CSE 1115 · OBJECT-ORIENTED PROGRAMMING

Java Programming Fundamentals

Lecture 02 — program structure, syntax, data types, strings, and input/output

Main.java

```java
public class Main {

    public static void main(String[] args) {

        System.out.println("Hello, World!");

    }

}
```



How today's first program travels from text to a running program

Sheikh Tanvir Ahmed
Lecturer, Department of Computer Science and Engineering
United International University

TODAY'S ROADMAP

1. Java at a glance: source files, naming, program structure
2. The main method and basic syntax
3. Compiling and running: javac, bytecode, the JVM
4. Variables and the eight primitive types
5. Literals and Strings
6. Input and output with Scanner and printf

Where we left off

From C to Java

The same building blocks, variables, conditions and loops, with a different way of organizing a program.

Why OOP

Large programs stay manageable when data and the behavior that works on it are kept together.

The shift

A Java program is built from classes. Functions become methods that live inside them.

Now that we understand why we are learning Java, how do we actually write a Java program?

TODAY'S ROUTE



Program structure — **Compile and run** — **Variables and types** — **Strings** — **Input and output**

What kind of language is Java?

Characteristic	What it means	Seen in code
High-level	Readable text; no registers or raw memory addresses to manage	`System.out.println("Hi");`
Class-based	Code lives inside classes, the basic unit of a Java program	`public class Main { }`
Object-oriented	Programs model things as objects that combine data and behavior	`Student s = new Student();`
Strongly typed	Every variable has a declared type, checked by the compiler	`int age = "twenty"; // error`
Automatic memory management	A garbage collector reclaims objects that are no longer used	`new Student();  // no free()`
Platform-independent bytecode	Compiled once to bytecode that any Java Virtual Machine can run	`Main.class`

Observe Java is not purely object-oriented: the eight primitive types (`int`, `double`, …) are not objects. We meet them shortly.

FILES

Source files and bytecode files



Source code: text you write and edit

Bytecode: binary produced by the compiler

Extension	Holds	Created by	Used by
`.java`	Source code (plain text)	You, in an editor or IDE	The compiler, javac
`.class`	Compiled bytecode (binary)	The compiler, javac	The JVM, started with java

FILE NAME EXAMPLES: ONE PUBLIC CLASS PER FILE

`Main.java` → `class Main` `Student.java` → `class Student` `HelloWorld.java` → `class HelloWorld`

> **CASE SENSITIVITY**
>
> `.java` is always lowercase. To the Java compiler `Main.java` and `main.java` are different names, and so are `Student` and `student`.

Naming conventions

Element	Convention	Correct	Discouraged
Class	PascalCase: each word capitalized	`StudentInformation`	`studentinformation`, `student_information`
Variable	camelCase: first word lowercase	`studentName`	`StudentName`, `student_name`
Method	camelCase, usually a verb	`calculateTotal()`	`CalculateTotal()`, `calculate_total()`
Constant	UPPER_SNAKE_CASE	`MAX_VALUE`	`maxValue`, `Max_Value`
Package	all lowercase	`com.uiu.cse1115`	`Com.UIU.CSE1115`

RULES: ENFORCED BY THE COMPILER

An identifier cannot start with a digit, contain spaces, or be a reserved word such as `class` or `int`. Break a rule and the program does not compile.

CONVENTIONS: AGREED STYLE

`class studentinformation { }` compiles, but every Java programmer, and most tools, expect `StudentInformation`. Conventions make code readable to others.

