

Object Oriented Programming with Java



Sunitha C S
STDC, Kochi





Introduction to Java

- Java is a **high-level, versatile**, and widely-used programming language that was first developed by **James Gosling and his team at Sun Microsystems** (now owned by **Oracle Corporation**) in the mid-1990s.
- It is known for its **platform independence**, which means that Java code can run on different operating systems without modification, thanks to its "Write Once, Run Anywhere" (**WORA**) philosophy.



Introduction to Java

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded and dynamic language."



Java is simple...



- Removal of complicated statements
- Removal of struct and union
- Removal of the direct use of pointers
- Dynamic object creation
- Automatic garbage collection
- Rich set of classes for I/O, Network, Graphics
- Selective inclusion of Java classes



Java Language

- **Object Oriented language**
 - All programs and data always exist in the context of objects
- **A Distributed language**
 - Access of local or remote files with equal ease. Designed for the distributed Environment of Internet
- **Compiled/Interpreted language**
 - Java compiles programs into an intermediate representation called the bytecode. This code can be executed on any machine that has JVM
- **Architecturally Neutral**
 - Independent of any processor type and machine architecture. Goal used is “write once, run anywhere, anytime forever”



Java Language

- **Robust**

- In a well-written Java program, all run-time errors can be handled by the system
 - Exception handling, strong type checking

- **Secure**

- Enables construction of virus-free systems.
- Java program always runs in Java Runtime Environment with almost null interaction with system OS, hence it is more secure.

- **Portable**

- Java bytecode can be carried to any platform

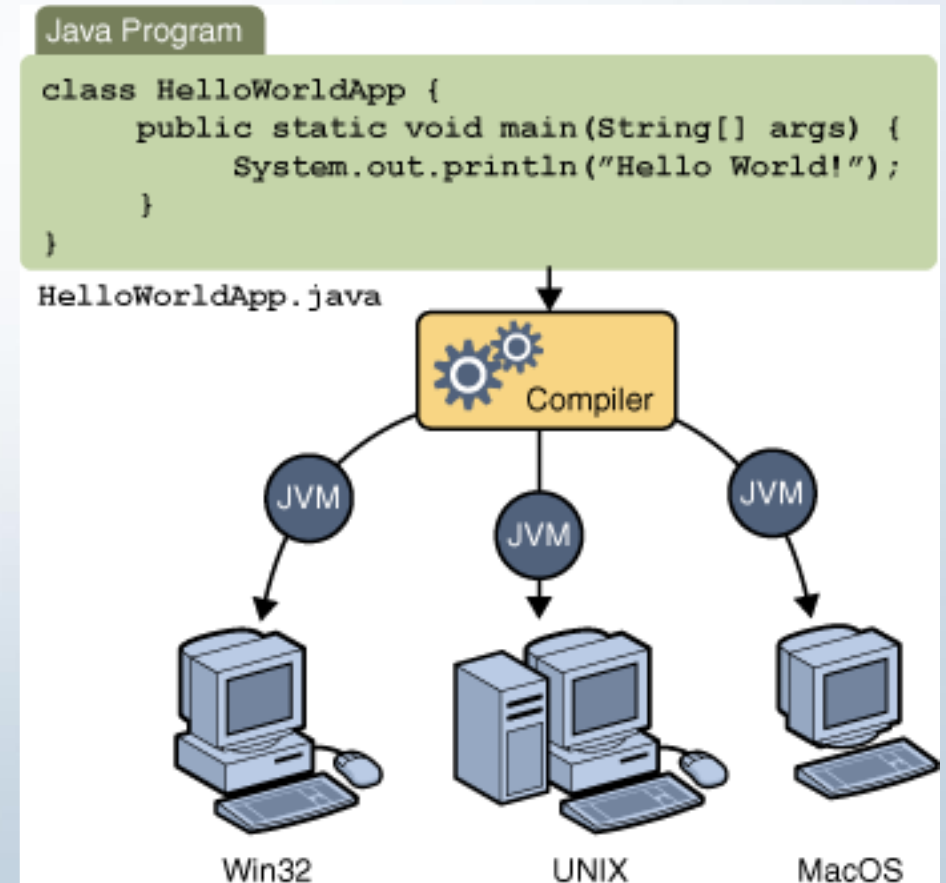
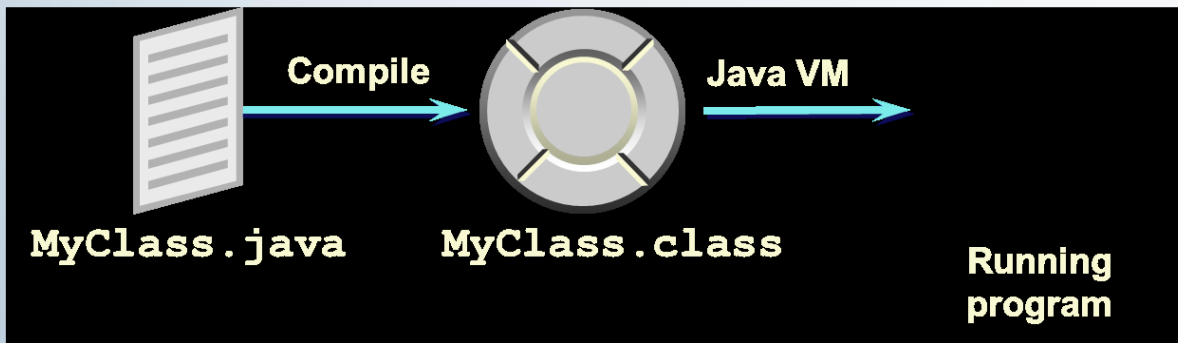
- **Multi-threaded**

- Java programs that deal with many tasks at once by defining multiple threads.



