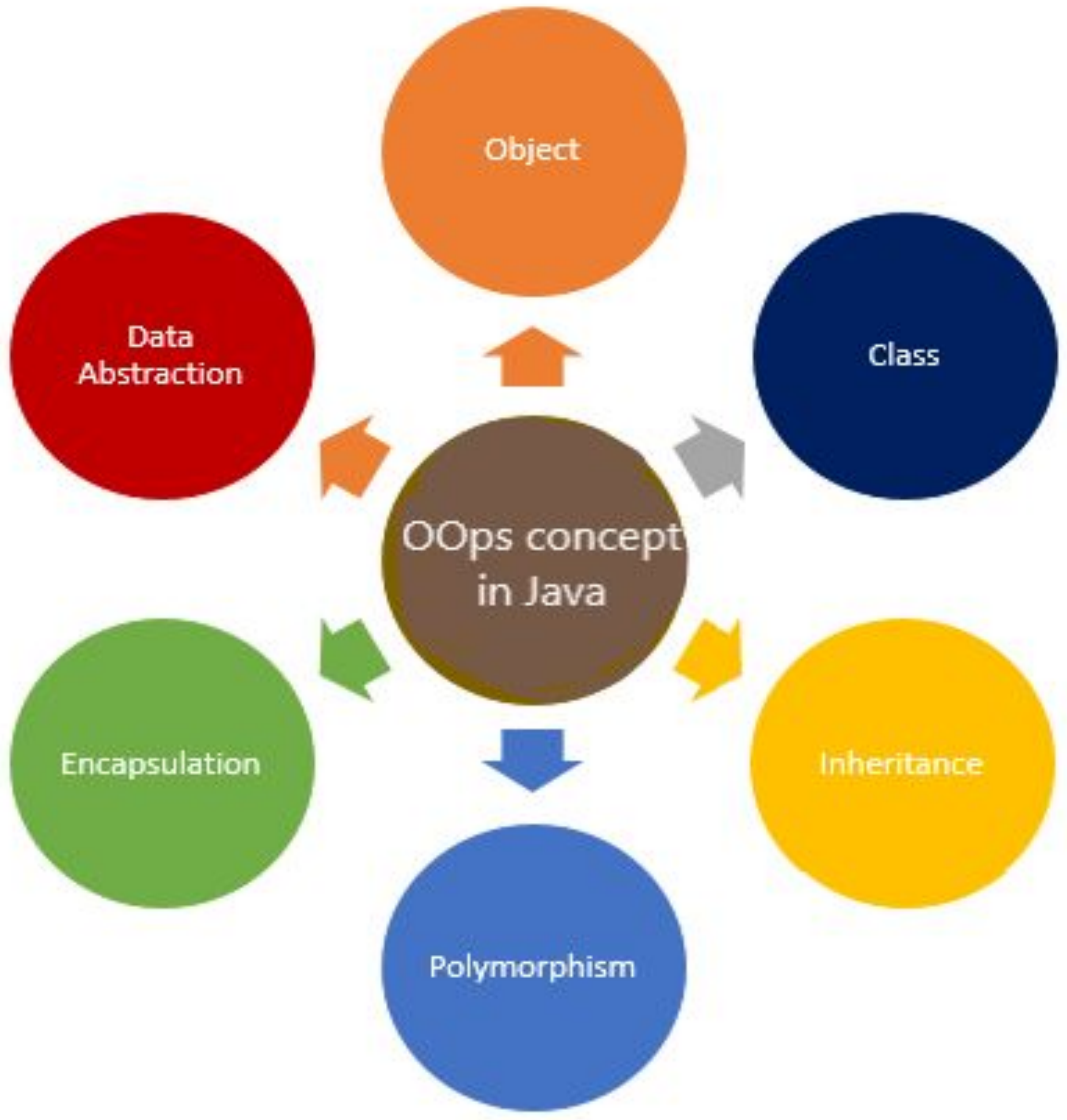


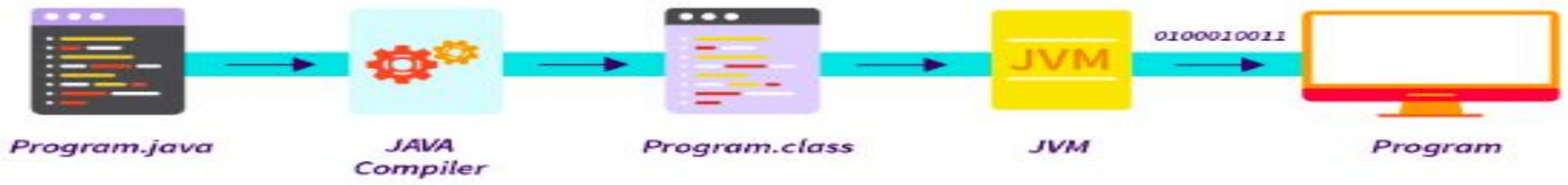
CS20004: Object Oriented Programming using Java

Lec-4

In this Discussion . . .

- Sample Java Program & Allied Explanation
- Java: Platform Independence, and Portability
- Java Basics
    - Data-types
        - Primitive
        - Non-primitive
    - Elements of Java Programs
    - Variables
        - Scope and lifetime of variables
- References

Working of the Java programming language



● We need to have a java source code otherwise we won't be able to run the program you need to save it with the **Program.java** extension.
● Secondly, we need to use a compiler so that it **compiles** the source code which in turn gives out the **java bytecode** and that needs to have a **Program.class** extension.   ● **The Java bytecode is a redesigned version of the java source codes, and this bytecode can be run anywhere irrespective of the machine on which it has been built.**
● Later on, we put the **java bytecode through the Java Virtual Machine which is an interpreter that reads all the statements thoroughly step by step from the java bytecode which will further convert it to the machine-level language so that the machine can execute the code.** We get the output only after the conversion is through.

Sample Java Program

Explanation

```
//Starting with a simple java program to
display a message

//Place to import packages
public class Program
{
     public static void main(String args[])
     {
          System.out.println("Welcome to
Java");
     }
}
```

// -> Single Line Comments in Java

public -> Access Specifier.
Suggests that members of the class are accessible outside the class, so that any other program can read them and use them.

class -> As java is object oriented, we need to write source code with at least one class. Here the class name is Program, & the first letter of a class should start with a capital letter (by convention)

main(String args[ ]) -> the arguments passed to main() method is to accept some value from outside. Here args[ ] is the array name which is of type String

Sample Java Program

Explanation

```
//Starting with a simple java program
to display a message

//Place to import packages
public class Program
{
    public static void main(String
args[])
    {
        System.out.println("Welcome to
Java");
    }
}
```

String args[ ] is required to be passed to main(), otherwise main() will not be recognized by JVM

static -> The static keyword is a non-access modifier used for methods and attributes.

Static methods/attributes can be accessed without creating an object of a class.
As main() method is called first before any creation of any object, we are using static to call it through class name.

public -> As JVM needs to access main() method from outside, the main() must be public

The main method must be present in every Java Application

Sample Java Program

Explanation

```
//Starting with a simple java program to
display a message

//Place to import packages
public class Program
{
    public static void main(String args[ ])
    {
        System.out.println("Welcome to
Java");
    }
}
```

System.out.println() ->

- Primarily, a method should be called using the format: object_name.methodName()
- We did not create an object of the class PrintStream (println() method present in this class).
- So, an alternate is to use: System.out
- System is a class present in java.lang package
- out is the static variable in System class
- When we call System.out, a PrintStream object will be created internally.

