Java, accessing an invalid array index results in an ArrayIndexOutOfBoundsException.

## Snippet 12:

```
public class Main {
	public static void main(String[] args) {
		while (true) {
			System.out.println("Infinite Loop");
		}
	}
}
```

**What happens when you run this code? How can you avoid infinite loops?**

- When we Run this code it will print “Infinite Loop” Infinite times.
- This happened Due to condition given to the While loop is boolean true, so the condition will never be false and it will print Infinite loop many times.
- To avoid infinite loop We have to give  a correct condition which will break a loop for finite times.

```
int count = 0;
while (count < 5) {
   System.out.println("Loop iteration: " + count);
   count++;  //  Increases count to avoid infinite loop
}
```

- Use a for Loop Instead For loops provide a natural way to avoid infinite loops.

```
for (int i = 0; i < 5; i++) {
   System.out.println("Loop iteration: " + i);
}
```

- Use a break Statement If you want to break the infinite loop based on a condition:

```
int count = 0;
while (true) {
   System.out.println("Iteration: " + count);
   if (count == 5) {
      break;  // Exits loop when count reaches 5
   }
   count++;
}
```

# Snippet 13:

```
public class Main {
        public static void main(String[] args) {
                String str = null;
                System.out.println(str.length());
        }
}
```

## What exception is thrown? Why does it occur?

```
Exception in thread "main" java.lang.NullPointerException
        at Main13.main(Main13.java:4)
```

1. **String str = null;**
- This means str does not reference any actual object in memory.

2. **Calling str.length()**
- Since str is null, Java cannot access its length() method.
- This results in a NullPointerException (NPE), which occurs whenever a method is called on a null reference.

## How to Fix This?

**Check for null Before Calling Methods**

```
if (str != null) {
   System.out.println(str.length());  // ✓ Executes only if str is not null
} else {
   System.out.println("String is null!");
}
```

**Assign a Default Value**

```
String str = "";  // Empty string instead of null
System.out.println(str.length());  //Output: 0
```

**Use Optional (Java 8+)**

**A better approach for handling potential null values:**

```
import java.util.Optional;

public class Main {
   public static void main(String[] args) {
      Optional<String> str = Optional.ofNullable(null);
      System.out.println(str.map(String::length).orElse(0));  // ✓ Safe handling
   }
}
```

# Snippet 14:

```
public class Main {
	public static void main(String[] args) {
		double num = "Hello";
		System.out.println(num);
	}
}
```

## What compilation error occurs? Why does Java enforce data type constraints?

```
Main14.java:3: error: incompatible types: String cannot be converted to double
double num = "Hello";
	^
1 error
```

**Why does Java enforce data type constraints?**

- Java enforces strict data type constraints to prevent runtime errors.
- Strings cannot be directly assigned to numeric types (int, double, etc.).
- Use explicit conversion (Double.parseDouble()) if a number is stored as a String.

## Snippet 15:

```
public class Main {
	public static void main(String[] args) {
		int num1 = 10;
		double num2 = 5.5;
		int result = num1 + num2;
		System.out.println(result);
	}
}
```

## What error occurs when compiling this code? How should you handle different data types in operations?

```
Main15.java:5: error: incompatible types: possible lossy conversion from double to int
int result = num1 + num2;
             ^
1 error
```

**Why Does This Error Occur?**

1. Data Type Conversion Rules in Java:

- num1 is an int (10).
- num2 is a double (5.5).
- Java automatically promotes int to double during arithmetic operations, so num1 + num2 results in a double (15.5).
- However, assigning a double value (15.5) to an int variable (result) is not allowed without explicit conversion because it loses precision (fractional part).

2. Java does not allow implicit conversion from double to int because it could lead to data loss.

**Explicitly Cast double to int (Risky)**

```
public class Main {
	public static void main(String[] args) {
		int num1 = 10;
		double num2 = 5.5;
		int result = (int)(num1 + num2); //Explicit type casting
		System.out.println(result);
	}
}
```

**Output**

```
15  // Fractional part (0.5) is lost
```

**Use Math.round() for Proper Rounding**
**If you want to round the value instead of truncating:**

```
int result = (int) Math.round(num1 + num2);  // Rounds before conversion
System.out.println(result);
```

## Output:

```
16  // 15.5 rounded to 16
```

## Snippet 16:

```
public class Main {
	public static void main(String[] args) {
		int num = 10;
		double result = num / 4;
		System.out.println(result);
	}
}
```

## What is the result of this operation? Is the output what you expected?

**Expected vs Actual**

**Actual Output in Java**
2.0

**Expected Output (If we assume normal division)**
2.5

**Why Does This Happen?**
Integer Division Rule in Java:

- num is an int (10).
- 4 is also an int.
- When both operands are int, Java performs integer division, which discards the decimal part.
- 10 / 4 = 2.5, but since both numbers are integers, Java truncates the decimal part, so it results in 2.
- Even though result is a double, the integer division has already happened before assignment, so 2 is stored as 2.0.