Platform Independence

- Java code is compiled into processor-independent bytecodes
- Bytecodes are interpreted by the Java Virtual Machine (VM) at run time





Java in a Big Picture

- Java Technology based software can run anywhere.
- Components don't care what kind of computer, phone, TV or Operating System they run on.
- Works on any kind of compatible device that supports the Java platform.



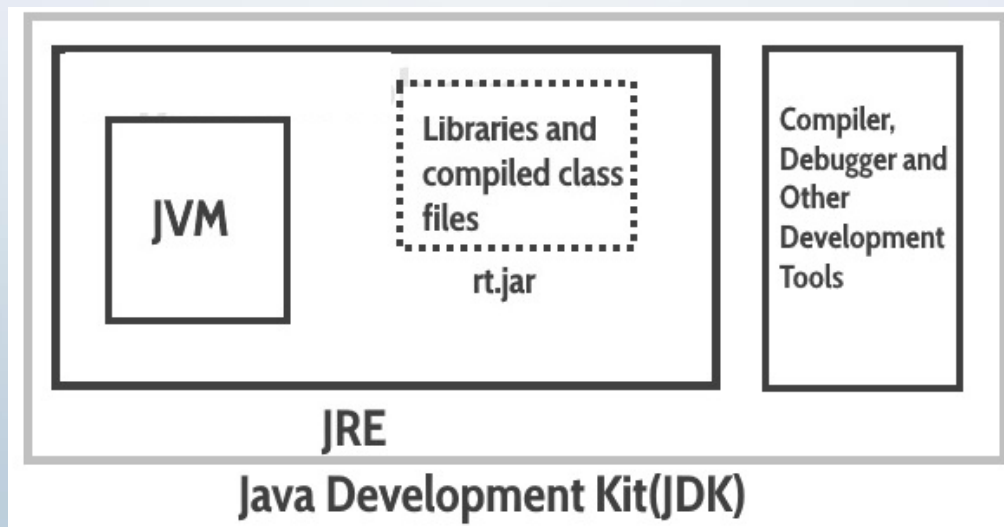
Java Compiler

- Java is a high level programming language. A program written in high level language cannot be run on any machine directly.
- First, it needs to be translated into that particular machine language. The **javac compiler** does this thing, it takes java program (.java file containing source code) and translates it into intermediate code (referred as **byte code** or **.class file**).



Difference JDK, JRE & JVM

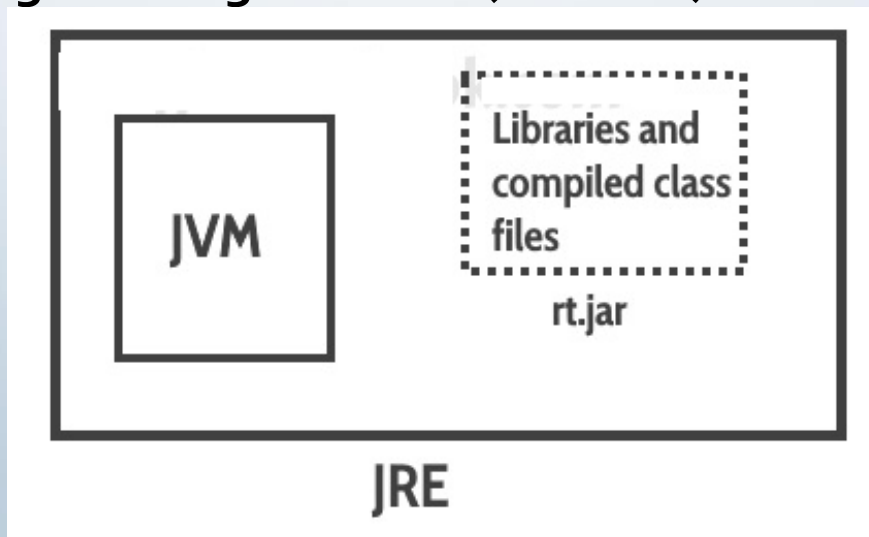
- **JDK– (Java Development Kit)** - it contains JRE, alongwith development tools include Compiler(javac), Java application launcher(java), Appletviewer, The Java Debugger(jdb), Java documentation generator (JavaDoc)etc...
- **JRE–Java Runtime Environment** is the environment with in which the java virtual machine runs. JRE contains Java virtual Machine(JVM), class libraries, and other files excluding development tools such as compiler and debugger.
- **JVM-Java Virtual Machine** runs the program by using class, libraries and files provided by JRE.





JRE

- **JRE has two components:**
 - **Java Virtual Machine (JVM)**
 - Virtual machine that executes byte code
 - Available on many types of h/w and s/w platforms which enables Java to function as a middleware and a platform on its own
 - JVM also performs garbage collection
 - **Java Application Programming Interface (Java API)**





JIT (Just-in-time compiler)

- JIT is a part of JVM that is used to speed up the execution time.
- JIT compiles the part of bytecode that have similar functionality at the same time and hence reduce the amount of time needed for compilation.

