Chapter 01

Programming Principles

Introduction to Programming and the AI Context

Mr. SOK Pongsametrey
metreysk@gmail.com

Content

- What is Programming?
- Why Programming Still Matters in the AI Era
- Setting Up Your Java Environment
- Variable & Scope
- Data Type
- Exception
- Class / Interface

What is Programming?

- Giving instructions to a machine to perform tasks
- Writing code that a computer can understand and execute
- Building software that solves real-world problems
- **Real-world results:** ATM systems, online shopping carts, ride-hailing apps, AI chatbots, automated billing

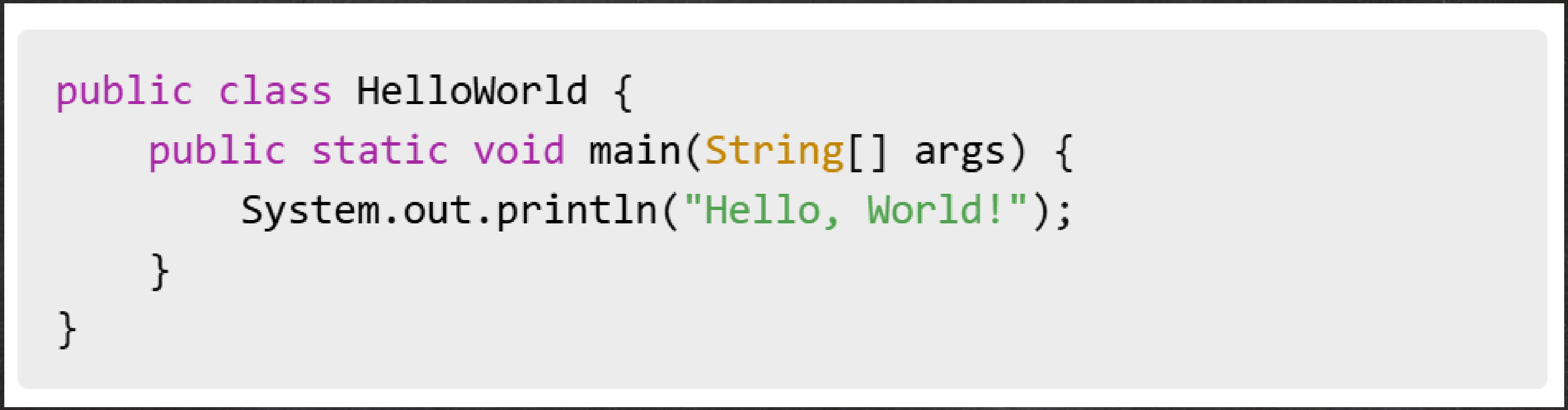
Why Programming Still Matters in the AI Era

- AI generates code, but humans define the problem
- Understanding code = debugging AI-generated results
- Critical thinking and logic are still human strengths
- **Real-world example:** AI may auto-generate website code, but only a human can judge if it meets the business need or is secure

Java Environment Setup

- JDK Installation
- IDEs: IntelliJ IDEA, Eclipse, VS Code
- Create your first "Hello World" Java project
- **Real-world context:** Software developers use IDEs like IntelliJ or Eclipse daily to write, test, and debug applications
- (Detail in another slide)

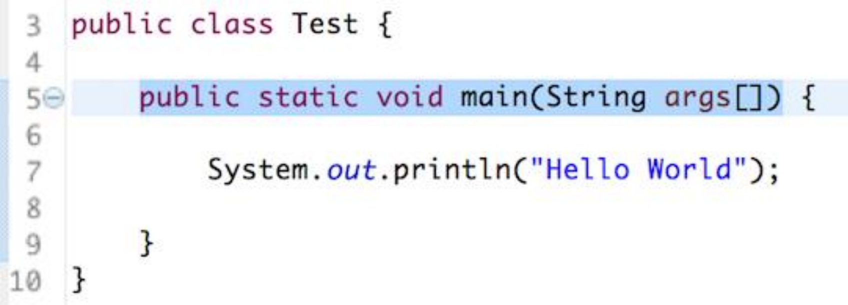
Your Hello World!