Anatomy of a Java program

```
public class Main {

    public static void main(String[] args) {

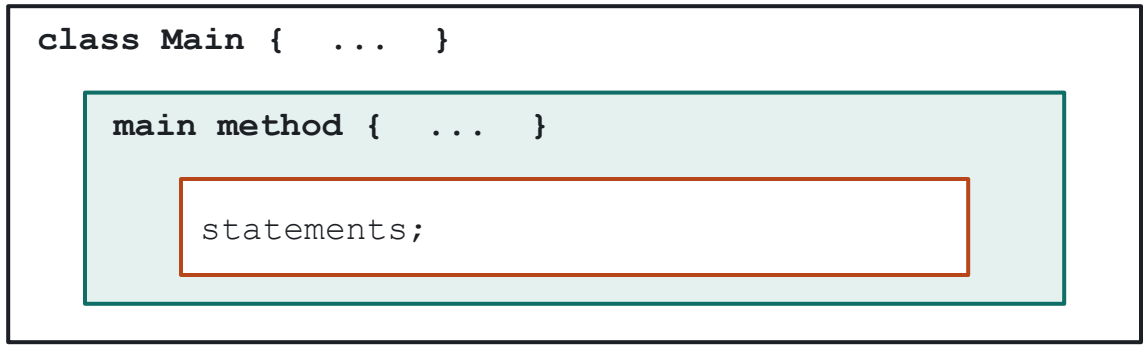
        System.out.println("Hello, World!");

    }

}
```

1. **Class declaration.** `public class Main` names the class; it is the container for everything below.
2. **Braces.** The opening `{` and its closing `}` mark the start and end of the class body.
3. **Main method.** The entry point: where the program starts executing.
4. **Statement.** One instruction. It ends with a semicolon `;`.
5. **Indentation.** Each block is indented one level (4 spaces) so structure is visible at a glance.

BLOCKS NEST INSIDE BLOCKS



- Every statement ends with `;`
- Braces always come in pairs
- Statements go inside a method; methods go inside a class
- Blank lines and indentation help people, not the compiler

The filename must match the public class

File	Declaration inside	Result
`Main.java`	`public class Main`	✓ compiles
`main.java`	`public class Main`	✗ error
`Test.java`	`public class Main`	✗ error

WHAT THE COMPILER SAYS

```
$ javac main.java
main.java:1: error: class Main is public, should be declared in
a file named Main.java
public class Main {
       ^
```

A public class must live in a file with exactly its name, capitalization included, plus `.java`.

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Helper.greet();
    }
}

class Helper {
    static void greet() {
        System.out.println("Hi from Helper");
    }
}
```

One file may contain several classes

but only **one public top-level class**, and it must match the filename. `Helper` has no `public`, so it can have any name.

The compiler writes one `.class` file per class: Helper.class, Main.class.

Understanding the main method

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

1 2 3 4 5 6

	Part	What it means
1	`public`	Access modifier. The Java launcher, which is outside your class, must be able to call this method.
2	`static`	The method belongs to the class itself, so it can run without creating an object first. At startup no object exists yet.
3	`void`	The method returns no value to its caller.
4	`main`	The method name the launcher looks for. It is case-sensitive: `Main` or `MAIN` will not work.
5	`String[]`	The parameter's type: an array of references to `String` objects.
6	`args`	The parameter's name (any valid identifier works). It holds command-line arguments: `java Main hi` gives `args[0]` = "hi".

WHY INSIDE A CLASS?

Java has no stand-alone functions. Every method belongs to a class. Running `java Main` loads the class `Main`, then calls its `main` method.

METHOD VS CLASS

A **class** is a blueprint that groups data and methods into one type. A **method** is a named block of statements inside a class that performs one action.

SYNTAX

Java syntax and coding conventions

Greeter.java

