UNIT-1 Introduction

Prepared By:
Mr. Deepak Kumar Sharma
Asst. Professor (Senior Scale)
Systems Cluster
UPES Dehradun

UNIT-I

- History of Java
- Java buzzwords
- Java Tokens/lexical elements
- Data types
- Variables
- Simple java program
- scope and life time of variables
- Operators, expressions, control statements
- Type conversion and casting
- Arrays
- Classes and objects – concepts of classes, objects
- Constructors, methods
- Packages
- Access control
- This keyword
- Overloading methods and constructors
- Parameter passing
- Recursion
- Static
- Garbage collection and finalize()

History of Java

- Java started out as a research project.

- Research began in 1991 as the Green Project at Sun Microsystems, Inc.

- Research efforts birthed a new language, OAK. (A tree outside of the window of James Gosling's office at Sun).

- It was developed as an embedded programming language, which would enable embedded system application.

- It was not really created as web programming language.

Deepak Sharma, UPES Dehradun

History of Java (contd...)

Language was created with 5 main goals:

➢ It should be object oriented.

➢ A single representation of a program could be executed on multiple operating systems. (i.e. write once, run anywhere)

➢ It should fully support network programming.

➢ It should execute code from remote sources securely.

➢ It should be easy to use.

- Oak was renamed Java in 1994.

- Now Sun Microsystems is a subsidiary of Oracle Corporation.
- Java is available as *jdk* and it is an open source s/w.

Deepak Sharma, UPES Dehradun

James Gosling

Green Team

Java Logo

Versions of Java

Version	Release date	End of Free Public Updates[3][8][9][10]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	September 2003	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
Java SE 5	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018 December 2026 for Azul[11]
Java SE 7	July 2011	July 2019	July 2022
Java SE 8 (LTS)	March 2014	**March 2022 for Oracle (commercial)** December 2030 for Oracle (non-commercial) December 2030 for Azul May 2026 for IBM Semeru[12] At least May 2026 for Eclipse Adoptium At least May 2026 for Amazon Corretto	December 2030[13]
Java SE 9	September 2017	March 2018 for OpenJDK	—
Java SE 10	March 2018	September 2018 for OpenJDK	—

Java SE 11 (LTS)	September 2018	September 2026 for Azul October 2024 for IBM Semeru[12] At least October 2024 for Eclipse Adoptium At least September 2027 for Amazon Corretto At least October 2024 for Microsoft[14][15]	September 2026 September 2026 for Azul[11]
Java SE 12	March 2019	September 2019 for OpenJDK	—
Java SE 13	September 2019	March 2020 for OpenJDK	—
Java SE 14	March 2020	September 2020 for OpenJDK	—
Java SE 15	September 2020	March 2021 for OpenJDK March 2023 for Azul[11]	—
Java SE 16	March 2021	September 2021 for OpenJDK	—
Java SE 17 (LTS)	September 2021	September 2029 for Azul At least September 2027 for Microsoft[14] At least September 2027 for Eclipse Adoptium	September 2029 or later September 2029 for Azul
Java SE 18	March 2022	September 2022 for OpenJDK and Adoptium	—
Java SE 19	September 2022	March 2023 for OpenJDK	—
Java SE 20	March 2023	September 2023 for OpenJDK	—
Java SE 21 (LTS)	September 2023	September 2028	September 2031[13]

Features of Java (Java Buzz Words)

- Simple
- Object Oriented
- Compiled, Interpreted and High Performance
- Portable
- Reliable
- Secure
- Multithreaded
- Dynamic
- Distributed
- Architecture-Neutral

Java Features

- **Simple**
  - No pointers
  - Automatic garbage collection
  - Rich pre-defined class library

- **Object Oriented**
  - Focus on the data (objects) and methods manipulating the data
  - All methods are associated with objects
  - Potentially better code organization and reuse

Java Features

- **Compile, Interpreted and High Performance**
  - Java compiler generate byte-codes, not native machine code
  - The compiled byte-codes are platform-independent
  - Java byte codes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)
  - Easy to translate directly into native machine code by using a just-in-time compiler.
- **Portable**
  - Same application runs on all platforms
  - The sizes of the primitive data types are always the same
  - The libraries define portable interfaces

Java Features

- **Reliable/Robust**
  - Extensive compile-time and runtime error checking
  - No pointers but real arrays. Memory corruptions or unauthorized memory accesses are impossible
  - Automatic garbage collection tracks objects usage over time

- **Secure**
  - Java's robustness features makes java secure.
  - Access restrictions are forced (private, public)

Java Features

- **Multithreaded**

  - It supports multithreaded programming.

  - Need not wait for the application to finish one task before beginning another one.

- **Dynamic**

  - Libraries can freely add new methods and instance variables without any effect on their clients

  - Interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented .

Java Features

- **Distributed**
  - Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
  - Allows objects on two different computers to execute procedures remotely by using package called *Remote Method Invocation (RMI)*.

- **Architecture-Neutral**
  - Goal of java designers is "write once; run anywhere, any time, forever."

Java Platforms

There are three main platforms for Java:

- **Java SE** (Java Platform, Standard Edition) – runs on desktops and laptops.

- **Java ME** (Java Platform, Micro Edition) – runs on mobile devices such as cell phones.

- **Java EE** (Java Platform, Enterprise Edition) – runs on servers.

Java Terminology

The Java platform is the name given to the computing platform from Oracle that helps users to run and develop Java applications. The platform consists of two essential softwares:

• Java Runtime Environment (JRE), which is needed to run Java applications and applets;

• Java Development Kit (JDK), which is needed to develop those Java applications and applets

Java Terminology

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. The JVM performs following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

Java Terminology

Java Development Kit:

It contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.

Java Virtual Machine:

An abstract machine architecture specified by the Java Virtual Machine Specification.

It interprets the byte code into the machine code depending upon the underlying OS and hardware combination. JVM is platform **dependent**. (It uses the class libraries, and other supporting files provided in JRE)
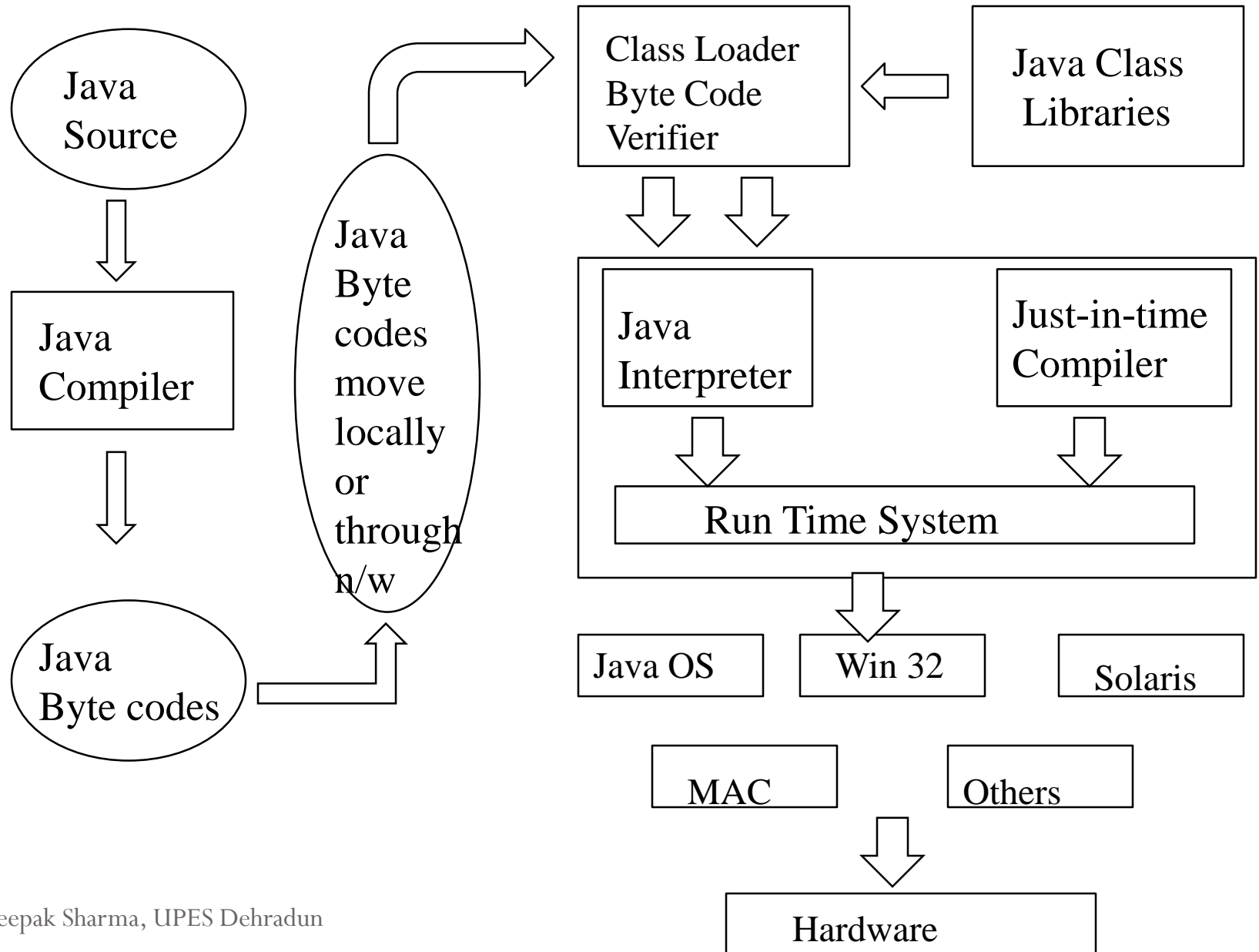
Java Terminology (contd…)

➤ Java Runtime Environment:

A runtime environment which implements Java Virtual Machine, and provides all class libraries and other facilities necessary to execute Java programs. This is the software on your computer that actually runs Java programs.