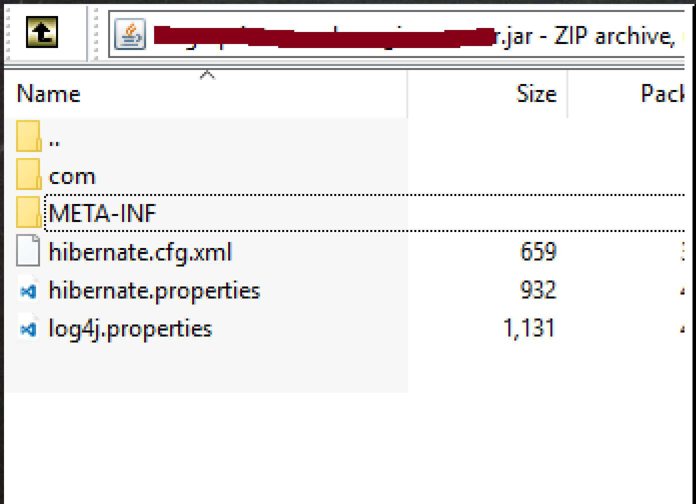
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Java Application Entry Point

- Java main method entry point is always: “*public static void main(String args[])*”
- In jar package, you can find this class in Manifest.txt: Main-Class: MyPackage.MyClass

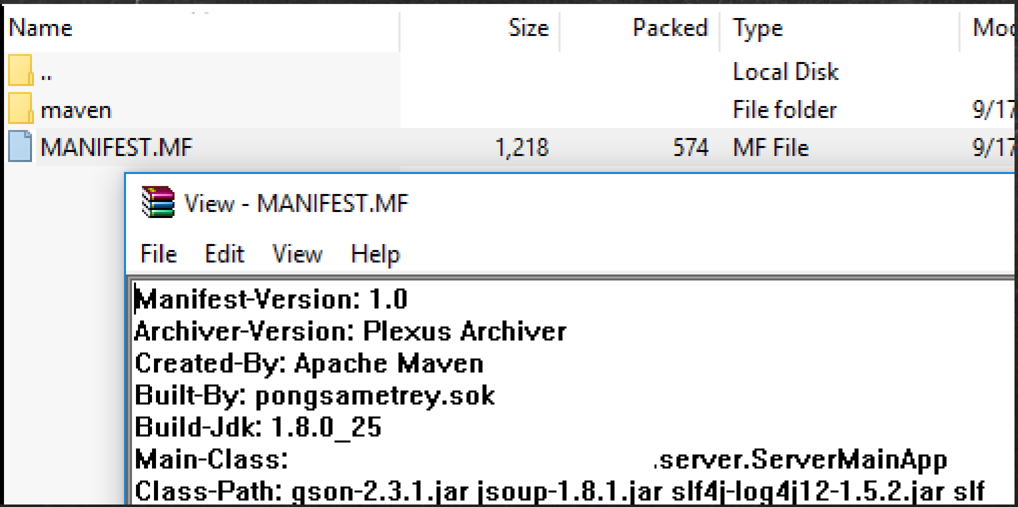


```
public class Test {

    public static void main(String args[]) {

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

    }
}
```



[redacted].jar - ZIP archive,

Name	Size	Pack
..		
com		
META-INF		
hibernate.cfg.xml	659	
hibernate.properties	932	
log4j.properties	1,131	



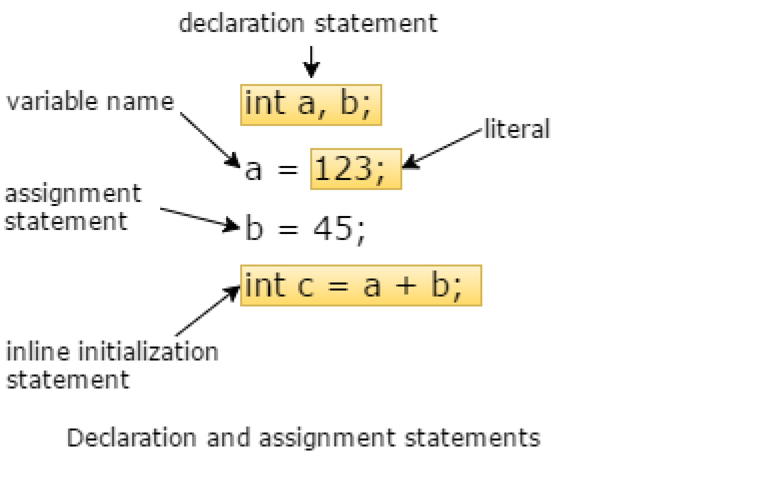
Name	Size	Packed	Type	Mod
..			Local Disk	
maven			File folder	9/17
MANIFEST.MF	1,218	574	MF File	9/17