```
// Single-line comment: runs to the end of the line

/* Multiline comment:
   it can span several lines */

/**
 * Documentation comment: read by the javadoc tool
 * to generate API documentation.
 */
public class Greeter {
    public static void main(String[] args) {
        int count = 3;
        if (count > 2) {
            System.out.println("Count is large");
        }
    }
}
```

Symbol	Purpose	Example
`;`	Ends a statement	`int count = 3;`
`{ }`	Define a block of code	`class, method, if, loop bodies`
`( )`	Parameters, calls, conditions	`main(String[] args), if (x > 2)`
`[ ]`	Array types and indexing	`String[] args, args[0]`

Comment	Use
`// text`	Single-line: runs to the end of the line
`/* text */`	Multiline: everything up to the closing */
`/** text */`	Documentation: read by the javadoc tool

Observe Indentation and spacing make code readable, but braces define the blocks. Java is case-sensitive: `Main` and `main`, `System` and `system` are different.

EXECUTION

How a Java program runs



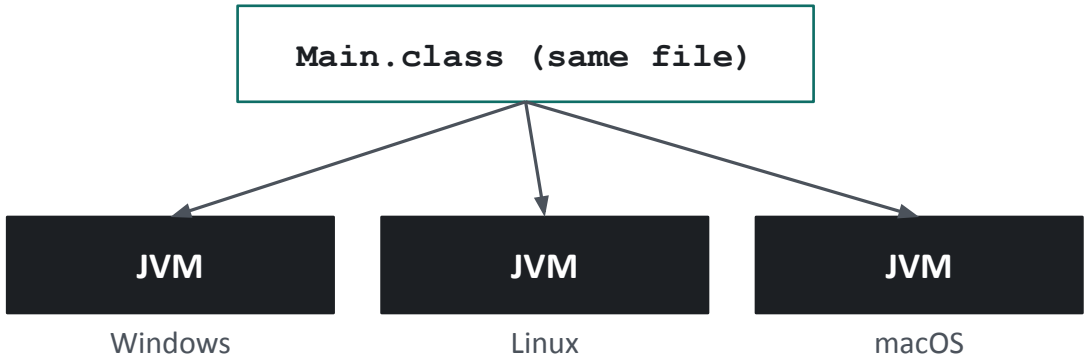
JDK — Java Development Kit: the tools to write and run Java, including `javac`, `java` and the standard libraries.

javac — Checks syntax and types and translates `.java` into bytecode. It does not run the program.

Bytecode — Platform-neutral instructions stored in `.class` files. It is not machine code for any one CPU.

JVM — Loads bytecode and executes it on the real machine. Each platform has its own JVM.

PLATFORM INDEPENDENCE



Compile once, run on any platform that has a JVM.

INTERPRETATION AND JIT

The JVM first interprets bytecode. Code that runs often is compiled by the JIT (just-in-time) compiler into machine code while the program runs.

Variables in Java

DECLARE, INITIALIZE, ASSIGN

```
int age;                // declaration
age = 20;               // assignment
int year = 2026;        // declaration + initialization
age = age + 1;          // assignment again
double cgpa = 3.75;
char grade = 'A';
boolean passed = true;
```

IDENTIFIER RULES

- Letters, digits, `_` and `$`; cannot start with a digit
- No spaces, no reserved words (`class`, `int`, `new`...)
- Case-sensitive: `age` and `Age` are different variables
- Convention: camelCase, meaningful names, `studentName` not `sn`

STRONG TYPING

```
int age = 20;
age = "twenty";
int total;
System.out.println(total);
```

`age = "twenty";` error: incompatible types: String cannot be converted to int
`println(total);` error: variable total might not have been initialized

	C	Java
Declaration	`int age = 20;`	`int age = 20;`
Unassigned local	`holds garbage`	`compile error`
Constant	`const int MAX = 10;`	`final int MAX = 10;`
Boolean	`int (0 is false)`	`boolean: true or false`
`int` size	`depends on platform`	`always 32 bits`

The eight primitive data types

Category	Type	Size	Range	Default (field)	Example
Integer	`byte`	8 bits	−128 to 127	`0`	`byte b = 100;`
Integer	`short`	16 bits	−32,768 to 32,767	`0`	`short s = 30000;`
Integer	`int`	32 bits	−2,147,483,648 to 2,147,483,647	`0`	`int i = 2_000_000_000;`
Integer	`long`	64 bits	-2^{63} to $2^{63} - 1$ (about $\pm 9.2 \times 10^{18}$)	`0L`	`long l = 9_000_000_000L;`
Floating-point	`float`	32 bits	about $\pm 3.4 \times 10^{38}$, 6–7 digits	`0.0f`	`float f = 3.14f;`
Floating-point	`double`	64 bits	about $\pm 1.8 \times 10^{308}$, 15–16 digits	`0.0`	`double d = 3.14159;`
Character	`char`	16 bits	0 to 65,535 (a UTF-16 code unit)	`'\u0000'`	`char c = 'J';`
Logical	`boolean`	not specified	true or false	`false`	`boolean flag = true;`

PRIMITIVE VS REFERENCE

A primitive variable holds the **value itself**. `String`, arrays and class types are **reference types**: the variable holds a reference to an object.

DEFAULT VALUES

Defaults apply to **fields** only. A local variable has no default and must be assigned before use.

Literals and type considerations

Kind	Examples	Notes
Integer	`42` `0b1010` `0x1F` `1_000_000`	`int` by default; add `L` for `long`: 3000000000L
Floating-point	`3.14` `2.5e3` `3.14f`	`double` by default; add `f` or `F` for `float`
Character	`'A'` `'\n'` `'\u0041'`	Single quotes; one UTF-16 character
Boolean	`true` `false`	Lowercase words only, never 1 or 0

JAVA