Sample Java Program

Explanation

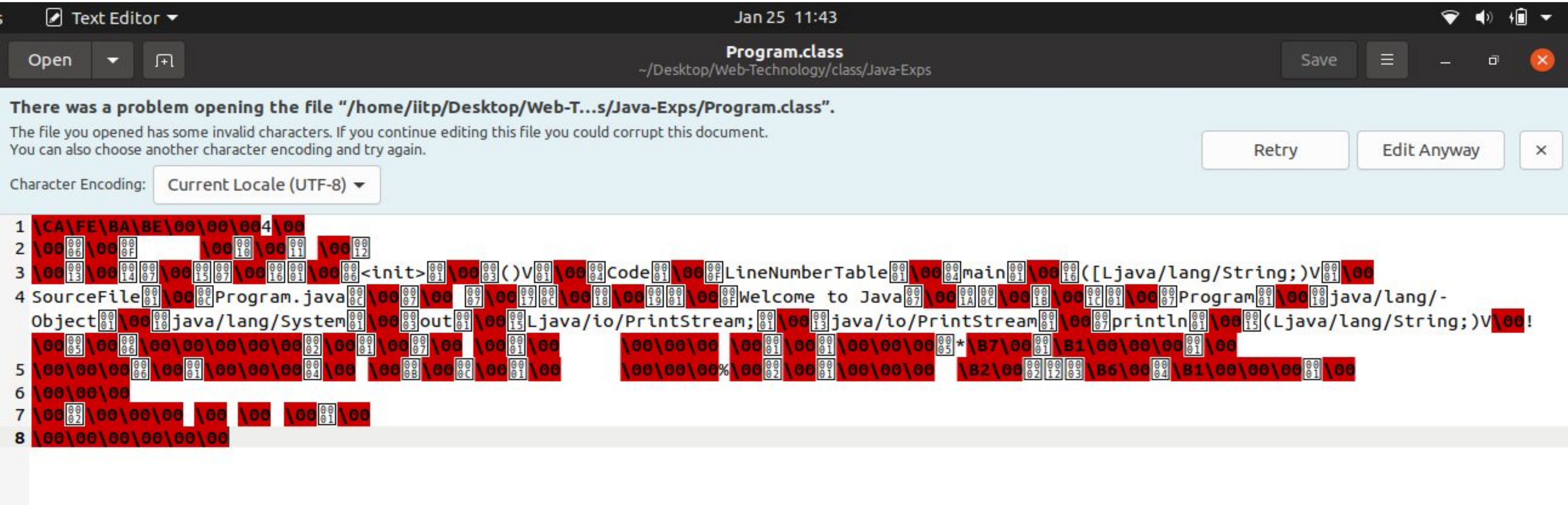
//Starting with a simple java program to display a message
 	
//Place to import packages
public class Program
{
    public static void main(String args[ ])
    {
        System.out.println("Welcome to Java");
    }
}
	● Methods to display on the monitor:
	
	1. println() : Displays the result on the monitor and places the cursor to a new line
	2. print() : Displays the result and retains the cursor in the same line

Java Program Execution

- Open the command prompt (Windows) or Terminal (Linux) and browse to the directory where the source code of the java program is present.

Compiling the Source code : **javac** Program.java

The above command will generate a .class file which contains byte code instructions. A sample screenshot of its contents, with respect to our program is:

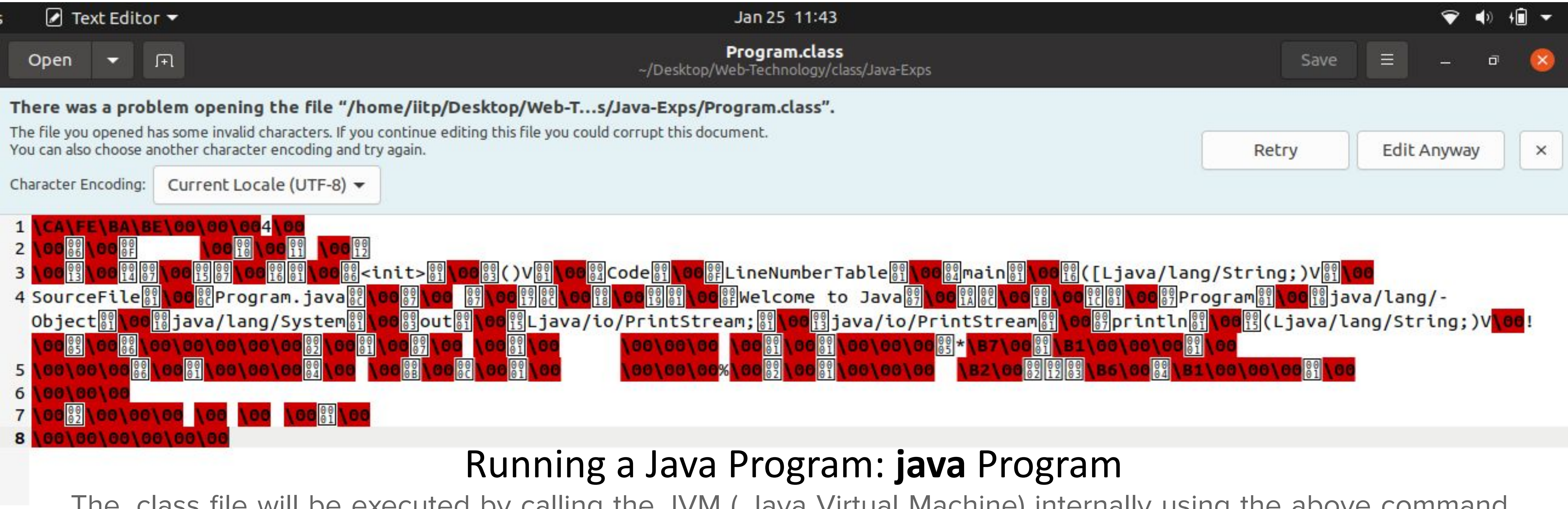


There was a problem opening the file "/home/iitp/Desktop/Web-T...s/Java-Exps/Program.class".
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

Character Encoding: Current Locale (UTF-8)

```
\CA\FE\BA\BE\00\00\004\00
\00\00        \00\00 \00
\00\00\00\00\00<init>\00()V\00Code\00LineNumberTable\00main\00([Ljava/lang/String;)V\00
SourceFile\00Program.java\00\00 \00\00\00\00Welcome to Java\00\00\00\00Program\00java/lang/-
Object\00java/lang/System\00out\00Ljava/io/PrintStream;\00java/io/PrintStream\00println\00(Ljava/lang/String;)V\00!
\00\00\00\00\00\00\00\00\00\00 \00\00     \00\00\00 \00\00\00\00\00*\B7\00\B1\00\00\00\00
\00\00\00\00\00\00\00\00 \00\00\00\00     \00\00\00%\00\00\00\00\00  \B2\00\B6\00\B1\00\00\00\00
\00\00\00
\00\00\00\00 \00 \00  \00\00
\00\00\00\00\00\00
```

Java Program Execution



There was a problem opening the file "/home/iitp/Desktop/Web-T...s/Java-Exps/Program.class".

The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

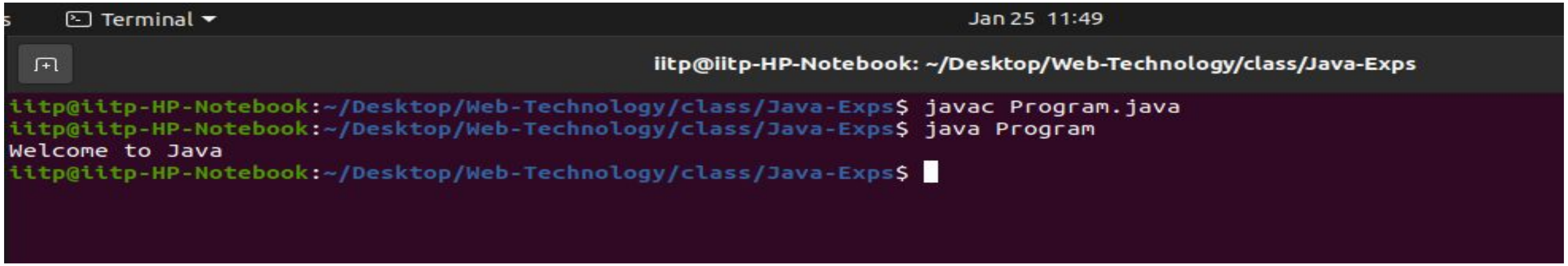
Character Encoding: Current Locale (UTF-8)

```
\CA\FE\BA\BE\00\00\004\00
\0006\000F      \0010\0011 \0012
\0013\001407\001507\001601\0006<init>01\0003()V01\0004Code01\000FLineNumberTable01\0004main01\0016([Ljava/lang/String;)V01\00
SourceFile01\000CProgram.java0C\0007\00 07\00170C\0018\001901\000FWelcome to Java07\001A0C\001B\001C01\0007Program01\0010java/lang/-
Object01\0010java/lang/System01\0003out01\0015Ljava/io/PrintStream;01\0013java/io/PrintStream01\0007println01\0015(Ljava/lang/String;)V\00!
\0005\0006\00\00\00\00\0002\0001\0007\00 \0001\00      \00\00\00 \0001\0001\00\00\0005*\B7\0001\B1\00\00\0001\00
\00\00\0006\0001\00\00\0004\00 \000B\000C\0001\00      \00\00\00%\0002\0001\00\00\00  \B2\00021203\B6\0004\B1\00\00\0001\00
\00\00\00
\0002\00\00\00 \00 \00 \0001\00
\00\00\00\00\00\00
```