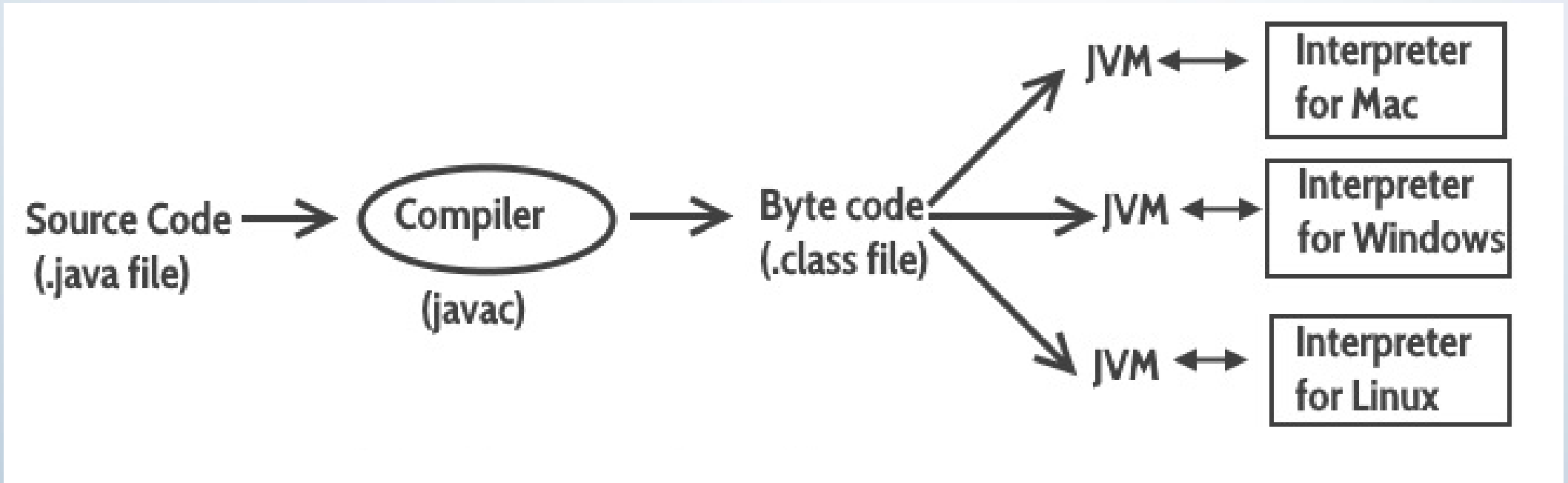


JVM – Java Virtual Machine

- Java Virtual Machine (JVM) is a virtual machine that resides in the real machine (your computer) and the **machine language for JVM is byte code**. This makes it easier for compiler as it has to generate byte code for JVM. JVM executes the byte code generated by compiler and produce output.
- **JVM is the one that makes java platform independent.**
- So, the primary function of JVM is to execute the byte code produced by compiler.
- **Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.** Which means that the byte code generated on Windows can be run on Mac OS and vice versa. That is why we call java as platform independent language.