```java
int count = 42;
long big = 3000000000L;
float price = 3.14f;
double pi = 3.14;
char letter = 'A';
boolean done = true;
System.out.println(1.0f / 3);
System.out.println(1.0 / 3);
```

WHY float AND double BEHAVE DIFFERENTLY

`3.14` is a **double** (64 bits). Storing it in a `float` (32 bits) could lose precision, so Java refuses unless you write `3.14f`.

`float f = 3.14;` error: incompatible types: possible lossy conversion from double to float

`1.0f / 3` prints 0.33333334 (about 7 digits)
`1.0 / 3` prints 0.3333333333333333 (about 16 digits)

Strings in Java

C

```c
char name[20] = "Tanvir";
char msg[40] = "Hello ";
strcat(msg, name);
printf("%zu\n", strlen(msg));
```

fixed size, terminated by '\0', library functions

JAVA

```java
String name = "Tanvir";
String message = "Hello " + name;
System.out.println(message);
System.out.println(message.length());
```

prints: Hello Tanvir then 12

WHAT THE VARIABLES HOLD



A primitive variable holds its value directly. A `String` variable holds a **reference** to a `String` object stored elsewhere.

- `String` is a class from `java.lang`; no import is needed
- Text goes in **double quotes**; a `char` uses single quotes
- `+` joins strings; joining with a number converts it to text
- `message.length()` returns the number of characters
- Compare text with `equals()`, not `==` (we return to this later)

Output with print, println, and printf

JAVA

```
System.out.print("Java ");
System.out.print("rocks");
System.out.println();
System.out.println("CSE " + 1115);
System.out.printf("CGPA: %.2f%n", 3.756);
System.out.printf("%s is %d years old%n", "Arif", 20);
System.out.printf("Grade: %c, passed: %b%n", 'A', true);
```

OUTPUT

```
Java rocks
CSE 1115
CGPA: 3.76
Arif is 20 years old
Grade: A, passed: true
```

`print` stays on the line, `println` ends it, `printf` formats the output.

Specifier	Prints	Example
`%d`	int, long, short, byte	`%d → 20`
`%f`	float or double (6 decimals)	`%f → 3.756000`
`%.2f`	decimals rounded to 2 places	`%.2f → 3.76`
`%s`	a String (any object)	`%s → Arif`
`%c`	a single char	`%c → A`
`%b`	a boolean	`%b → true`
`%n`	a line break (portable)	`like \n`

COMPARED WITH C

C: `printf("%.2f\n", x);`

Java: `System.out.printf("%.2f%n", x);`

Same specifiers. Java uses `%n` for newline, one `%f` serves float and double, and a wrong specifier throws an exception at run time.

Input with the Scanner class

Main.java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

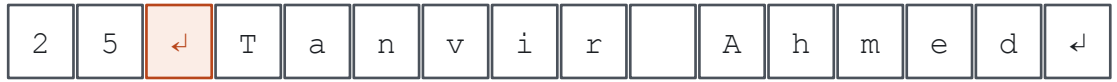
        int age = input.nextInt();
        double cgpa = input.nextDouble();
        String word = input.next();
        String line = input.nextLine();

        input.close();
    }
}
```

Method	Reads
`nextInt()`	the next token as an int
`nextLong()`	the next token as a long
`nextFloat()`	the next token as a float
`nextDouble()`	the next token as a double
`next()`	one word, up to whitespace
`nextLine()`	the rest of the line, spaces included

C: `scanf("%d", &age);` Java: `age = input.nextInt();` No `&`, no format string; one method per type.