View - MANIFEST.MF

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: pongsametrey.sok
Build-Jdk: 1.8.0_25
Main-Class:                        .server.ServerMainApp
Class-Path: gson-2.3.1.jar jsoup-1.8.1.jar slf4j-log4j12-1.5.2.jar slf
```

Variable

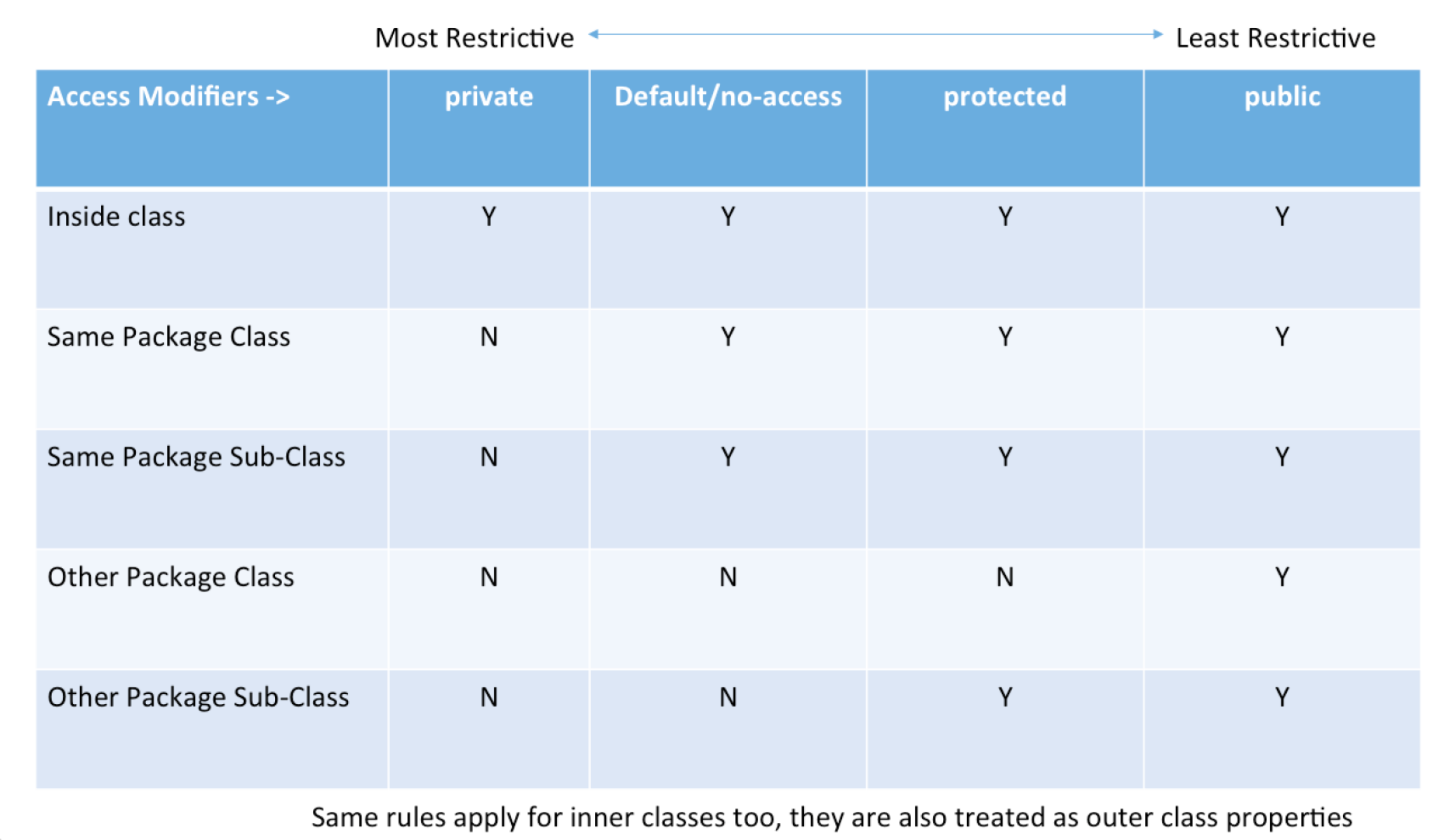
- Every variable must have a data type.
- Two types of data type
  - Primitive
  - Reference
- Scope
  - Private
  - None specifier (or Default)
  - Protected
  - Public