Running a Java Program: **java** Program

The .class file will be executed by calling the JVM (Java Virtual Machine) internally using the above command (Here the keyword java is followed by the class file called Program without specifying its extension)

Output:


Terminal
Jan 25 11:49
iitp@iitp-HP-Notebook: ~/Desktop/Web-Technology/class/Java-Exps

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exps$ javac Program.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exps$ java Program
Welcome to Java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Exps$ 
```

Java : Platform Independence

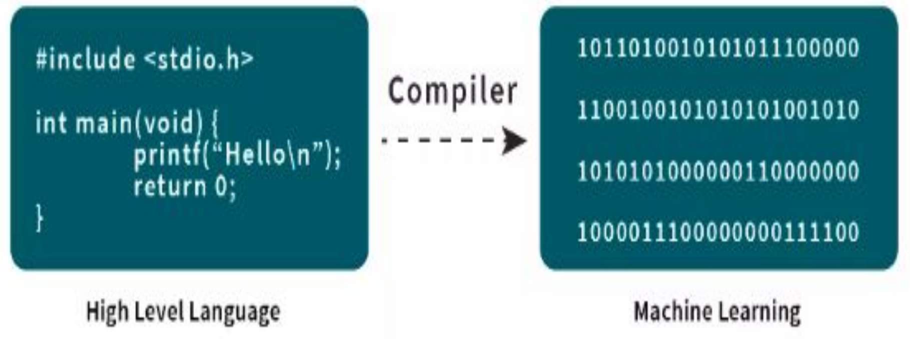
- The Java compiler doesn't generate "machine code" in the sense of native hardware instructions--rather, it generates **bytecodes:** a **high-level, machine-independent code for a hypothetical machine that is implemented by the Java interpreter and run-time system.**
- The solution that the Java system adopts to solve the binary-distribution problem is a "binary code format" that's **independent of hardware architectures, operating system interfaces, and window systems.**
- The format of this system-independent binary code is Platform Independent..
- *If the Java runtime platform is made available for a given hardware and software environment, an application written in Java can then execute in that environment without the need to perform any special porting work for that application.*

C/C++ Program Executions

- C/C++ is a high-level programming language that first needs to be compiled by their respective compiler which produces respective an executable file (.exe file), which can then be executed on the platform where these source codes are written.
  - Meaning if you write a program using C/C++ on windows and compiled it.
  - There itself it will create an .exe file (i.e., executable code). This .exe file can only be available and executed on that particular machine.

A compiler is a program that converts high-level language like C, C++, Java, etc., to machine code understandable by a computer. C is a compiled language which means code needs to be compiled first to run it.

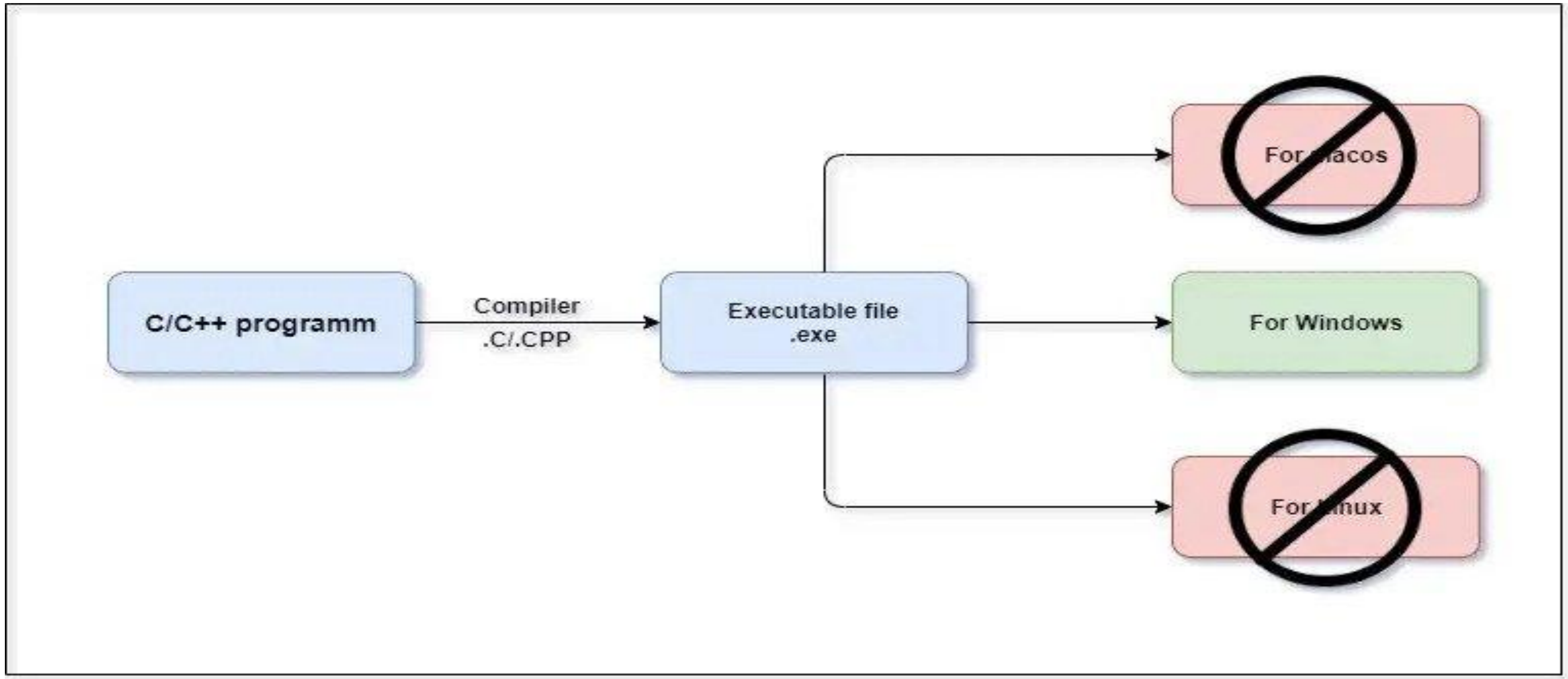
Compiler
01110
11001