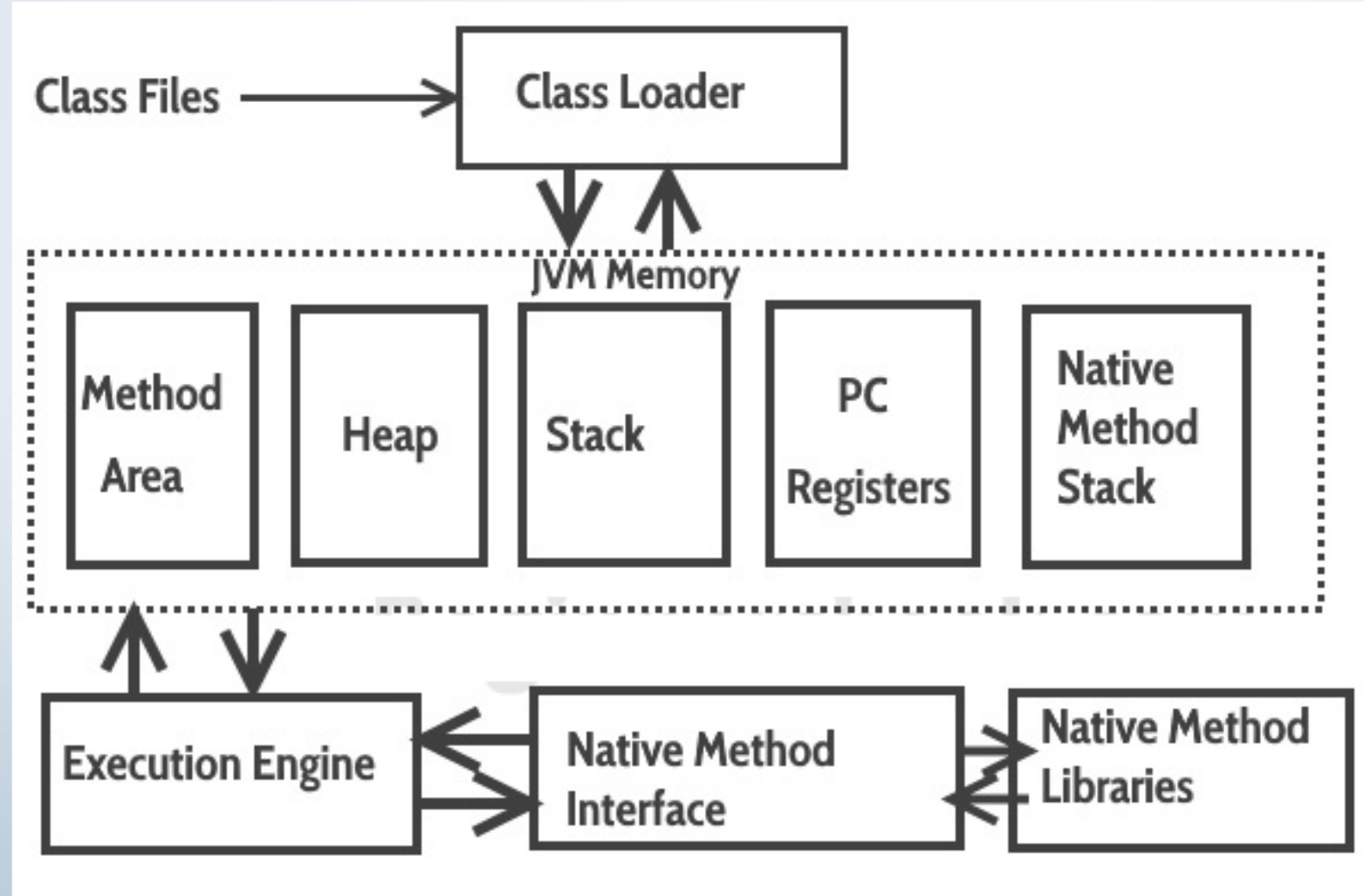


JVM – Java Virtual Machine





JVM Architecture



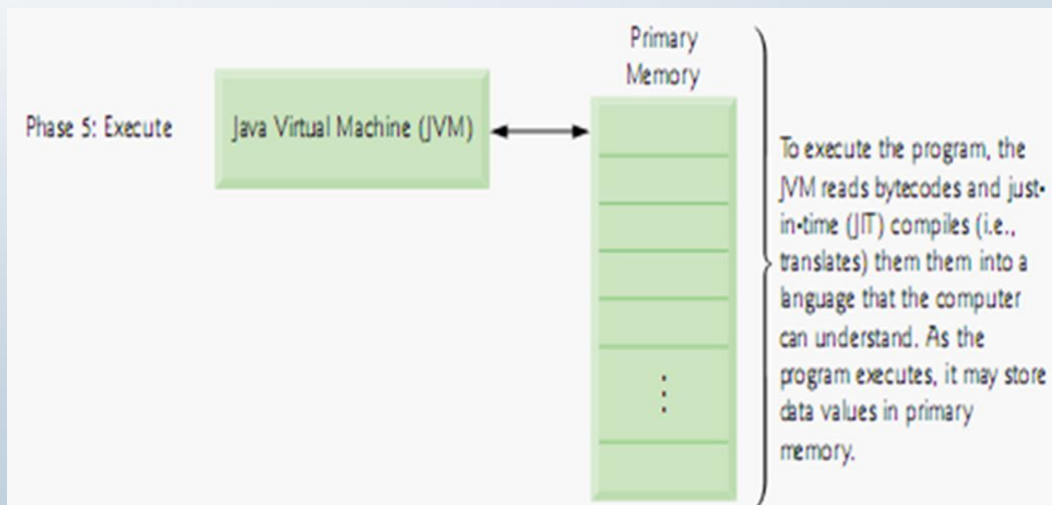
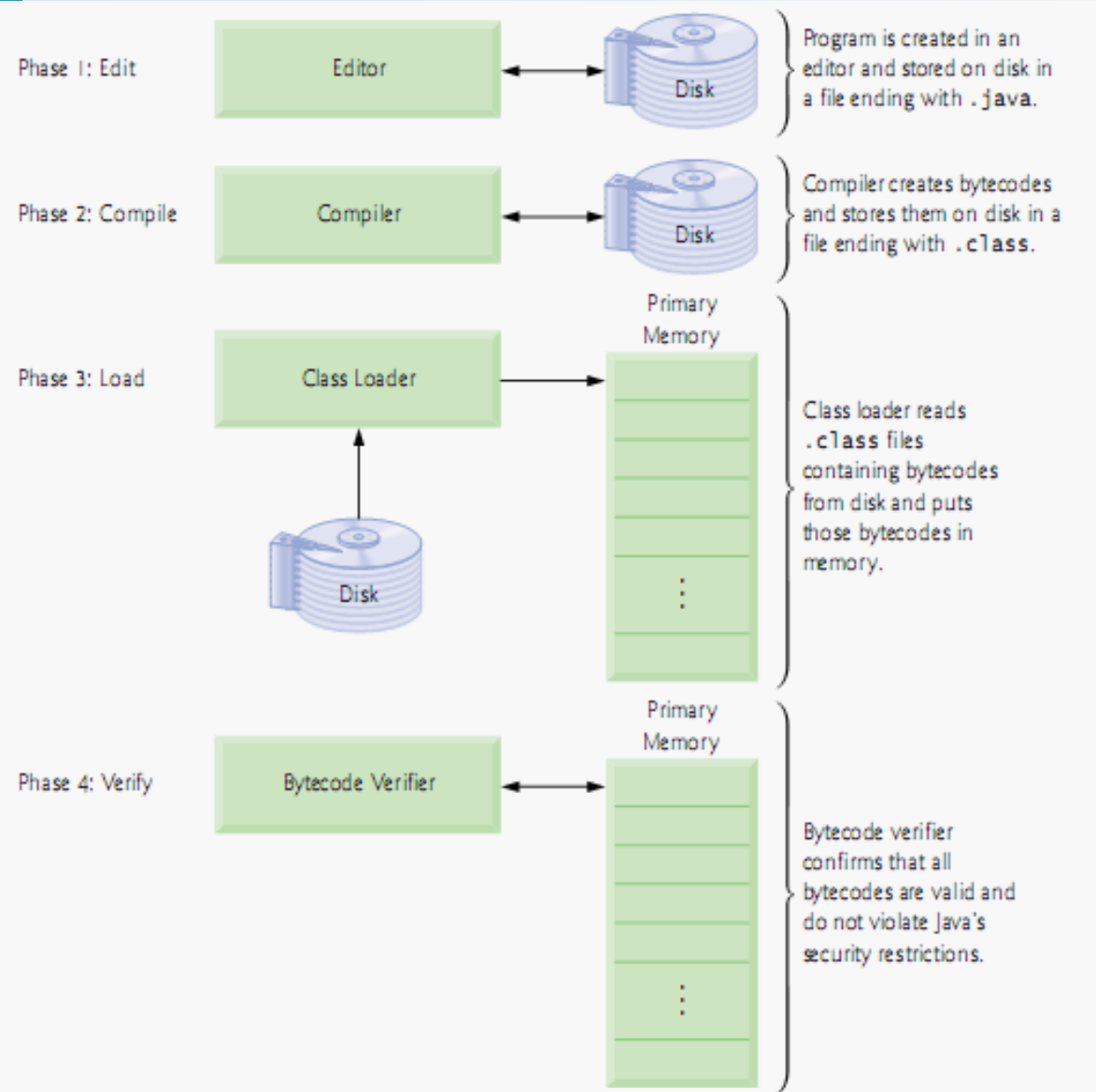