```
int a, b;
a = 123;
b = 45;
int c = a + b;
```

Declaration and assignment statements

Scope (Access Modifiers)



Most Restrictive ↔ Least Restrictive

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Same rules apply for inner classes too, they are also treated as outer class properties

Final Variables

- Declare in any scope
- The value can not be changed
- It is used to declare the "Constant"

```
public static final int JAVA_LESSON = 1;

    public static void main(String[] args) {

            JAVA_LESSON = 2;

    }
```

Initializing Variables

- You need to assign the value for the variable before using it

```
public class testApp { public static void main(String[] args)
        {
                int i;
                System.out.println("The value of i is " + i);
        }
```

Data Type

- Primitive type
- Reference Type

Type	Explanation
int	A 32-bit (4-byte) integer value
short	A 16-bit (2-byte) integer value
long	A 64-bit (8-byte) integer value
byte	An 8-bit (1-byte) integer value
float	A 32-bit (4-byte) floating-point value
double	A 64-bit (8-byte) floating-point value
char	A 16-bit character using the Unicode encoding scheme
boolean	A true or false value

```
Integer
Short
Long …
```

Reference Type

- Based on a class
- Difference from primitive type is memory location
  - Primitive Type -> actual value
  - Reference Type -> address (pointer)

Exception Handling

- Exception
- RuntimeException

Check Exception

Uncheck Exception

```
try {
    statement(s)
} catch (exception type name) {
    statement(s)
} finally {
    statement(s)
}
```

Exception & RuntimeExcetpion

```
public void storeDataFromUrl(String url){
        try {
            String data = readDataFromUrl(url);
        } catch (BadUrlException e) {
            e.printStackTrace();
        }
    }

    public String readDataFromUrl(String url)
    throws BadUrlException{
        if(isUrlBad(url)){
            throw new BadUrl  ception("Bad URL: " + url);
        }

        String data = null;
        //read lots of data over HTT      eturn
        //it as a String instance.

        return data;
    }

    private boolean isUrlBad(String
    {
    return false;
    }
```

Check Exception

```
public class BadUrlException extends Exception {

        public BadUrlException(String message)
        {
                super (message);
        }

}
```

Exception & RuntimeExcetpion

```
public void storeDataFromUrl(String url) {
String data = readDataFromUrl(url);
}

public String readDataFromUrl(String url) throws
BadUrlException{
if (isUrlBad(url)) {
throw new BadUrlException("Bad URL: " + url);
}

String data = null;

return data;
}

private boolean isUrlBad(S
    {
    return false;
    }
```

UnCheck Exception

```
public class BadUrlException extends RunException {

        public BadUrlException(String message)
        {
                super (message);
        }
}
```

Creating and Cleaning up Objects

- Create new object by :
  - **new** operator and **constructor**

Heap in java

- Return the reference
- Garbage collector
  - Happen automatically during the life time of java program
  - De-allocate

OutOfMemoryError …

Using super

- Overrides one of its superclass's methods
- Pass default constructor values if needed

```
public class ClassA {

protected int getValue()
{
        return 10;
}

}
```

```
public class ClassB extends ClassA {
private void printValue()
{
        System.out.println("Value 1 -
        [" + super.getValue() +"]");