JRE = JVM + Java Packages Classes (like util, math, lang, awt, swing etc) +runtime libraries.

Byte code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).The JVM is an interpreter for byte code.

Java Execution Procedure

Java Source

Java Compiler

Java Byte codes

Java Byte codes move locally or through n/w

Class Loader Byte Code Verifier

Java Class Libraries

Java Interpreter

Just-in-time Compiler

Run Time System

Java OS

Win 32

Solaris

MAC

Others

Hardware

Binary form of a .class file(partial)

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

```
0000: cafe babe 0000 002e 001a 0a00 0600 0c09 ................
0010: 000d 000e 0800 0f0a 0010 0011 0700 1207 ................
0020: 0013 0100 063c 696e 6974 3e01 0003 2829 .....<init>...()
0030: 5601 0004 436f 6465 0100 046d 6169 6e01 V...Code...main.
0040: 0016 285b 4c6a 6176 612f 6c61 6e67 2f53 ..([Ljava/lang/S
0050: 7472 696e 673b 2956 0c00 0700 0807 0014 tring;)V........
0060: 0c00 1500 1601 000d 4865 6c6c 6f2c 2057 ........Hello, W
0070: 6f72 6c64 2107 0017 0c00 1800 1901 0005 orld!...........
0080: 4865 6c6c 6f01 0010 6a61 7661 2f6c 616e Hello...java/lan
0090: 672f 4f62 6a65 6374 0100 106a 6176 612f g/Object...java/
00a0: 6c61 6e67 2f53 7973 7465 6d01 0003 6f75 lang/System...ou ...
```

Creating hello java example

Let's create the hello java program:

```java
class Simple{
    public static void main(String args[]){
    System.out.println("Hello Java");
    }
}
```
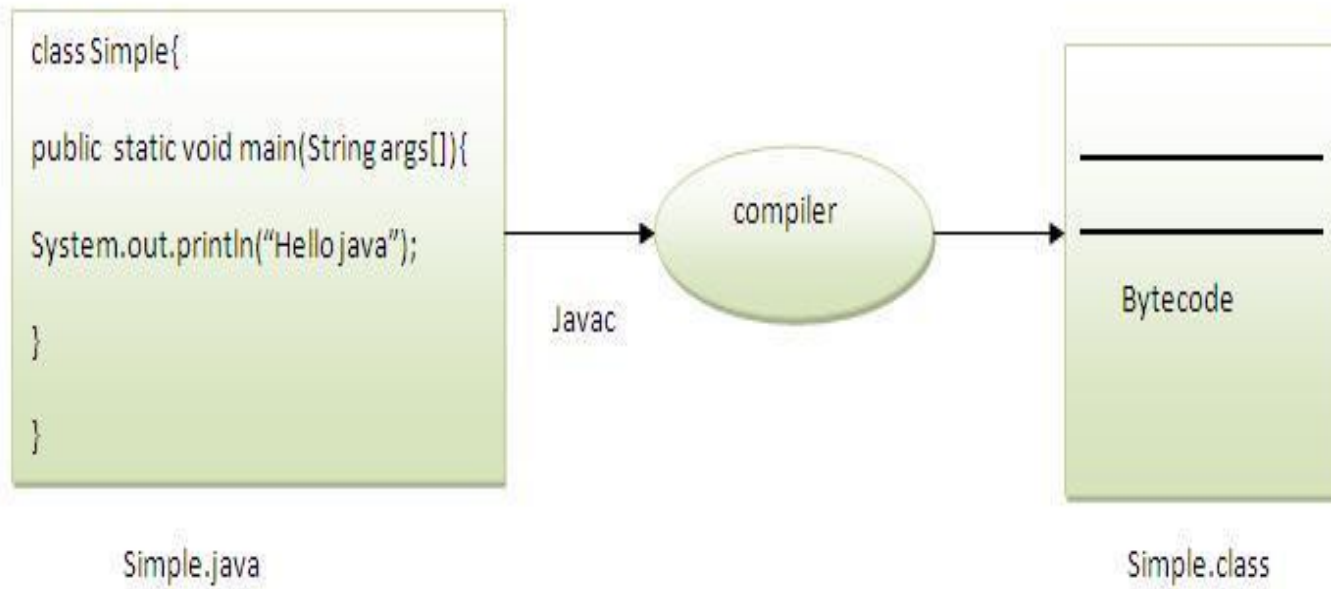
save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Output: Hello Java

At Runtime:


```
class Simple{

public static void main(String args[]){

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

}

}
```

Simple.java → Javac → compiler → Bytecode

Simple.class

Understanding first java program

- **class** keyword is used to declare a class in java.