JVM Architecture

- **Class Loader:** The class loader reads the .class file and save the byte code in the **method area**.
- **Method Area:** There is only one method area in a JVM which is shared among all the classes. This holds the class level information of each .class file.
- **Heap:** Heap is a part of JVM memory where objects are allocated. JVM creates a Class object for each .class file.
- **Stack:** Stack is a also a part of JVM memory but unlike Heap, it is used for storing temporary variables.
- **PC Registers:** This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC register.
- **Native Method stack:** A native method can access the runtime data areas of the virtual machine.
- **Native Method interface:** It enables java code to call or be called by native applications. Native applications are programs that are specific to the hardware and OS of a system.
- **Garbage collection:** A class instance is explicitly created by the java code and after use it is automatically destroyed by garbage collection for memory management.



Typical Java Development Environment (JDE) in JDK





Simple Java Program

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



- Java is purely Object Oriented. Hence we **cannot write a java program without having class.**
- The “**main**” method is the method in which execution to any java program **begins.**
- The method is “**public**” because it be accessible to the JVM to begin execution of the program.
- It is “**static**” because it be available for execution without an object instance.
- It returns only a **void** because, once the main method execution is over, the program terminates. So there can be no data that can be returned by the main method .



- The last **parameter** is String args[]. This is used to signify that the user may opt to enter parameters to the java program at command line. We can use both String[] args or String args[]. The Java compiler would accept both forms.
- A class code starts with a {and ends with a}.
- In Java, **println()** method is used to display something on the monitor.
- A method should be called by using **objName.methodname()**. println() is a **static function** of the class **System.out**
- In the Sample program **System** and **String** are the classes present in **java.lang** package.



Simple Java Application

- A .java file can be compiled to a .class file using **javac** compiler as follows:

```
prompt> javac SayHello.java  
... compiler output ...
```

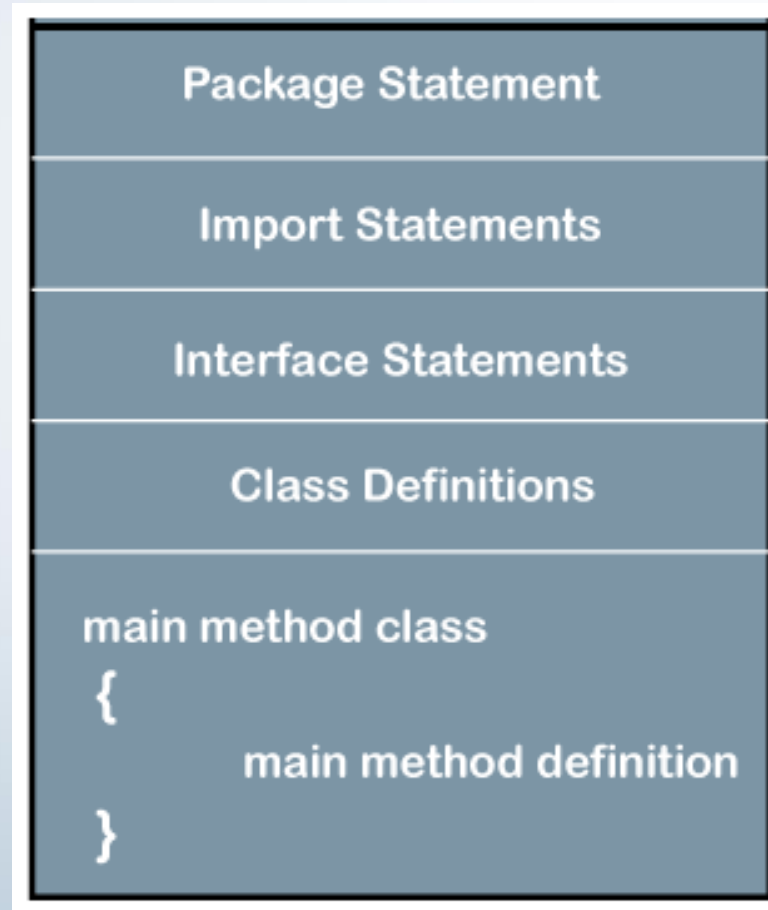
- An application .class file is executed using Java application launcher **java** as follows:

```
prompt> java SayHello  
Hello world  
prompt>
```

<i>Task</i>	<i>Tool to use</i>	<i>Output</i>
Write the program	Any text editor	File with .java extension
Compile the program	Java Compiler	File with .class extension (Java bytecodes)
Run the program	Java Interpreter	Program Output