```
#include <stdio.h>

int main(void) {
        printf("Hello\n");
        return 0;
}
```

```
1011010010101011100000
1100100101010101001010
1010101000000110000000
1000011100000000111100
```

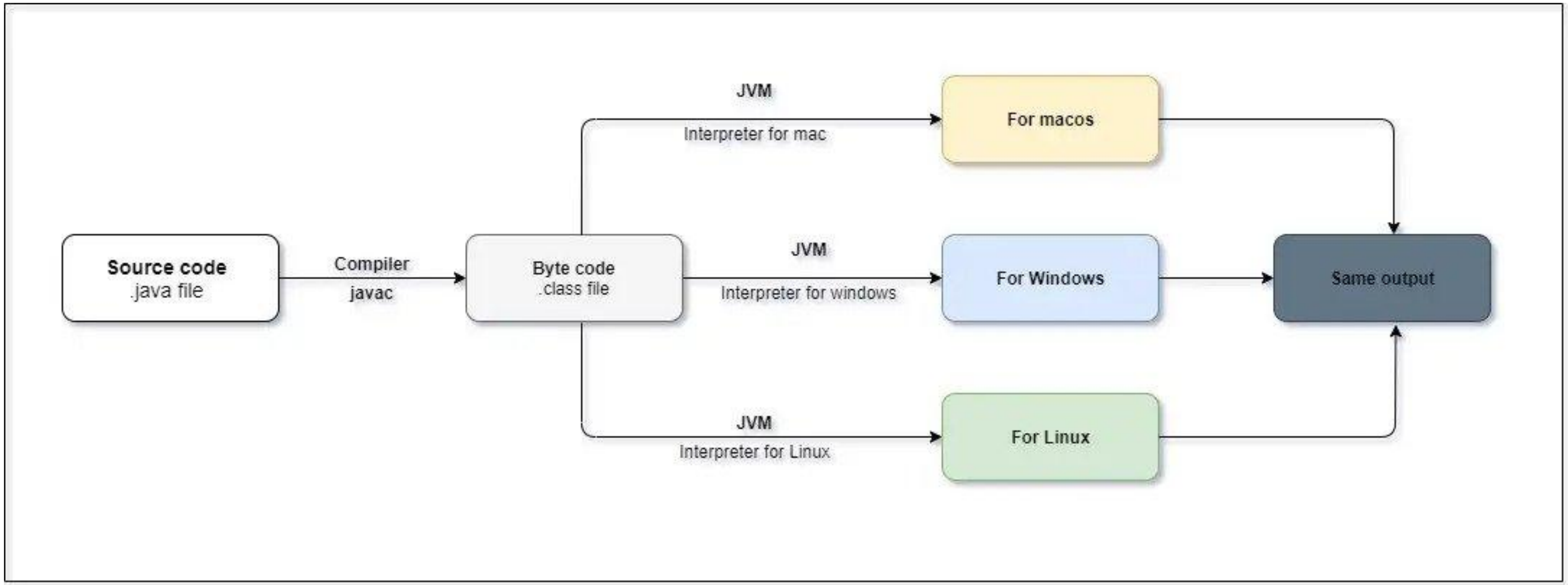
C/C++ Program Executions



- From the above image, it is clear that when a code is compiled on a Windows machine it creates windows executable file that is .exe.
- Likewise, if the code was compiled on Linux or MacOS it would have created respective executable for Linux and MacOS.
- So, We can see that each operating system has its own compiler, which makes C/C++ platform dependent.

Platform Independence: Java

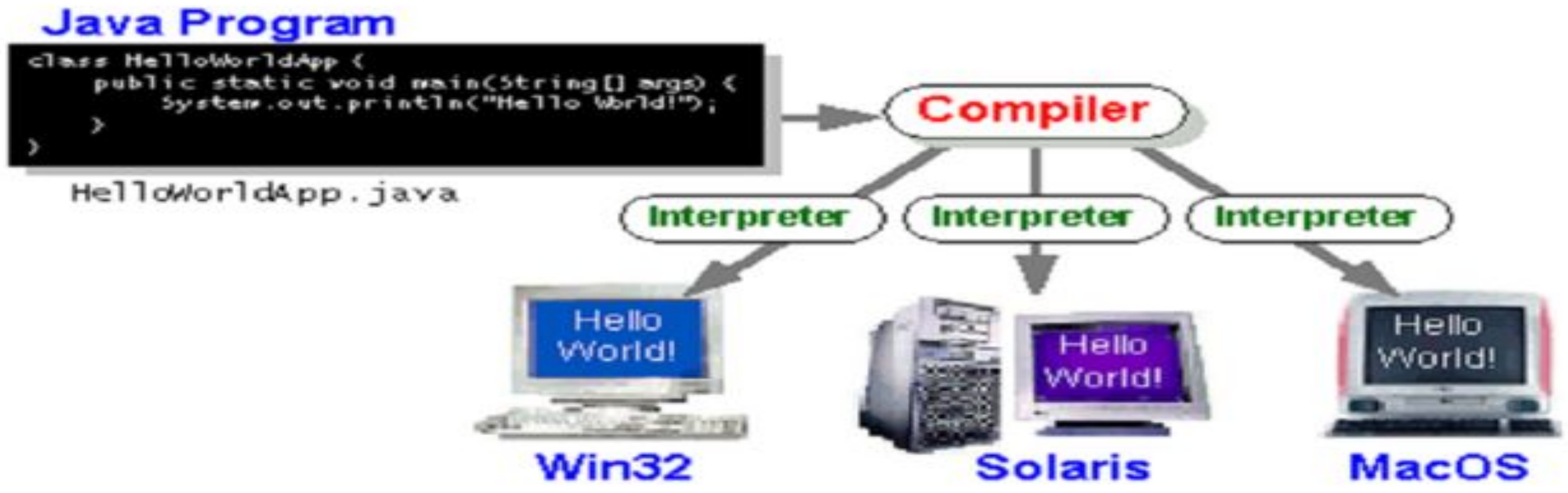
This feature makes java a platform-independent language or we can say java is a portable language.



The above image clearly tells us that how java file is executed.
We can also see that you can write code on various platform but the output will be the same.

- For Ex- if you write some sorting problem on windows then you don't need to do any changes in order to run it on macOS or Linux all you need to do is simply execute the code on the respective platform and there you go.

Platform Independence: Java



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

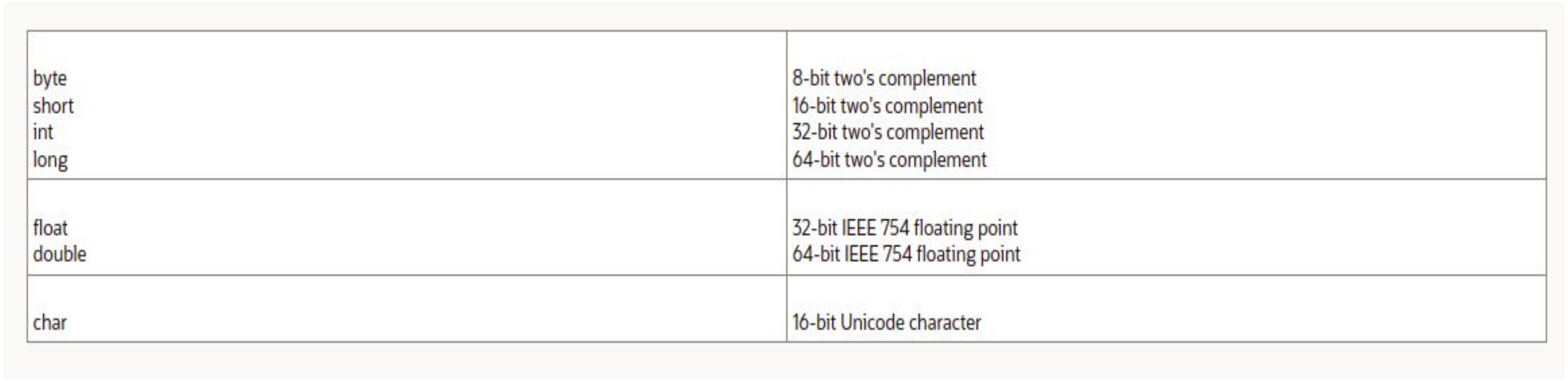
HelloWorldApp.java

Java : Portability

- The primary benefit of the interpreted bytecode approach is that compiled Java language programs are **portable** to any system on which the Java interpreter and run-time system have been implemented.
- The platform independence aspect discussed in previous slides is one major step towards being portable, but there's more to it than that.
- C and C++ both suffer from the defect of designating many fundamental data types as "implementation dependent". Programmers labor to ensure that programs are portable across architectures by programming to a lowest common denominator.
- Java eliminates this issue by defining standard behavior that will apply to the data types across all platforms. Java specifies the sizes of all its primitive data types and the behavior of arithmetic on them.

Java : Portability

- Here are the data types:



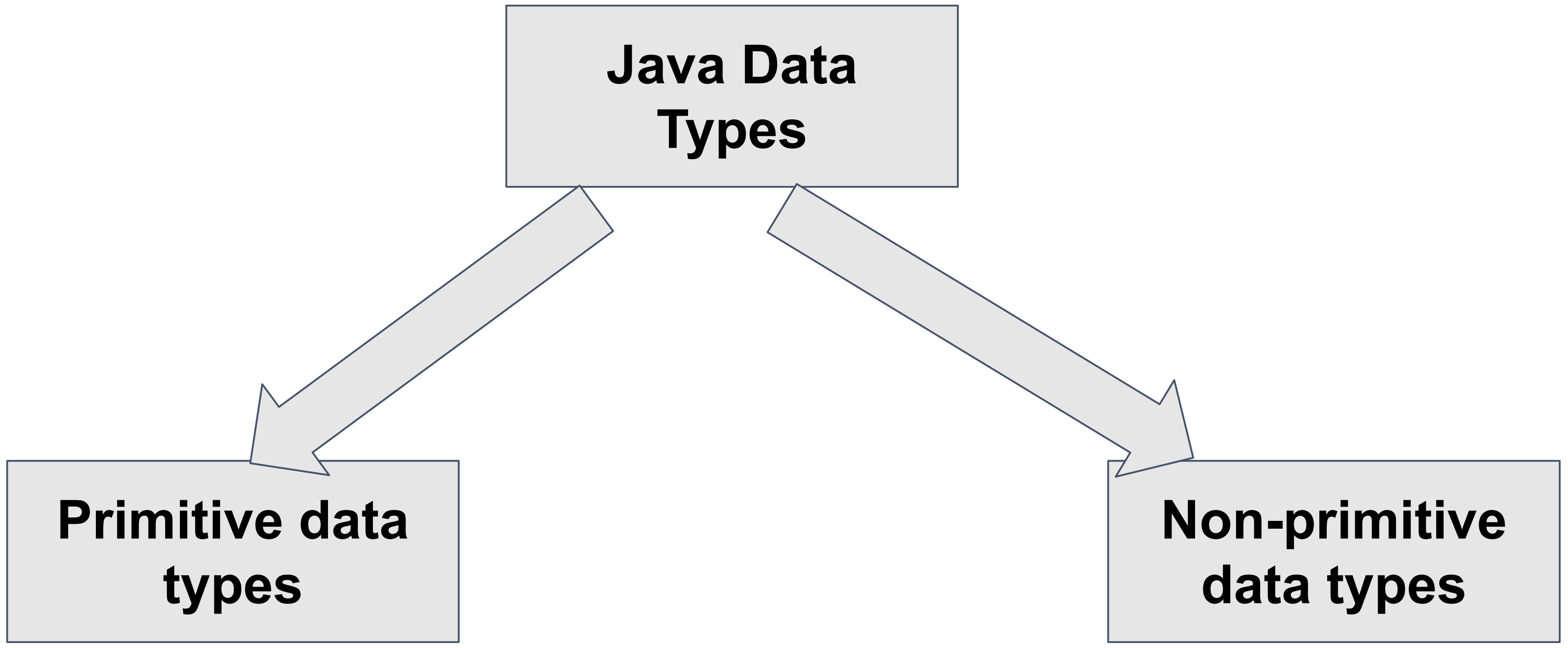
byte short int long	8-bit two's complement 16-bit two's complement 32-bit two's complement 64-bit two's complement
float double	32-bit IEEE 754 floating point 64-bit IEEE 754 floating point
char	16-bit Unicode character

The data types and sizes described above are standard across all implementations of Java. These choices are reasonable given current microprocessor architectures because essentially all central processor architectures in use today share these characteristics.

The Java environment itself is readily portable to new architectures and operating systems. The Java compiler is written in Java. The Java runtime system is written in ANSI C with a clean portability boundary which is essentially POSIX-compliant.

Java Data Types

- Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable’s type and name.
- Data types specify the different sizes and values that can be stored in the variable.



The primitive data types include boolean, **char, byte, short, int, long, float and double**.

The non-primitive data types include **Classes, Interfaces, and Arrays.**

Java Data types: Primitives

- Java is a strongly-typed language
- Every variable and expression has a type
- Every type is strictly defined
- All assignments are checked for type-compatibility
- No automatic conversion of non-compatible, conflicting types
- Java compiler type-checks all expressions and parameters
- Any typing errors must be corrected for compilation to succeed

Java Data types: Primitives

- There are 8 types of primitive data types:

Data Type	Description	Default value	Default size
boolean	Used to store only two possible values: true and false. Ex- boolean one = false	false	1 bit