- **public** keyword is an access modifier which represents visibility, it means it is visible to all.

- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.

- **void** is the return type of the method, it means it doesn't return any value.

- **main** represents startup of the program.

- **String[] args** is used for command line argument.

- **System.out.println()** is used print statement.

Object Oriented Programming Concepts

- Objects
- Classes
- Data abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding

A **class** is collection of objects of similar type or it is a template.

Ex: fruit mango;

class   object

Objects are instances of the type class.

The wrapping up of data and functions into a single unit (called class) is known as **encapsulation**. Data encapsulation is the most striking features of a class.

Abstraction refers to the act of representing essential features without including the background details or explanations

Inheritance is the process by which objects of one class acquire the properties of another class. The concept of inheritance provides the **reusability.**

Polymorphism:

It allows the single method to perform different actions based on the parameters.

Dynamic Binding: When a method is called within a program, it associated with the program at run time rather than at compile time is called dynamic binding.

Benefits of OOP

- Through inheritance, we can **eliminate redundant code** and extend the use of existing classes.

- The principle of data hiding helps the programmer to build **secure** programs.

- It is easy to partition the work in a project based on objects.

- Object oriented system can be easily **upgraded** from small to large systems.

- Software complexity can be easily managed.

Differences b/w C++ and Java

C++	Java
1. Global variable are supported.	1. No Global variables. Everything must be inside the class only.
2. Multiple inheritance is supported.	2. No direct multiple Inheritance.

C++	Java
3.Constructors and Destructors supported.	3.Java supporting constructors only & instead of destructors garbage collection is supported.
4.In c++ pointers are supported.	4.No pointer arithmetic in Java.
5.C++ supporting ASCII character set.	5. Java supports Uni code Character set.

Topic	Java	Python
Compilation process	Java is both compiled and interpreted language. The source code is first compiled and converted to bytecode, and afterward, it depends on JVM whether the bytecode will be compiled or interpreted	Python is an interpreted language, i.e., it is compiled and executed simultaneously line by line.

Length of code	The length of the code of java programs is more as compared to that of Python as every program has to be written in a class. For Eg- to write hello world program, the code is- public class HelloWorld {
	    public static void **main**(String[] args) {
	      System.out.**println**("Hello, World");
	} }
		Python has shorter lines of code as you directly write the code and it gets interpreted.
		For eg-
		print('Hello, world!')

Complexity of syntax	Java is a statically typed programming language. There are hardcore rules for braces and semi-colon.	Python is dynamically typed and there are no hardcore rules for semi-colon and braces. It works on indentation.
Ease of typing	Strongly typed, need to define exact types of variables.	Dynamically typed, no need to define the exact type of variables.
Speed	Java is faster as compared to Python.	Python is relatively slow as it is interpreted language and it determines the type of the variable at run time, which makes it slow.
Usage	It has been in trend for a long time and is vastly used in Android application development, embedded systems, and web applications.	Data science and machine language are made very simple, using Python. Also, it is being used for web development.

Points to remember

- **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** have different meaning in Java.

- **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

- **Example:** *class MyFirstJavaClass*

- **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

- **Example:** *public void myMethodName()*

Points to remember

- **Program File Name** − Name of the program file should match the class name.

Note: In case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.

- When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

- **Example:** Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as '*MyFirstJavaProgram.java*'

- **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Tokens

Java tokens are smallest elements of a program which are identified by the compiler. Tokens in java include:

- Keywords
- Identifiers
- Literals
- Operators
- Separators
- Comments

Keywords

- These are the **pre-defined** reserved words of any programming language.
- Each keyword has a special meaning.
- It is always written in lower case. Some of the keywords are:

01. abstract	02. boolean	03. byte	04. break	05. class
06. case	07. catch	08. char	09. continue	10. default
11. do	12. double	13. else	14. extends	15. final
16. finally	17. float	18. for	19. if	20. implements
21. import	22. instanceof	23. int	24. interface	25. long
26. native	27. new	28. package	29. private	30. protected
31. public	32. return	33. short	34. static	35. super
36. switch	37. synchronized	38. this	39. thro	40. throws
41. transient	42. try	43. void	44. volatile	45. while
46. assert	47. const	48. enum	49. goto	50. strictfp

Identifier

Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows −

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Data Types

- Java Is a Strongly Typed Language
  - Every variable has a type, every expression has a type, and every type is strictly defined.
  - All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
  - There are no automatic conversions of conflicting types as in some languages.
- *For example, in C/C++ you can assign a floating-point value to an integer. In Java, you cannot.*

Data Types

Simple Type

Derived Type

E.g: Array, String…

User Defined Type

Numeric Type

Non-Numeric

class

Interface

Integer

Float

Char

Boolean

float

double

byte   short   int   long

Primitive Data Types

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)($2^7 -1$)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

Primitive Data Types

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^{15})
- Maximum value is 32,767 (inclusive) ($2^{15} - 1$)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

Primitive Data Types

int

- Int data type is a 32-bit signed two's complement integer.