**How to Fix This?**

1. Use a double in the Division
   To get the correct decimal result, make at least one operand double:

   ```
   double result = num / 4.0;  // One operand is double
   System.out.println(result);
   ```

   **Output**
   2.5

2. Explicit Type Casting
   Convert one of the numbers to double:

   ```
   double result = (double) num / 4;  // Cast to double
   System.out.println(result);
   ```

   **Output:**
   2.5

# Snippet 17:

```
public class Main {
        public static void main(String[] args) {
        int a = 10;
        int b = 5;
        int result = a ** b;
        System.out.println(result);
        }
}
```

## What compilation error occurs? Why is the ** operator not valid in Java?

```
Main17.java:5: error: illegal start of expression
int result = a ** b;
             ^
1 error
```

**Why Does This Happen?**

1. Java Does Not Support ** for Exponentiation

- In some languages like Python, ** is used for exponentiation (a ** b means a raised to the power of b).
- Java does not recognize ** as a valid operator, causing a compilation error.

2. How to Perform Exponentiation in Java?

- Java provides the Math.pow() method for exponentiation:

  **double result = Math.pow(a, b);**

  Math.pow(a, b) calculates a^b and returns a double.

# Corrected Code

```
public class Main {
   public static void main(String[] args) {
      int a = 10;
      int b = 5;
      double result = Math.pow(a, b);  // Correct exponentiation
      System.out.println(result);
   }
}
```

## Snippet 18:

```
public class Main {
        public static void main(String[] args) {
        int a = 10;
        int b = 5;
        int result = a + b * 2;
        System.out.println(result);
}
}
```

**What is the output of this code? How does operator precedence affect the result?**

### Output:
20

### Operator Precedence in Java
- Multiplication (*), Division (/), and Modulus (%) have higher precedence than Addition (+) and Subtraction (-).
- Expressions inside parentheses () are evaluated first.
- If operators have the same precedence, evaluation is from left to right.

# Snippet 19:

```
public class Main {
	public static void main(String[] args) {
		int a = 10;
		int b = 0;
		int result = a / b;
		System.out.println(result);
	}
}
```

## What runtime exception is thrown? Why does division by zero cause an issue in Java?

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Main19.main(Main19.java:5)
```

### Why Does This Happen?

1. In Java, division by zero (/ 0) using integers is not allowed because mathematically, division by zero is undefined.
2. The Java Virtual Machine (JVM) detects this issue and throws an ArithmeticException at runtime.
3. Floating-point division (double or float) behaves differently—instead of an exception, it results in Infinity or NaN (Not a Number).

### How to Handle Division by Zero?

**1. Check for zero before division:**

```
if (b != 0) {
    int result = a / b;
    System.out.println(result);
} else {
    System.out.println("Division by zero is not allowed!");
}
```

**2. Use try-catch to handle exceptions:**

```
try {
    int result = a / b;
    System.out.println(result);
} catch (ArithmeticException e) {
    System.out.println("Error: Division by zero is not allowed!");
}
```

**3. For floating-point division, Java allows division by zero:**

```
double result = 10.0 / 0.0;  // Output: Infinity
System.out.println(result);
```

## Snippet 20:

```
public class Main {
	public static void main(String[] args) {
		System.out.println("Hello, World")
	}
}
```

**What syntax error occurs? How does the missing semicolon affect compilation?**

### Compileation Error

```
Main20.java:3: error: ';' expected
System.out.println("Hello, World")
                                ^
1 error
```

### Why Does This Happen?

1. Java requires a semicolon (;) at the end of every statement.

The System.out.println("Hello, World") statement is missing a semicolon.

2. The compiler expects a semicolon (;) to mark the end of the statement.

Without it, the compiler gets confused about where the statement ends, leading to a syntax error.

### Corrected code

```
public class Main {
	public static void main(String[] args) {
		System.out.println("Hello, World");
	}
}
```

## Snippet 21:

```
public class Main {
	public static void main(String[] args) {
		System.out.println("Hello, World!");
		// Missing closing brace here
}
```

## What does the compiler say about mismatched braces?

```
Main21.java:5: error: reached end of file while parsing
}
 ^
1 error
```

## Why Does This Happen?

1. Java expects every opening brace { to have a matching closing brace }.

- The class Main starts with {, but it is never properly closed.
- The main method also starts with {, but the program ends before closing it.

2. The compiler reads the file until the end and realizes that a closing brace } is missing.

- This causes the error "reached end of file while parsing", meaning the compiler was expecting more code before the file ended.

## Corrected Code

```
public class Main {
	public static void main(String[] args) {
		System.out.println("Hello, World!");
		// Missing closing brace here
	}
}
```

## Snippet 22:

```
public class Main {
	public static void main(String[] args) {
		static void displayMessage() {
		System.out.println("Message");
		}
	}
}
```

### What syntax error occurs? Can a method be declared inside another method?