byte	It is an 8-bit signed two's complement integer. Value-range: [-128 to 127] . It saves space because a byte is 4 times smaller than an integer. Ex- byte a = 10, byte b = -20	0	1 byte
short	The short data type is a 16-bit signed two's complement integer. Value-range : [-32,768 to 32,767]. It saves memory as it is 2 times smaller than an integer. Ex- short s = 10000, short r = -5000	0	2 bytes

Java Data types: Primitives (Contd.)

- There are 8 types of primitive data types:

Data Type	Description	Default value	Default size
int	It is a 32-bit signed two's complement integer. Value-range: [- 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1)]. Ex- int a = 100000, int b = -200000	0	4 byte
long	It is a 64-bit two's complement integer. Value-range: [-9,223,372,036,854,775,808 (-2^63) to 9,223,372,036,854,775,807(2^63 -1)]. Ex- long a = 100000L, long b = -200000L	0L	8 byte
float	It is a single-precision 32-bit IEEE 754 floating point.The float data type should never be used for precise values, such as currency. Its value is limited. Ex- float f1 = 234.5f	0.0f	4 bytes

Java Data types: Primitives (Contd.)

- There are 8 types of primitive data types:

Data Type	Description	Default value	Default size
double	It is is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float Ex-double d1 = 12.3	0.0d	8 byte
char	It is The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters. Ex- char letterA = 'A'	'\u0000'	2 byte

Elements of Java Program

Name	Description
Whitespaces	● space character, horizontal tab character, form feed character, and line terminator characters Ex- class HelloWorld
Identifiers	● Used to identify variables, methods and classes. ● They can be a sequence of alphabets, numbers, underscore character and a dollar-sign character. ● It can’t begin with a digit. ● Identifiers are case- sensitive ● Ex- int e, E;
Literals	● Used to specify constant values in a java program ● Ex- 100, 300.56,’M’, “India”

Elements of Java Program (Contd.)

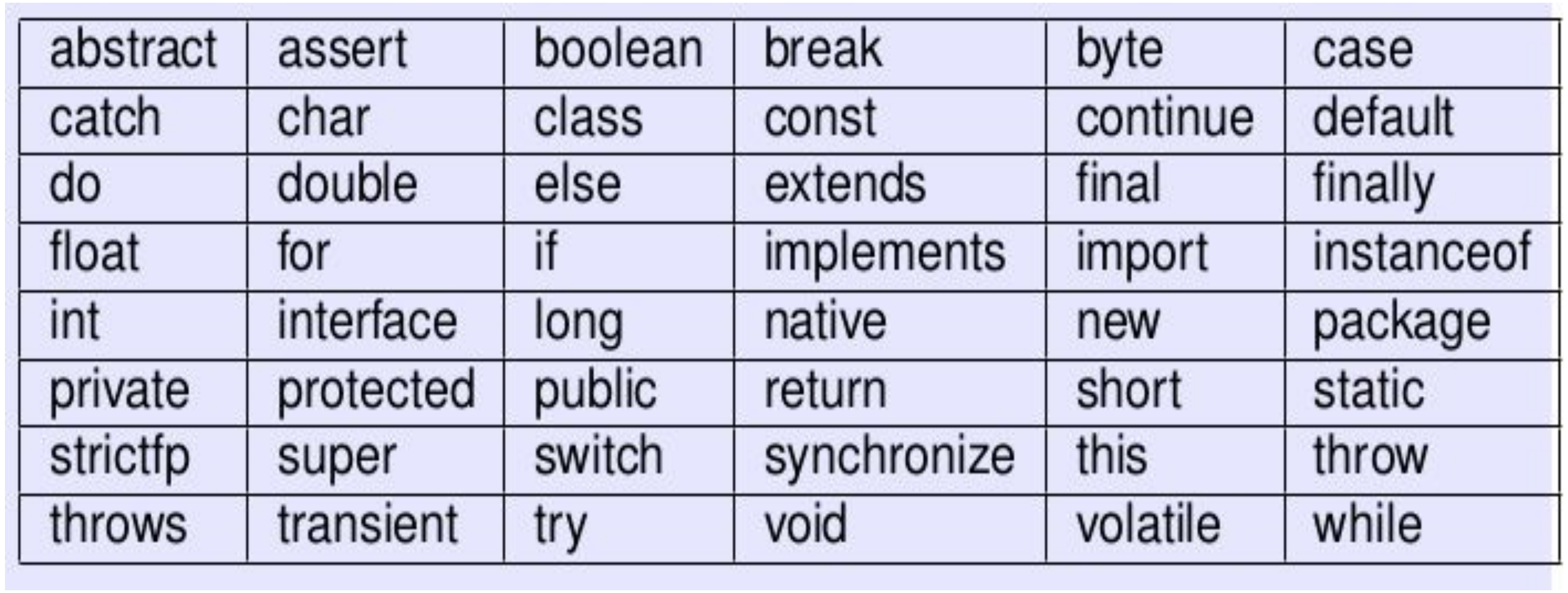
Name	**Description**
Comments	● Single-line Comments: //end of the loop /*end of the loop*/ ● Multiline Comments: /*this is a multiline comments of two lines*/ ● Documentation Comments: /** documentation */
Separators	● Semicolon (;) ● Period (.) ● Comma (,) ● Brackets ([ ]) ● Curly braces ({ }) ● Parentheses (( ))

Elements of Java Program (Contd.)

Name	**Description**
Escape Sequences	● \ddd: octal character ddd ● \uxxxx: hexadecimal Unicode character xxxx ● \’: single quote ● \": double quote ● \\: backslash ● \r: carriage return ● \n: new line ● \f: form feed ● \t: tab ● \b: backspace

Elements of Java Program (Contd.)

- Java keywords are also known as reserved words.
- Keywords are particular words that act as a key to a code.
- These are predefined words by Java so they cannot be used as a variable or object name or class name.



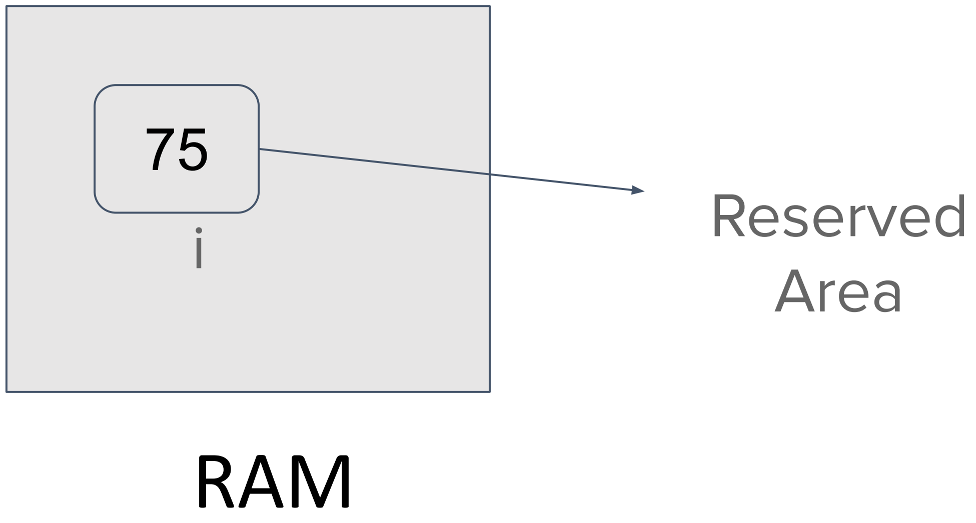
abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	extends	final	finally
float	for	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronize	this	throw
throws	transient	try	void	volatile	while

Java Variables

Syntax:- ***datatype identifier [=value];***

- A variable is a container which holds the value while the Java program is executed.
- A variable is assigned with a data type. All variables must be declared before they can be used.
- A variable is the name of a reserved area allocated in memory.
- In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.
- There are **three types** of **variables** in java:
  - **local**
  - **instance**
  - **class or static**.

Ex- int i = 75;



Java Variables: Local

- A variable declared inside the body of the method is called local variable.
- This variable can only be used within that method and the other methods in the class aren't even aware that the variable exists.

Note:- A local variable cannot be defined with "static" keyword.