        System.out.println("Value 2 -
        [" + getValue() + "]");
}

public int getValue()
{
        return 20;
}
}
```

```
public class ClassB extends ClassA {
private void printValue()
{
        System.out.println("Value 1 -       [" +
super.getValue() +"]");

        System.out.println("Value 2 -       [" +
getValue() + "]");
}

private int getValue()
{
        return 20;
}
}
```

Must be **protected or public**

```
public class ClassC extends
ClassB {

private void printValue()
{
System.out.println("Value 1 - ["
+ super.getValue() +"]");
System.out.println("Value 2 - ["
+ getValue() + "]");
}

public int getValue()
{
return 30;
}
```

Writing Final Classes and Methods

- Final class
  - Can not be sub classed
- Final method
  - Can not override

```
public class ClassA {

protected final int getValue()
{
        return 10;
}

}
```

```
public class ClassB extends ClassA {
private void printValue()
{
        System.out.println("Value 1 -
        [" + super.getValue() +"]");

        System.out.println("Value 2 -
        [" + getValue() + "]");
}

public  etVa  ()
{
      turn
}
}
```

Writing Abstract Classes and Methods

- Should have at least one abstract method
- Can not initiate object
- Must be extends from other class
- Every abstract method must be implemented
- Can contain non-abstract method

Writing Abstract Classes and Methods

```
public abstract class AbstractA {

abstract public int getAValue();

public void printValue()
{
System.out.println("This is the non abstract method");
}
}
```

```
public class NormalClassA extends AbstractA {

}
```

```
public class NormalClassA extends AbstractA {

private void displayValue()
{
            System.out.println(getAValue());
            printValue();
}

@Override
public int getAValue() {
            return 20;
}


public static void main(String[] args)
{
            NormalClassA n = new NormalClassA();
            n.displayValue();
}
}
```

OR ANOTHER WAY TO WRITE

```
AbstractA a = new AbstractA() {
    @Override
    public int getAValue() {
        return 25;
        }
    };

System.out.println(a.getAValue());
```

Interface

- Use to declare the abstract method & constant
- All methods in Interface are abstract
- By default methods are abstract & public
  - No need to write public abstract
- No body
- Can implement multiple interface

Interface

```
public interface Working
{
  public void work();
}

public class WorkingMan extends Man implements
Working
{
  public void work()
  {
             //Your implement work here
       }
}
```

```
public class WorkingMan extends
Man implements Working, Behavior
```

Interfaces vs Abstract Classes

feature	interface	abstract class
multiple inheritance	A class may implement several interfaces.	A class may extend only one abstract class.
default implementation	An interface cannot provide any code at all, much less default code.	An abstract class can provide complete code, default code, and/or just stubs that have to be overridden.
constants	Static final constants only, can use them without qualification in classes that implement the interface	Both instance and static constants are possible. Both static and instance initialize code are also possible to compute the constants.