```
Main22.java:3: error: illegal start of expression
static void displayMessage() {
^
Main22.java:7: error: class, interface, or enum expected
}
^
2 errors
```

**Why Does This Happen?**

1. Methods Cannot Be Declared Inside Other Methods

- In Java, you cannot declare a method inside another method.
- The displayMessage() method is declared inside main(), which is not allowed.

2. The static Keyword is Misplaced

- The static keyword is used to declare a method inside a class, not inside another method.

## Corrected code

```
public class Main {
   public static void displayMessage() {  // Correct placement
      System.out.println("Message");
   }

   public static void main(String[] args) {
      displayMessage(); // Calling the method correctly
   }
}
```

## Snippet 23:

```
public class Main{
        public static void main(String[] args) {
                int value = 2;
                switch(value) {
                        case 1:
                                System.out.println("Value is 1");
                        case 2:
                                System.out.println("Value is 2");
                        case 3:
                                System.out.println("Value is 3");
                        default:
                                System.out.println("Default case");
                }
        }
}
```

**Error to Investigate: Why does the default case print after "Value is 2"?**

**How can you prevent the program from executing the default case**

### Output

Value is 2
Value is 3
Default case

### Why Does This Happen?

The issue here is missing break statements in the switch cases.

- When value = 2, execution starts from case 2.
- Since there is no break statement, the program falls through to the next cases (case 3 and default), executing them even if they don't match.

### Corrected Code

```
public class Main{
   public static void main(String[] args) {
      int value = 2;
      switch(value) {
         case 1:
            System.out.println("Value is 1");
            break;  //  Prevents fall-through
         case 2:
            System.out.println("Value is 2");
            break;  //  Stops execution here
         case 3:
            System.out.println("Value is 3");
            break;  // Stops execution here
         default:
            System.out.println("Default case");
      }
   }
}
```

## Snippet 24:

```
public class Main{
        public static void main(String[] args) {
                int level = 1;
                switch(level) {
                case 1:
                        System.out.println("Level 1");
                case 2:
                        System.out.println("Level 2");
                case 3:
                        System.out.println("Level 3");
                default:
                        System.out.println("Unknown level");
                }
        }
}
```

**Error to Investigate: When level is 1, why does it print "Level 1", "Level 2", "Level 3", and "Unknown level"? What is the role of the break statement in this situation?**

### Output

Level 1
Level 2
Level 3
Unknown level

### Why Does This Happen?

The issue here is missing break statements in the switch cases.

- When value = 2, execution starts from case 2.
- Since there is no break statement, the program falls through to the next cases (case 3 and default), executing them even if they don't match.

### Corrected Code

```
public class Main {
   public static void main(String[] args) {
      int value = 1;
      switch(value) {
         case 1:
            System.out.println("Level 1");
            break;  //  Prevents fall-through
         case 2:
            System.out.println("Level 2");
            break;  //  Stops execution here
         case 3:
            System.out.println("Level 3");
            break;  // Stops execution here
         default:
            System.out.println("Unknown level");
      }
   }
}
```

## Snippet 25:

```
public class Main{
        public static void main(String[] args) {
                double score = 85.0;
                switch(score) {
                        case 100:
                                System.out.println("Perfect score!");
                                break;
                        case 85:
                                System.out.println("Great job!");
                                break;
                        default:
                                System.out.println("Keep trying!");
                }
        }
}
```

**Error to Investigate: Why does this code not compile? What does the error tell you about the types allowed in switch expressions? How can you modify the code to make it work?**

Compilation Error

```
Main25.java:4: error: incompatible types: possible lossy conversion from double to int
switch(score) {
    ^
1 error
```

### Why Does This Happen?

1. The switch statement in Java does not support double (floating-point) values as the expression type.
2. In Java, switch only works with the following data types:

- byte, short, char, int
- String (Java 7+)
- enum
- Wrapper classes (Byte, Short, Character, Integer, String)
- Not double or float, because of precision issues with floating-point numbers.

### Corrected Code

```
public class Main{
   public static void main(String[] args) {
      int score = 85;  // Changed double to int
      switch(score) {
         case 100:
            System.out.println("Perfect score!");
            break;
         case 85:
            System.out.println("Great job!");
            break;
         default:
            System.out.println("Keep trying!");
      }
   }
}
```

## Snippet 26:

```
public class Main{
	public static void main(String[] args) {
		int number = 5;
		switch(number) {
			case 5:
				System.out.println("Number is 5"); break;
			case 5:
				System.out.println("This is another case 5");
				break;
			default:
				System.out.println("This is the default case");
		}
	}
}
```

**Error to Investigate: Why does the compiler complain about duplicate case labels? What happens when you have two identical case labels in the same switch block?**

### Compilation Error

```
Main26.java:7: error: duplicate case label
case 5:
^
1 error
```

### Why Does This Happen?

- case labels must be unique within a switch statement.
- The compiler detects that case 5 appears twice, which is not allowed because switch needs distinct labels to determine which block to execute.
- If duplicates were allowed, the program wouldn't know which case to execute first.

### Corrected Code

```
public class Main{
   public static void main(String[] args) {
      int number = 5;
      switch(number) {
         case 5:
            System.out.println("Number is 5");
            break;
         default:
            System.out.println("This is the default case");
      }
   }
}
```