Structure of a Java Program





Comments

- Comments explain the program to yourself and others
- Block comments
 - Can span several lines
 - Begin with `/*`
 - End with `*/`
 - Compiler ignores all text between `/*` and `*/`
- Line comments
 - Start with `//`
 - Compiler ignores text from `//` to end of line



Identifiers - Symbolic Names

- Identifiers are used to name classes, variables, and methods.
 - Must start with a "Java letter"
 - A - Z, a - z, `_`, `$`
 - Can contain essentially any number of Java letters and digits, but no spaces
 - Case sensitive!!
 - *Number1* and *number1* are different!
 - Cannot be keywords or reserved words



Program Building Blocks

- Statement
 - Performs some action
 - Terminates with a semicolon (;)
 - Can span multiple lines
- Block
 - 1 or more statements
 - Begins and ends with curly braces { }
 - Can be used anywhere a statement is allowed.



White Space Characters

- Space, tab, newline are white space characters
- At least one white space character is required between a keyword and identifier
- Any amount of white space characters are permitted between identifiers, keywords, operators, and literals



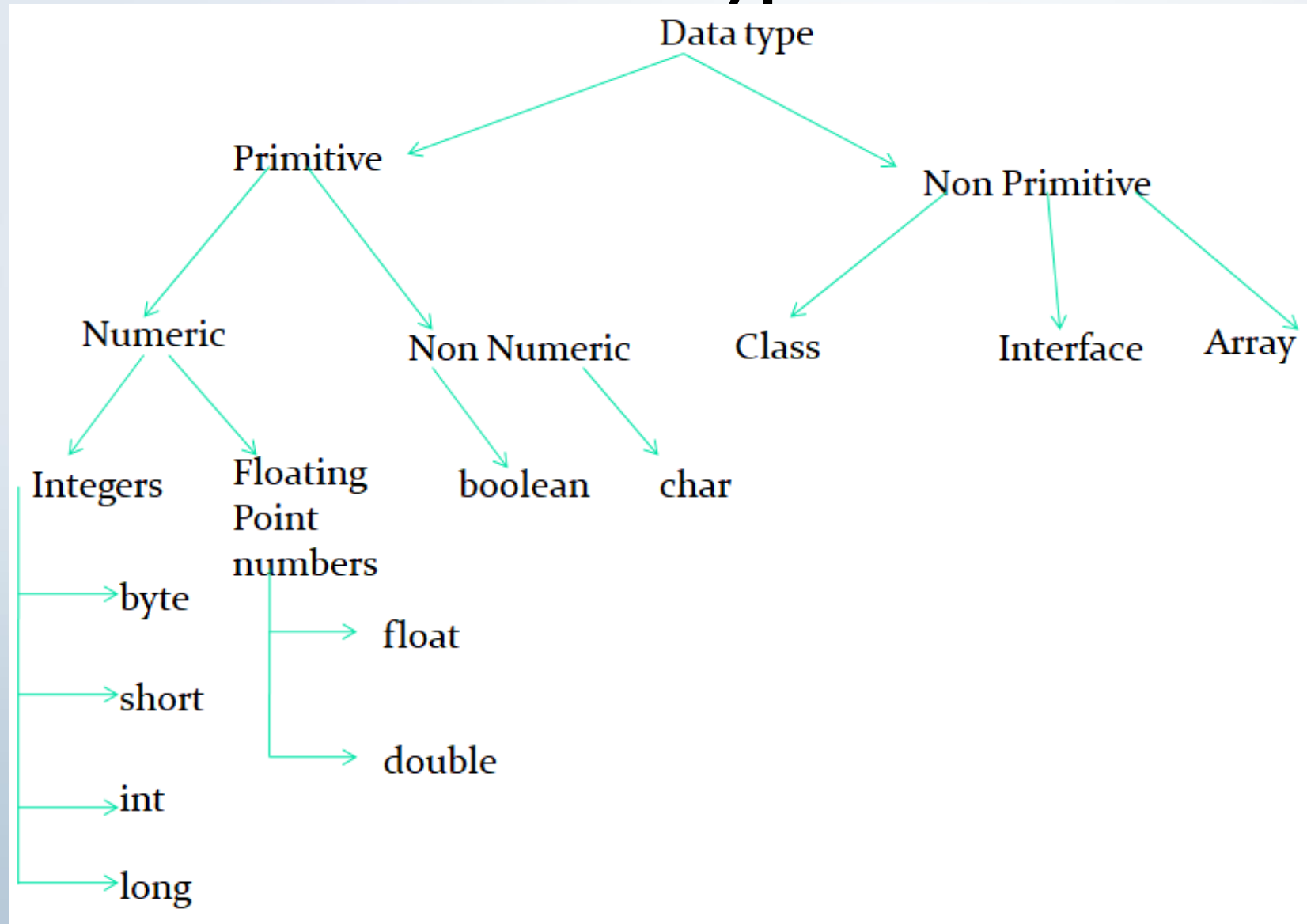
Data Types

- For all data, assign a name (identifier) and a data type
- Data type tells compiler:
 - How much memory to allocate
 - Format in which to store data
 - Type of operations to perform on data
- Compiler monitors use of data
 - Java is a "strongly typed" language
- Java "primitive data types"

byte, short, int, long, float, double, char, boolean



Java Data Types





Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values



Variable Declaration

- The Java programming language uses both "**fields**" and "**variables**" as part of its terminology.
- **Instance variables** (non-static fields) are unique to each instance of a class.
- **Class variables** (static fields) are fields declared with the static modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated.
- **Local variables** store temporary state inside a method.
- **Parameters** are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields").
- When naming your fields or variables, there are rules and conventions that you should (or must) follow.