THE NEWLINE TRAP



`nextInt()` reads `25` but leaves the newline ↵ in the buffer. The next `nextLine()` reads from there to the end of the line and returns an **empty string**.

`Name read: []` → empty! (the buggy version)

Fix: call `input.nextLine();` once after `nextInt()` to consume the leftover newline, or read every value with `nextLine()` and convert it.

Result after the fix: `Name read: [Tanvir Ahmed]`

A complete input/output program

StudentInfo.java

```
import java.util.Scanner;

public class StudentInfo {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("Enter name: ");
        String studentName = input.nextLine();

        System.out.print("Enter ID: ");
        String studentId = input.next();

        System.out.print("Enter CGPA: ");
        double cgpa = input.nextDouble();

        System.out.println("---- Student Information ----");
        System.out.printf("Name : %s%n", studentName);
        System.out.printf("ID   : %s%n", studentId);
        System.out.printf("CGPA : %.2f%n", cgpa);

        input.close();
    }
}
```

SAMPLE RUN

```
$ java StudentInfo
Enter name: Arif Hasan
Enter ID: 011221001
Enter CGPA: 3.8
---- Student Information ----
Name : Arif Hasan
ID   : 011221001
CGPA : 3.80
```

Things to notice

- `import java.util.Scanner;` comes first, above the class
- The ID is a `String`: leading zeros matter and we never calculate with it
- The name is read first with `nextLine()`, so there is no newline trap
- `%n` ends each line; `%.2f` shows CGPA with two decimals

DEMONSTRATION

Compiling and running from the command line

TERMINAL

```
$ ls
Main.java
$ javac Main.java
$ ls
Main.class  Main.java
$ java Main
Hello, World!
```

Command	What it does
`javac Main.java`	Compiles the source file, reports errors with line numbers, and creates `Main.class`
`java Main`	Starts the JVM, loads the class `Main`, and calls its `main` method. No extension.

Arguments after the class name arrive in `args`:

`java Main Tanvir 42` → `args[0]` is "Tanvir", `args[1]` is "42"

Since Java 11, `java Main.java` compiles and runs in one step. We use the two-step form to see the pipeline.

COMMON MISTAKES, WITH THE REAL MESSAGES

You typed	Result
`javac Main`	error: Class names, 'Main', are only accepted if annotation processing is explicitly requested
`java Main.class`	Error: Could not find or load main class Main.class
`java main`	Error: Could not find or load main class main (names are case-sensitive)

What to take away

KEY TAKEAWAYS

- Source lives in `.java` files; `javac` makes `.class` bytecode; the JVM runs it
- A public class must match its filename, capitalization included
- Classes and variables use PascalCase and camelCase; constants use UPPER_SNAKE_CASE
- Execution starts at `public static void main(String[] args)`
- Eight primitive types; `String` is a reference type
- `float` literals need `f`, `long` literals need `L`
- `printf` formats output; `Scanner` reads input, and `nextLine()` after `nextInt()` needs care

CHECK YOUR UNDERSTANDING

Why must `public class Student` be saved as `Student.java`?

What is the difference between `javac` and `java`?

Why does `float f = 3.14;` not compile?

Is `String` a primitive type? How can you tell?

What does `static` allow in the declaration of `main`?

PRACTICE PROGRAMS

- Print your name, department and university using `println` and `printf`
- Read two integers and print their sum and average with `printf`
- Read a radius and print the circle's area to two decimals
- Read an integer, then a full name with `nextLine()`, without losing the name