```
public class Var1
{
   void method()
   {
      int n=90;//local variable - I
   }
   void resource()
   {
       int n = 81; //local variable - II
   }
   public static void main(String args[])
   {
   }
}//end of class
```

Java Variables: Instance

- A variable declared inside the class but outside the body of the method, is called an instance variable.
- It is called an instance variable because its value is instance-specific and is not shared among instances.

Note:- An Instance is not declared as static.

```
public class Var1
{
    int k = 9100; //instance variable
    void method()
    {
        int n=90;//local variable - I
    }
    void resource()
    {
         int n = 81; //local variable - II
    }
    public static void main(String args[])
    {
    }
}//end of class
```

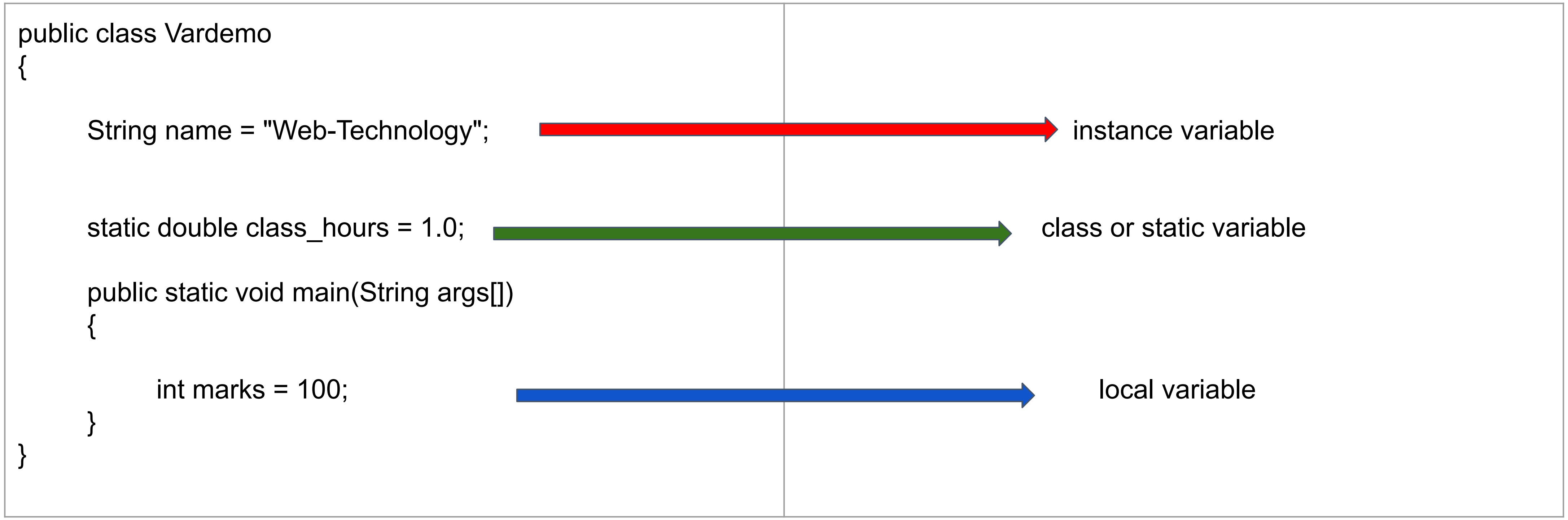
Java Variables: class or static

- Class variables are variables declared within the class outside any method. These variables contain the keyword **static**.
- A variable that is declared as static is called a static variable.
- You can create a single copy of the static variable and share it among all the instances of the class.
- Memory allocation for static variables happens only once when the class is loaded in the memory.

Note:- A static variable cannot be local.

```
public class Var1
{
   static int t = 1001; // static variable

    void method()
    {
        int n=90;//local variable - l
    }

    public static void main(String args[])
    {

    }
}//end of class
```

Demo Program Example: With respect to local, instance, and static variables