```
public interface InterfaceA {

        void getValue();
        int JAVA_LESSION = 1;
}
```

SECTION 2

AI-Assisted Coding Tools

- GitHub Copilot (based on OpenAI Codex)
- ChatGPT for debugging & explaining code
- Tools assist, but do not replace logic
- **Real-world usage:** Many developers use Copilot to autocomplete functions, but they must still test and validate results

AI-Assisted Coding Tools

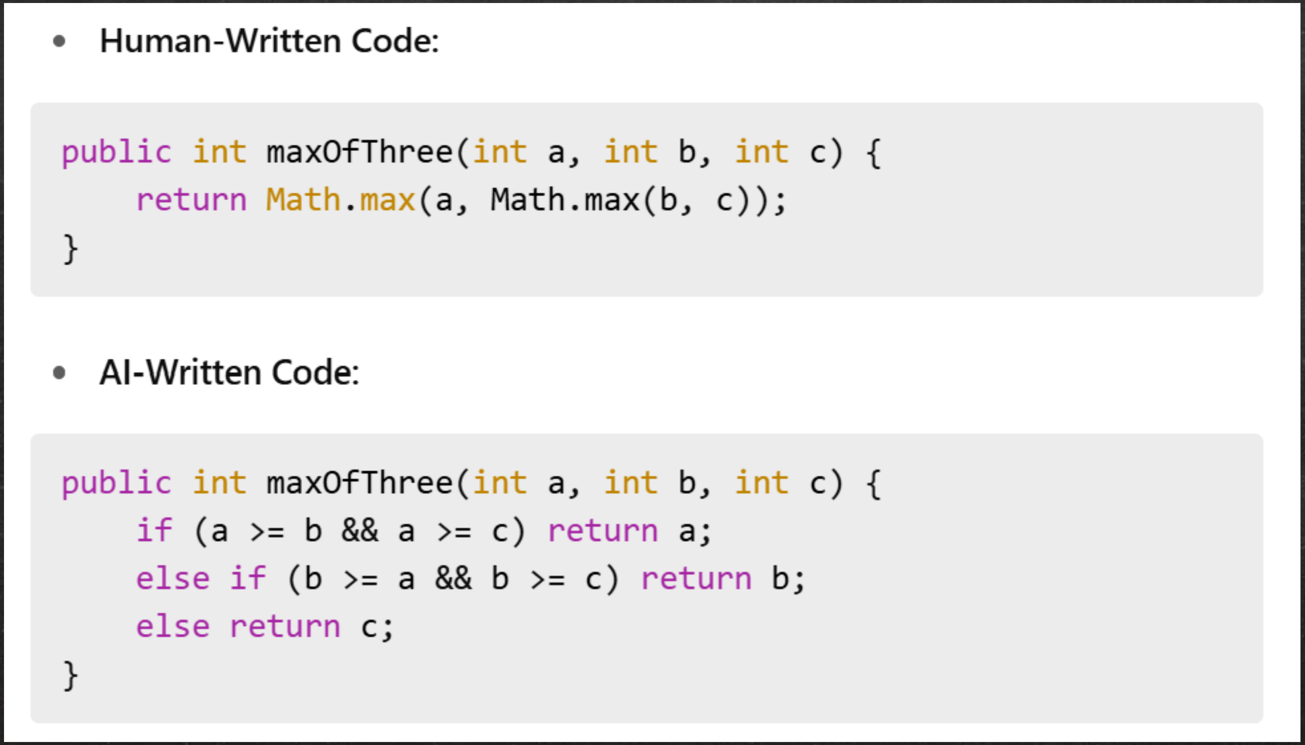
- CodeWhisperer (by AWS) – intelligent code generation in AWS environments
- Tabnine – AI-powered autocompletion for multiple IDEs and languages
- CodiumAI – generates meaningful unit tests and suggestions
- Mutable.ai – fast code and documentation generation for teams

AI-Assisted Coding Tools

- Replit Ghostwriter – real-time AI coding assistant inside Replit IDE
- AskCodi – AI for writing functions, documentation, SQL queries, and more
- Kite (discontinued but worth knowing) – early AI code completion tool
- Codeium – free alternative to Copilot, supports many languages and IDEs
- Claude (Anthropic)

Comparing Human vs AI-Written Code

- **Example Task:** Create a Java method to find the max of 3 numbers



- Human-Written Code:

```
public int maxOfThree(int a, int b, int c) {
    return Math.max(a, Math.max(b, c));
}
```

- AI-Written Code:

```
public int maxOfThree(int a, int b, int c) {
    if (a >= b && a >= c) return a;
    else if (b >= a && b >= c) return b;
    else return c;
}
```

Comparing Human vs AI-Written Code

- **Comparison Points:**
  - Clarity and readability
  - Use of built-in functions
  - Maintainability and potential bugs
- **Real-world impact:** Choosing between concise and verbose styles is a regular task in code reviews

Lab Activity

- Install Java JDK + IDE
- Create Hello World app
- Try GitHub Copilot / ChatGPT to generate basic Java code
- **Real-world context:** This setup mirrors how developers onboard to new projects or companies

Assignment

- **Task:** Reflect on your personal experience during the first lab
  - Describe what you learned while installing and writing your first Java program. What challenges did you face, and how did you overcome them? Did you use any AI tool, and how helpful was it?
  - How do you imagine applying this Java knowledge in your future job or career? What kind of project, product, or industry might it relate to?