Declaring Variables

- Variables hold one value at a time, but that value can change
- Syntax:

dataType identifier;

or

dataType identifier₁, identifier₂, ...;



```
byte val = 100;  
System.out.println(val);
```

```
long val1 = 150000000000L;  
System.out.println(val1);
```

```
float val2 = 5.75f;  
System.out.println(val2);
```

```
boolean isJavaFun = true;  
boolean isClassInteresting = false;
```

```
String greeting = "Hello World";  
System.out.println(greeting);
```

A String in Java is actually a **non-primitive** data type, because it refers to an object.



char Data Type

- One Unicode character (16 bits - 2 bytes)

Type	Size in Bytes
------	---------------

<i>char</i>	2
-------------	---



Default Values for Data types

The following chart summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false



Literals

Literal in Java denotes a value

- Integer literals
- Floating point literals
- *Char Literals*

Integer 0 1 42 -23795
02 077 0123
0x0 0x2a
0b1010
123_456_789
365L 077L

*Floating
point* 1.0 4.2 .47
1.22e19 4.61E-9
6.2f 6.21F

Character 'a' '\n' '\t'



String

- Text contained within double quotes.
- Must begin and end on the same line
- String concatenation operator is '+'

String "Hello, world"



Operators



Expressions and Arithmetic Operators

- The Assignment Operator and Expressions
- Arithmetic Operators
- Operator Precedence
- Integer Division and Modulus
- Division by Zero
- Mixed-Type Arithmetic and Type Casting
- Shortcut Operators



Operators	Precedence
postfix	<code>expr++ expr--</code>
unary	<code>++expr --expr +expr -expr ~ !</code>
multiplicative	<code>* / %</code>
additive	<code>+ -</code>
shift	<code><< >> >>></code>
relational	<code>< > <= >= instanceof</code>
equality	<code>== !=</code>
bitwise AND	<code>&</code>
bitwise exclusive OR	<code>^</code>
bitwise inclusive OR	<code> </code>
logical AND	<code>&&</code>
logical OR	<code> </code>
ternary	<code>? :</code>
assignment	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>



Assignment Operator

Syntax:

```
target = expression;
```

Example:

```
x=10;
```

expression: operators and operands that evaluate to a single value

- value is then assigned to target

- target must be a variable (or constant)

- value must be compatible with target's data type



Arithmetic Operators

- The Java programming language provides operators that perform addition, subtraction, multiplication, and division.

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator



The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand.

== equal to

!= not equal to

> greater than

>= greater than or equal to

< less than

<= less than or equal to



Mixed-Type Arithmetic

- When performing calculations with operands of different data types:
 - Lower-precision operands are promoted to higher-precision data types, then the operation is performed
 - Promotion is effective only for expression evaluation; not a permanent change
 - Called "implicit type casting"
- Any expression involving a floating-point operand will have a floating-point result.



Rules of Promotion

Applies the first of these rules that fits:

1. If one operand has the *double* type, the other operand is converted to *double*.
2. If one operand has the *float* type, the other operand is converted to *float*.
3. If one operand has the *long* type, the other operand is converted to *long*.
4. If one operand has the *int* type, the other operand is converted to *int*.



Explicit Type Casting

- **Syntax:**

`(targetType) value`

eg:

```
float a=20.5f;
```

```
int b;
```

```
b = (int) a;
```



Compound Assignment Operators

Operators	Usage	Equivalent Expression
<code>+=</code>	<code>a += 3;</code>	<code>a = a + 3;</code>
<code>-=</code>	<code>a -= 10;</code>	<code>a = a - 10;</code>
<code>*=</code>	<code>a *= 4;</code>	<code>a = a * 4;</code>
<code>/=</code>	<code>a /= 7;</code>	<code>a = a / 7;</code>
<code>%=</code>	<code>a %= 10;</code>	<code>a = a % 10;</code>



Unary Operators

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

- The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.



```
class UnaryDemo {  
    public static void main(String[] args)  
    {  
        int result = +1;    // result is now 1  
        System.out.println(result);  
        result--;    // result is now 0  
        System.out.println(result);  
        result++;    // result is now 1  
        System.out.println(result);  
        result = -result;    // result is now -1  
        System.out.println(result);  
        boolean success = false;    // false  
        System.out.println(success);    // true  
        System.out.println(!success);  
    }  
}
```

```
class PrePostDemo {  
    public static void main(String[] args)  
    {  
        int i = 3;  
        i++;  
        System.out.println(i); // prints 4  
  
        ++i;  
        System.out.println(i); // prints 5  
  
        System.out.println(++i); // prints 6  
  
        System.out.println(i++); // prints 6  
  
        System.out.println(i); // prints 7  
    }  
}
```




`++` increment by 1

`--` decrement by 1

Example:

```
count++;    // count = count + 1;
```

```
count--;    // count = count - 1;
```

Postfix version (`var++`, `var--`)

Prefix version (`++var`, `--var`)



Logical/ Conditional Operators

- The && and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.
- Binary operator (requires two operands)
- Unary operator (requires just one operand)
- | Symbol | Description |
|--------|-----------------------|
| && | and (binary operator) |
| | or (binary operator) |
| ! | not (unary operator) |



```
class Conditional1 {
```