```
public class Vardemo
{

        String name = "Web-Technology";


        static double class_hours = 1.0;

        public static void main(String args[])
        {

                int marks = 100;
        }
}
```

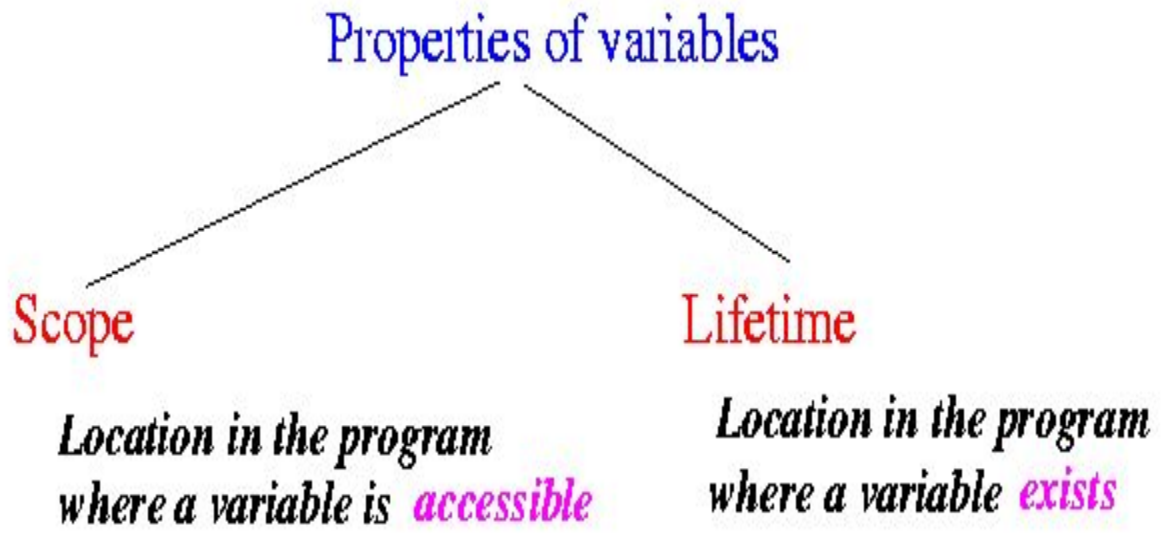
Scope and lifetime of variables

- Scope determines the visibility of program elements with respect to other program elements
- Block defines a scope. A block starts with a ’{’ and ends with a ’}’
- Each time a new block is started, a new scope begins
- The life of the variable is within its scope

- **Scope**: The scope of an element is the portion of the program where the element is visible.

  Lifetime: The lifetime of an element is the period of time during which it is alive

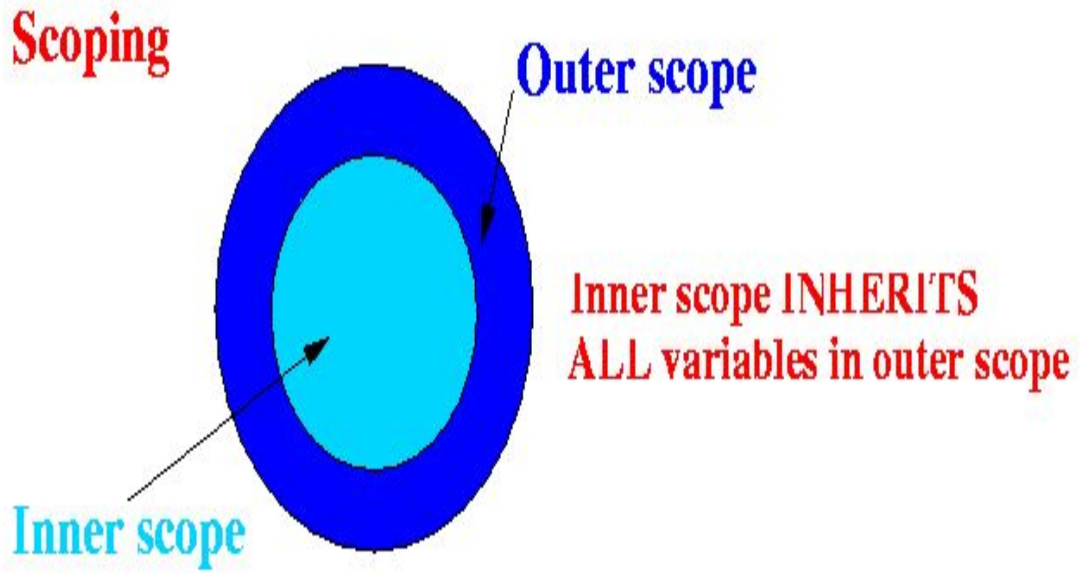
Scope and lifetime of variables



Location in the program where a variable is accessible

Location in the program where a variable exists

Note: the scope is a subrange of the lifetime
(if the variable does not exist, it cannot be accessed....)



Inner scope INHERITS ALL variables in outer scope

Scope of variables

- The scope of variable in Java can be classified into three main types:
  - Local Scope
  - Instance Scope
  - Class or static Scope

Scope of variables (Contd.): Local Scope

- Imagine you are organizing a small gathering at your home. You have invited a group of friends to enjoy a movie night. In order to create a cozy atmosphere, you decide to decorate the living room with fairy lights.
- In this scenario, you can relate the local scope to the concept of local variables. When you start decorating, you may need a temporary variable, let’s call it “**lightCount**,” to keep track of the number of fairy lights you have hung.

Scope of variables (Contd.): Local Scope

- Here’s how the local scope works in this example:
  - You declare the variable “lightCount” within the specific block or area where you are hanging the fairy lights.
  - This variable is only accessible within that specific block or area.
  - You can use it to increment the count as you hang each light.
  - Once you finish hanging the lights and exit that block or area, the variable “lightCount” is no longer accessible or relevant.

Scope of variables (Contd.): Local Scope

```
public class LightDecoration {
    public void hangLights() {
                // Local variable declared within the method
int lightCount = 0;

        for (int i = 0; i < 10; i++) {
 // Increment the light count as each light is hung
            lightCount++;
   System.out.println("Hanging light number " +
lightCount);
        }

 // The lightCount variable is only accessible //within the
hangLights() method
        System.out.println("Total lights hung: " + lightCount);
    }
    public static void main(String[] args) {
  LightDecoration decoration = new LightDecoration();
        decoration.hangLights();
    }
}
```

- In this example, the lightCount variable is declared within the hangLights() method.
- It is used to keep track of the number of lights being hung.
- As each light is hung, the lightCount variable is incremented.
- However, once the method execution is complete, the lightCount variable is no longer accessible.

Scope of variables (Contd.): Local Scope

- The local scope of the lightCount variable ensures that it is only relevant within the hangLights() method and does not interfere with other parts of the program.
- It allows us to perform specific calculations or tasks within a confined context without affecting the rest of the program's execution.

Scope of variables: Instance Scope

- Think of a real-life scenario where you have a class named Person.
- Each person has their own unique name and age.
- The name and age of each person represent the instance variables.
- Just like in Java, each instance of the Person class will have its own name and age values that are separate from other instances.

Scope of variables (Contd.): Instance Scope

```
public class Person {
    private String name; // Instance variable
    private int age; // Instance variable

    public void setName(String newName) {
        name = newName;
    }

    public void setAge(int newAge) {
        age = newAge;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
public static void main(String[] args) {
        Person person1 = new Person();
        person1.setName("John");
        person1.setAge(25);
        person1.displayInfo();

        Person person2 = new Person();
        person2.setName("Sarah");
        person2.setAge(30);
        person2.displayInfo();
    }
}
```

- Two instances of the Person class, person1, and person2, are created in the main() method.
- To change the names of person1 and person2 to “John” and “Sarah,” respectively, we call the setName() function on each of them.
- We call the setAge() method on person1 and person2 to set their ages to 25 and 30 respectively.

Scope of variables (Contd.): Instance Scope

```
public class Person {
    private String name; // Instance variable
    private int age; // Instance variable

    public void setName(String newName) {
        name = newName;
    }

    public void setAge(int newAge) {
        age = newAge;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
public static void main(String[] args) {
        Person person1 = new Person();
        person1.setName("John");
        person1.setAge(25);
        person1.displayInfo();

        Person person2 = new Person();
        person2.setName("Sarah");
        person2.setAge(30);
        person2.displayInfo();
    }
}
```

- In order to output the names and ages of person1 and person2 to the terminal, we finally call the displayInfo() function on each of them.
- The information for both people is displayed in the output.
- **Output:**

```
Name: John
Age: 25
Name: Sarah
Age: 30
```

Scope of variables: class or static Scope

- Imagine you are working in a company where you need to keep track of the total number of employees hired.
- Each time a new employee is hired, you want to increment this count.
- However, you want to ensure that the count is shared across all employees and not specific to individual employees.

Scope of variables (Contd.): class or static Scope Example

```
public class Employee {
    private static int employeeCount = 0; // Static variable

    public Employee() {
        employeeCount++;
    }

    public static int getEmployeeCount() {
        return employeeCount;
    }

    public static void main(String[] args) {
        Employee emp1 = new Employee();
        Employee emp2 = new Employee();
        Employee emp3 = new Employee();

        System.out.println("Total employees: " +
Employee.getEmployeeCount());
    }
}
```

- In the code, we have a class called Employee that has a static variable employeeCount to keep track of the total number of employees.
- Each time an Employee object is created (in the constructor), the employeeCount is incremented by 1.
- **Output:**