- Minimum value is - 2,147,483,648 (-2^31)

- Maximum value is 2,147,483,647(inclusive) (2^31 -1)

- Integer is generally used as the default data type for integral values unless there is a concern about memory.

- The default value is 0

- Example: int a = 100000, int b = -200000

Primitive Data Types

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^{63})
- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^{63} -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

Primitive Data Types

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

Primitive Data Types

double

- double data type is a double-precision 64-bit IEEE 754 floating point

- This data type is generally used as the default data type for decimal values, generally the default choice

- Double data type should never be used for precise values such as currency

- Default value is 0.0

- Example: double d1 = 123.4

Primitive Data Types

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

Primitive Data Types

char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

Data Types

Name	Width in bits	Range
long	64	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	−2,147,483,648 to 2,147,483,647
short	16	−32,768 to 32,767
byte	8	−128 to 127
double	64	4.9e−324 to 1.8e+308
float	32	1.4e−045 to 3.4e+038
char	16	0 to 65,536.

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

        System.out.println(" For an Integer ");
        System.out.println("Size is : "+Integer.SIZE);

                int i1 = Integer.MAX_VALUE;
                int i2 = Integer.MIN_VALUE ;

        System.out.println("Max value is : "+i1);
        System.out.println("Min  Value is : "+i2);

         System.out.println(" For an Byte");
        System.out.println("Size is : "+Byte.SIZE);

                byte b1 = Byte.MAX_VALUE;
                byte b2 = Byte.MIN_VALUE ;

        System.out.println("Max value is : "+b1);
        System.out.println("Min  Value is : "+b2);

        System.out.println(" For an Short");
        System.out.println("Size is : "+Short.SIZE);

         short s1 = Short.MAX_VALUE;
         short s2 = Short.MIN_VALUE ;

        System.out.println("Max value is : "+s1);
        System.out.println("Min  Value is : "+s2);

        System.out.println(" For an Long");
        System.out.println("Size is : "+Long.SIZE);

        long l1 = Long.MAX_VALUE;
        long l2 = Long.MIN_VALUE ;

        System.out.println("Max value is : "+l1);
        System.out.println("Min  Value is : "+l2);

     }
}
```
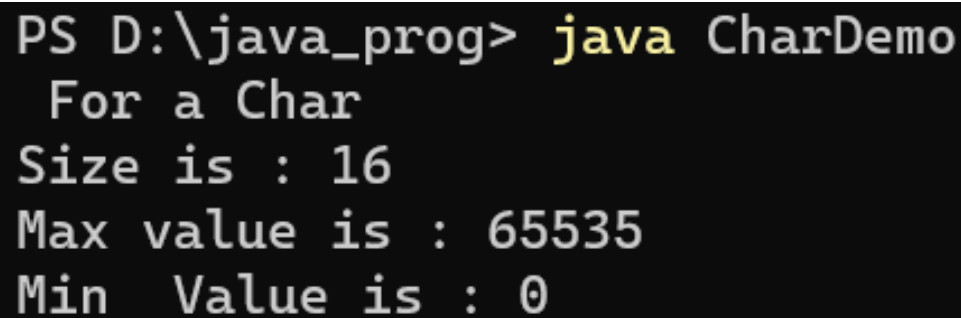
Output

```
PS D:\java_prog> javac IntDemo.java
PS D:\java_prog> java IntDemo
 For an Integer
Size is : 32
Max value is : 2147483647
Min  Value is : -2147483648
 For an Byte
Size is : 8
Max value is : 127
Min  Value is : -128
 For an Short
Size is : 16
Max value is : 32767
Min  Value is : -32768
 For an Long
Size is : 64
Max value is : 9223372036854775807
Min  Value is : -9223372036854775808
```

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

        System.out.println(" For an Float");
        System.out.println("Size is : "+Float.SIZE);

        float f1 = Float.MAX_VALUE;
        float f2 = Float.MIN_VALUE ;

        System.out.println("Max value is : "+f1);
        System.out.println("Min  Value is : "+f2);

        System.out.println(" For an Double");
        System.out.println("Size is : "+Double.SIZE);

        double d1 = Double.MAX_VALUE;
        double d2 = Double.MIN_VALUE ;

        System.out.println("Max value is : "+d1);
        System.out.println("Min  Value is : "+d2);
    }
}
```

```
PS D:\java_prog> java FloatDemo
 For an Float
Size is : 32
Max value is : 3.4028235E38
Min  Value is : 1.4E-45
 For an Double
Size is : 64
Max value is : 1.7976931348623157E308
Min  Value is : 4.9E-324
```