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

```
        int value1 = 1;
```

```
        int value2 = 2;
```

```
        if((value1 == 1) && (value2 == 2))
```

```
        { System.out.println("value1 is 1 AND value2 is 2"); }
```

```
        if((value1 == 1) || (value2 == 1))
```

```
        { System.out.println("value1 is 1 OR value2 is 1"); }
```

```
    }
```

```
}
```



- Another conditional operator is **`?:`**, which can be thought of as **shorthand for an if-then-else statement** (discussed in the Control Flow Statements section). This operator is also known as the **ternary operator** because it uses **three operands**. In the following example, this operator should be read as: "If someCondition is true, assign the value of value1 to result. Otherwise, assign the value of value2 to result."



```
class Conditional2 {  
  
    public static void main(String[] args){  
        int value1 = 1;  
        int value2 = 2;  
        int result;  
        result = value1 < value2 ? value1 : value2;  
        System.out.println(result);  
    }  
}
```



Bitwise operators

- **&** **bitwise AND**
- **|** **bitwise OR**
- **^** **bitwise Ex-OR**
- **Ternary operator ?:**

```
class bitAND {  
    public static void main(String[] args) {  
  
        int number1 = 12, number2 = 25, result;  
  
        // bitwise AND between 12 and 25  
        result = number1 & number2;  
        System.out.println(result); // prints 8  
    }  
}
```

```
class bitOR {  
    public static void main(String[] args) {  
  
        int number1 = 12, number2 = 25, result;  
  
        // bitwise OR between 12 and 25  
        result = number1 | number2;  
        System.out.println(result); // prints 29  
    }  
}
```

```
class bitCompliment {  
    public static void main(String[] args) {  
  
        int number = 35, result;  
  
        // bitwise complement of 35  
        result = ~number;  
        System.out.println(result); // prints -36 54  
    }  
}
```



Bitwise Compliments

- The bitwise complement operator is a unary operator (works with only one operand). It is denoted by ~
- It changes binary digits 1 to 0 and 0 to 1.

It is important to note that the bitwise complement of any integer N is equal to $-(N + 1)$. For example,

Consider an integer 35. As per the rule, the bitwise complement of 35 should be $-(35 + 1) = -36$. Now let's see if we get the correct answer or not.

In the above example, we get that the bitwise complement of 00100011 (35) is 11011100. Here, if we convert the result into decimal we get 220.

However, it is important to note that we cannot directly convert the result into decimal and get the desired output. This is because the binary result 11011100 is also equivalent to -36.

35 = 00100011 (In Binary)

// using bitwise complement operator
~ 00100011

11011100

```
class bitCompliment {  
    public static void main(String[] args) {  
  
        int number = 35, result;  
  
        // bitwise complement of 35  
        result = ~number;  
        System.out.println(result); // prints -36  
    }  
}
```



Operator Precedence

Operator	Order of evaluation	Operation
()	left - right	parenthesis for explicit grouping
* / %	left - right	multiplication, division, modulus
+ -	left - right	addition, subtraction
=	right - left	assignment



Command Line Arguments

- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the String array passed to `main()`.
- A Java application can accept any number of arguments from the command line.
- The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run



```
public class SayHello {  
    public static void main(String args[]) {  
        if (args.length > 0)  
            System.out.println("Hi " + args[0]);  
        else  
            System.out.println("Hi");  
    }  
}
```

java SayHello Sunitha



Control Statements



Control statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code.

- **The if-then Statements**
- **The if-then-else Statement**
- **The switch Statement**



Decision : if
switch

Iteration/ : for
loops while
do...while

Jump : break
continue
return



If , if else

- if (<conditional expression>
 <statement>
- if (<conditional expression>
 <statement1>
 else
 <statement2>



Nested if

```
if (<conditional expression>
{
    if (<conditional expression>
    {
    }
    }
else
{
}
```



else if ladder

```
if (<cond. expr.1>
    <statement 1>
else if (<cond. expr.2>)
    < statement 2>
...
else if (<cond. expr. n>)
    < statement n>
else
    < statement else>
```




switch.... case

```
switch ( <expr.> ) {  
    case <const. expr. 1> : <statement 1>  
    case < const. expr. 2> : < statement 2>  
        :  
        :  
    case < const. expr. n> : < statement n>  
    default : < statement>  
}
```



Loops

- **while**

```
while (expression) {  
    statement(s)  
}
```

- **do-while**

```
do {  
    statement(s)  
} while (expression);
```

```
class WhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        while (count < 11) {  
            System.out.println("Count is: " + count);  
            count++;  
        }  
    }  
}
```

```
class DoWhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        do {  
            System.out.println("Count is: " + count);  
            count++;  
        } while (count < 11);  
    }  
}
```



Loops

- **for**

for (initialization; termination; increment/decrement) {
statement(s)
}

- **for each**

```
class EnhancedForDemo {  
    public static void main(String[] args){  
        int[] numbers =  
            {1,2,3,4,5,6,7,8,9,10};  
        for (int item : numbers) {  
            System.out.println("Count is: " + item);  
        }  
    }  
}
```


```
class ForDemo {  
    public static void main(String[] args){  
        for(int i=1; i<11; i++){  
            System.out.println("Count is: " + i);  
        }  
    }  
}
```



Jump statements

- Break
 - To move control out of the block

```
int i = 1;
while (i<10) {
    if (i == 3)
        break;
    System.out.println("This is a " + i + " iteration");
    ++i;
}
```





Jump - continue

- To move control to the start of next iteration

```
for (i=1; i<=5; ++i) {  
    if (i % 2 == 0)  
        continue;  
    System.out.println("This is a " + i + " iteration");  
}
```



Jump - return

- To explicitly return from a method
- Program control transfers back to the caller of the method