```
Total employees: 3
```

Scope of variables (Contd.): class or static Scope Example

```
public class Employee {
    private static int employeeCount = 0; // Static variable

    public Employee() {
        employeeCount++;
    }

    public static int getEmployeeCount() {
        return employeeCount;
    }

    public static void main(String[] args) {
        Employee emp1 = new Employee();
        Employee emp2 = new Employee();
        Employee emp3 = new Employee();

        System.out.println("Total employees: " +
Employee.getEmployeeCount());
    }
}
```

- The getEmployeeCount() method is declared as static, allowing us to access the total employee count without creating an instance of the Employee class.
- In the main() method, we create three Employee objects: emp1, emp2, and emp3.

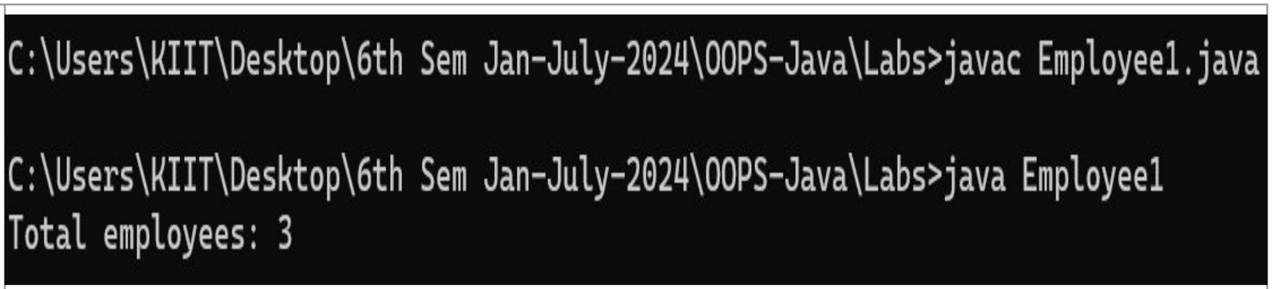
Scope of variables (Contd.): class or static Scope Example

```
public class Employee {
   private static int employeeCount = 0; // Static variable

   public Employee() {
      employeeCount++;
   }

   public static int getEmployeeCount() {
      return employeeCount;
   }

   public static void main(String[] args) {
      Employee emp1 = new Employee();
      Employee emp2 = new Employee();
      Employee emp3 = new Employee();

      System.out.println("Total employees: " +
Employee.getEmployeeCount());
   }
}
```

- Finally, we call the getEmployeeCount() method using the class name (Employee) to retrieve the total number of employees and display it as output.
- In this example, the static scope allows the employeeCount variable to be shared across all instances of the Employee class.
- It ensures that the count remains consistent and accessible to all employees.

Scope of variables (Contd.): class or static Scope Example (Variant)

```
public class Employee1
{
        private static int employeeCount = 0; // Static variable
        public Employee1()
        {
                employeeCount++;
        }
        public static int getEmployeeCount()
        {
                return employeeCount;
        }
        public static void main(String[] args)
        {
                Employee1 emp1 = new Employee1();
                Employee1 emp2 = new Employee1();
                Employee1 emp3 = new Employee1();
                System.out.println("Total employees: "
+getEmployeeCount());
        }
}
```

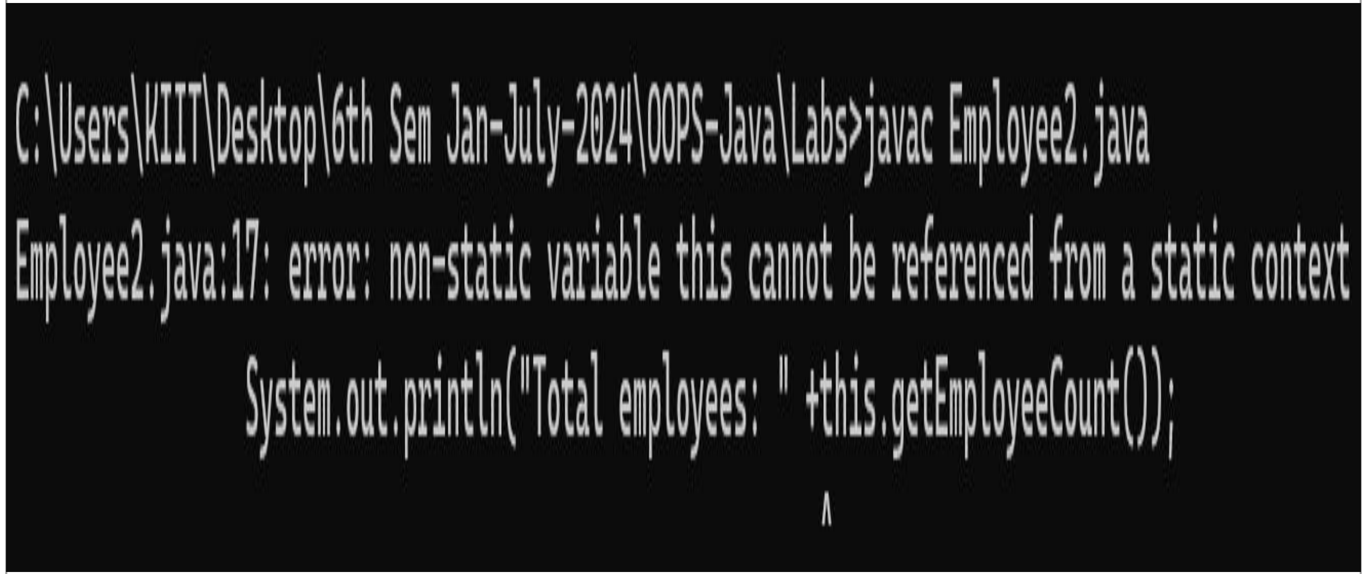


```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Employee1.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java Employee1
Total employees: 3
```

- Due to the static nature of the function getEmployeeCount(), we can directly call in the static main method without using the class name (Employee) / creating an exclusive object to retrieve the total number of employees and display it as output.
- In this example, the static scope allows the employeeCount variable to be shared across all instances of the Employee class.

Scope of variables (Contd.): class or static Scope Example (Variant using **this**)

```
public class Employee2
{
        private static int employeeCount = 0; // Static variable
        public Employee2()
        {
                employeeCount++;
        }
        public static int getEmployeeCount()
        {
                return employeeCount;
        }
        public static void main(String[] args)
        {
                Employee2 emp1 = new Employee2();
                Employee2 emp2 = new Employee2();
                Employee2 emp3 = new Employee2();
                System.out.println("Total employees: "
+this.getEmployeeCount());
        }
}
```



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Employee2.java
Employee2.java:17: error: non-static variable this cannot be referenced from a static context
                System.out.println("Total employees: " +this.getEmployeeCount());
                                                        ^
```

Scope: Conclusion

- **Local scope** variables are those that are declared inside a method or a block and are only usable within that particular method or block.
- **Instance scope** variables are declared inside a class but outside of any methods, and all of that class's methods can access and use them.
- The static keyword is used to declare variables having a **static scope**, which is shared by all instances of a class. They are frequently used to maintain global data or constants and can be accessed by the class name.

Scope and lifetime of variables (Contd.)

Variable Type	**Scope**	**Lifetime**
Instance variable	Throughout the class except in static methods	Until the object is available in the memory
Class or static variable	Throughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Java variable Types: Based on Scope

- In Java, there exists different types of variables based on scope:
    - Member Variables (Class Level Scope)
    - Local Variables (Method Level Scope)

References

1. https://www.javatpoint.com/difference-between-jdk-jre-and-jvm
2. https://www.oracle.com/java/technologies/architecture-neutral-portable-robust.html#:~:text=The%20Java%20environment%20itself%20is,which%20is%20essentially%20POSIX%2Dcompliant.
3. https://www.oreilly.com/library/view/the-java-language/9780133260335/ch03lev1sec6.html#:~:text=White%20space%20is%20defined%20as,terminator%20characters%20(%C2%A73.4).
4. https://www.javatpoint.com/scope-of-variables-in-java
5. https://www.cs.emory.edu/~cheung/Courses/255/Syllabus/C/C.-ver1/C-Advanced/scoping.html
6. https://blog.geekster.in/scope-of-variable-in-java/
7.