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

        System.out.println(" For a Char");
        System.out.println("Size is : "+Character.SIZE);
        int f1 = Character.MAX_VALUE;
        long f2 = Character.MIN_VALUE ;
        System.out.println("Max value is : "+f1);
        System.out.println("Min Value is : "+f2);
    }
}
```

```
PS D:\java_prog> java CharDemo
 For a Char
Size is : 16
Max value is : 65535
Min  Value is : 0
```

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Java provides five types of literals are as follows:

- Integer
- Floating Point
- Character
- String
- Boolean

Literal	Type
23	int
9.86	double
false, true	boolean
'K', '7', '-'	char
"HelloJava"	String
null	any reference type

- Literals can be assigned to any primitive type variable. E.g.
  byte a = 68;   char a = 'A';
- Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example :
  int decimal = 100;   int octal = 0144;   int hexa = 0x64;

Escape Sequence Characters

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Comments

- Comments allow us to specify information about the program inside our Java code. Java compiler recognizes these comments as tokens but excludes it form further processing. The Java compiler treats comments as whitespaces. Java provides the following two types of comments:

  •**Single Line:** It begins with a pair of forwarding slashes (*//*).

  •**Multiple Line:** It begins with /* and continues until it founds */*.

Separators

- The separators in Java is also known as **punctuators**. There are nine separators in Java, are as follows:

- separator <= ; | , | . | (|) | { | } | [|]

- **Square Brackets []:** It is used to define array elements. A pair of square brackets represents the single-dimensional array, two pairs of square brackets represent the two-dimensional array.

- **Parentheses ():** It is used to call the functions and parsing the parameters.

- **Curly Braces {}:** The curly braces denote the starting and ending of a code block.

- **Comma (,):** It is used to separate two values, statements, and parameters.

- **Assignment Operator (=):** It is used to assign a variable and constant.

- **Semicolon (;):** It is the symbol that can be found at end of the statements. It separates the two statements.

- **Period (.):** It separates the package name form the sub-packages and class. It also separates a variable or method from a reference variable.

Operators

- Arithmetic Operators

- Relational Operators

- Bitwise Operators

- Logical Operators

- Assignment Operators

- Misc Operators

Arithmetic Operators(1)

Operator	Result
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulus

Deepak Sharma, UPES Dehradun

Arithmetic Operators(2)

Operator	Result
++	Increment
+=	Addition assignment
− =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
− −	Decrement

Example:

```
class IncDec{
    public static void main(String args[]){
     int a = 1;
     int b = 2;
     int c,d;
     c = ++b;
     d = a++;
     c++;
     System.out.println("a = " + a);
     System.out.println("b = " + b);
     System.out.println("c = " + c);
     System.out.println("d = " + d);
    }
}
```

```java
class OpEquals{
    public static void main(String args[]){
     int a = 1;
     int b = 2;
     int c = 3;
      a += 5;
      b *= 4;
      c += a * b;
      c %= 6;
     System.out.println("a = " + a);
     System.out.println("b = " + b);
     System.out.println("c = " + c);
    }
}
```

Bitwise Operators(1)

➢ *bitwise operators* can be applied to the integer types, **long**, **int**, **short, byte** and **char.**

➢ These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
<<	Shift left

Bitwise Operators(2)

Operator	Result
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
<<=	Shift left assignment

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
  00101010      42              00101010      42
 &00001111      15             ^00001111      15
 --------------                -------------
  00001010      10              00100101      37
```

int a = 64;
a = a << 2;   256

int a = 32;
a = a >> 2;   8

Relational Operators

➢ The *relational operators* determine the relationship that one operand has to the other.

➢ They determine equality and ordering.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example:

```java
public class RelationalOperatorsDemo
{
  public static void main(String args[])
  {
    int x=10,y=5;
    System.out.println("x>y:"+(x>y));
    System.out.println("x<y:"+(x<y));
    System.out.println("x>=y:"+(x>=y));
    System.out.println("x<=y:"+(x<=y));
    System.out.println("x==y:"+(x==y));
    System.out.println("x!=y:"+(x!=y));
  }
}
```

Boolean Logical Operators(1)

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

&&   // Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

```java
class Test{
        public static void main(String args[]){
                int a=5,b=20;
                if ( a+b>40 && ++b< 10)
                        System.out.println("Hi");
                System.out.println("b="+b);
        }
}
```

```java
public class TernaryOperatorDemo
{
 public static void main(String args[])
  {
    int x=10,y=12;
    int z;
    z= x > y ? x : y;
    System.out.println("Z="+z);
}
```

Expressions

- An expression is a combination of constants (like 10), operators (like $+$), variables(section of memory) and parentheses (like "(" and ")") used to calculate a value.

        Ex1:   x = 1;

        Ex2:   y = 100 + x;

        Ex3:   x = (32 - y) / (x + 5)

Java Operator Precedence Table

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Left to Right
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Note: Larger number means higher precedence.

Control Statements

- Selection Statements: if & switch

- Iteration Statements: for, while, do-while, for each

- Jump Statements: break, continue and return

Selection Statements

if (*condition*)

    statement1;

else

    statement2;

if(*condition*)

    statement;

else if(*condition*)

    statement;

else if(*condition*)

    statement;

...

else

statement;

switch (*expression*)

{

    case *value1:*

  // statement sequence

    break;

    case *value2:*

  // statement sequence

    break;

  …

    case *valueN:*

  // statement sequence

    break;

    default:

  // default statement sequence

}

The *condition* is any expression that returns a **boolean** value.

Switch(Expression):

The *expression* must be of type **byte**, **short**, **int**, or **char**;
Each of the *values* specified in the **case** statements must be of a type
compatible with the expression.

Iteration Statements

while(*condition*)

{

   // body of loop

}

do

{

   // body of loop

} while (*condition);*

for(*initialization; condition; iteration*)

  {

    // body

  }

For-each loop

•It starts with the keyword **for** like a normal for-loop.

•Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.

•In the loop body, you can use the loop variable you created rather than using an indexed array element.

•It's commonly used to iterate over an array or a Collections class (eg, ArrayList)

Syntax:

```
for (type var : array)
{
    statements using var;
}
```

➡️

```
for (int i=0; i<arr.length; i++)
{
    type var = arr[i];
    statements using var;
}
```

Example: For Each Loop

```java
class For_Each
{
    public static void main(String[] arg) {
        int[] marks = { 125, 172, 95, 116, 110 };
        int highest_marks = marks[0];
        for (int num : marks)
        {
            if (num > highest_marks)
            {
                highest_marks = num;
            }
        }
        System.out.println("The highest score is " + highest_marks);
    }
}
```

OUTPUT: The highest score is 172

Jump Statements

continue;   //bypass the followed instructions

break;   //exit from the loop

label:
- - - -

- - - -

break label;   //it's like goto
statement

return;   //control returns to the caller

Sample Program

```
class HelloWorld {
    public static void main (String args []) {
    System.out.println ("Welcome to Java Programming…..");
    }
}
```

public allows the program to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared. In this case, main ( ) must be declared as public, since it must be called by code outside of its class when the program is started.

static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main ( ) is called by the Java interpreter before any objects are made.

void states that the main method will not return any value.

main() is called when a Java application begins. In order to run a class, the class must have a main method.

string args[] declares a parameter named args, which is an array of String. In this case, args receives any command-line arguments present when the program is executed.

System is a class which is present in **java.lang** package.

out is a static field present in System class which returns a **PrintStream** object. As out is a static field it can call directly with classname.

println() is a method which present in PrintStream class which can call through the PrintStream object return by static field **out** present in System class to print a line to console.

Classes in Java

```java
public class Dog {

    String breed;
    int age;
    String color;

    void barking() {
    }

    void hungry() {
    }

    void sleeping() {
    }
}
```

Java Class & Objects

Class	**Person**
Data Members	**unique_id** name age city gender
Methods	eat() study() sleep() play()

name- John
age- 35
city- Delhi
gender- male

name- Dessy
age- 20
city- Pune
gender- female
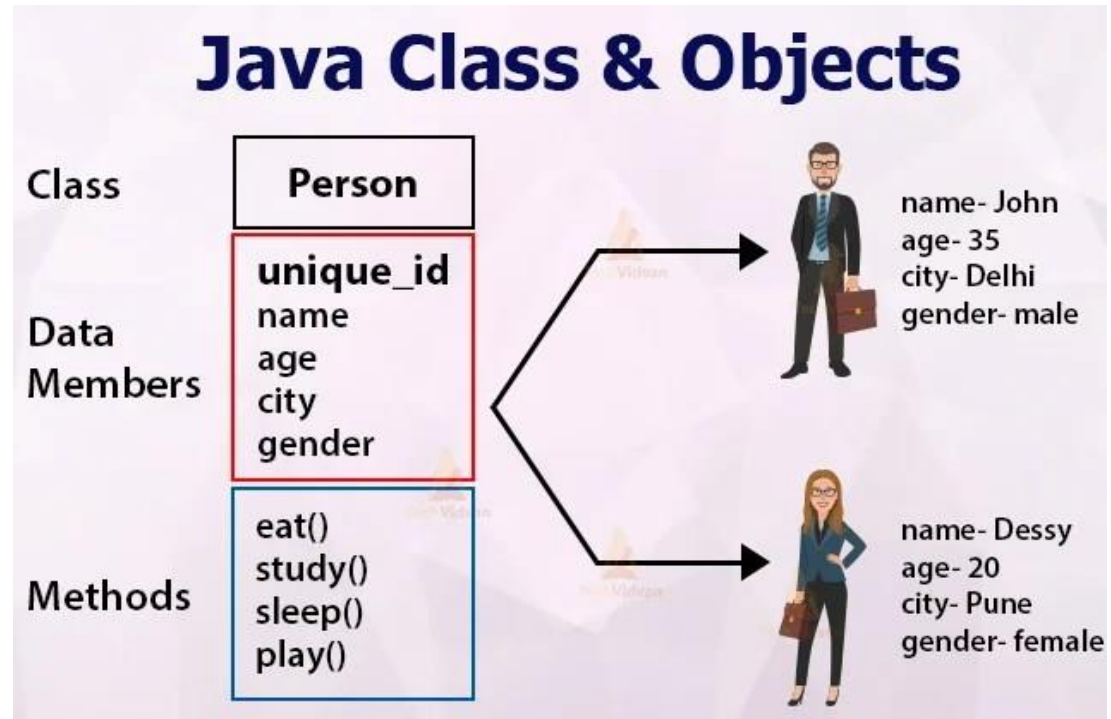
Constructor

- Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

- Each time a new object is created, at least one constructor will be invoked.

- The main rule of constructors is that they should have the same name as the class.

- A class can have more than one constructor.

```
public class Puppy {

   public Puppy() {
   }

   public Puppy(String name) {
      // This constructor has one parameter,
name.
   }
}
```

Creating Objects

```
class sample{
    public static void main(String args[]){
        System.out.println("sample:main");
        sample s=new sample();
        s.display();


    }
    void display(){
        System.out.println("display:main");
    }
}
```

Creating Objects

There are three steps when creating an object from a class −

- Declaration − A variable declaration with a variable name with an object type.

- Instantiation − The 'new' keyword is used to create the object.

- Initialization − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

```java
public class Puppy {
  public Puppy(String name) {
    // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name );
  }

  public static void main(String []args) {
    // Following statement would create an object myPuppy
    Puppy myPuppy = new Puppy( "tommy" );
  }
}
```

Variables

➤ The variable is the basic unit of storage in a Java program.

➤ A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable

➤ In Java, all variables must be declared before they can be used.

type identifier [= value][, identifier [= value] ...] ;

Types

➤ Instance Variable

➤ Class/Static Variable

➤ Local Variable

Local variables :

- Local variables are declared in methods, constructors, or blocks.

- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.

- Access modifiers cannot be used for local variables.

- Local variables are visible only within the declared method, constructor or block.

- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example

```java
public class Test {
  public void pupAge() {
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }
  public static void main(String args[]) {
    Test test = new Test();
    test.pupAge();
  }
}
```

Output
Puppy age is: 7

Example

```
public class Test {
  public void pupAge() {
    int age;
    age = age + 10;
    System.out.println("Puppy age is : " + age);
  }

  public static void main(String args[]) {
    Test test = new Test();
    test.pupAge();
  }
}
```

Output
Test.java:4:variable number might not have been initialized
age = age + 10;
        ^
1 error

Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.

- Instance variables are created when an object is created with the use of the key word **'new'** and destroyed when the object is destroyed.

- Access modifiers can be given for instance variables.

- The instance variables are visible for all methods, constructors and block in the class.

- Instance variables have default values.

- Instance variables can be accessed directly by calling the variable name inside the class.

- However within static methods and different class (when instance variables are given accessibility) that should be called using the fully qualified name **ObjectReference.VariableName**

Class/Static variables :

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.

- There would only be one copy of each class variable per class, regardless of how many objects are created from it.

- Static variables are stored in static memory.

- Static variables are created when the program starts and destroyed when the program stops.

- Visibility is similar to instance variables.

- Default values are same as instance variables.

- Static variables can be accessed by calling with the class name **ClassName.VariableName**

```java
class Variables{
    int i;//instance variable
    public int j ;//instance variable
    static long l=10;//class variable
    public static float f;//class variable
    char c;//instance variable
    boolean b;//instance variable
    void display(int a){
            i=a;
            System.out.println("i value in display: "+i);
    }
    public static void main(String args[]){
            double d=0.0;//local varible
            //public double d=0.0; invalid
            Variables v1=new Variables();
            Variables v2=new Variables();
            Variables v3=new Variables();
            v1.display(100);
            v1.i=2;
            v2.i=3;
            v3.i=4;
            System.out.println("i value is: "+v1.i);
            System.out.println("i value is: "+v2.i);
            System.out.println("i value is: "+v3.i);
            System.out.println("i value is: "+v1.j);
            v1.l=20;
            v2.l=30;
            v3.l=40;
            System.out.println("l value is: "+v1.l);
            System.out.println("l value is: "+v2.l);
            System.out.println("l value is: "+v3.l);

            System.out.println("f value is: "+f);
            System.out.println("c value is: "+v1.c);
            System.out.println("b value is: "+v1.b);

            System.out.println("d value is: "+d);
    }
}
```

The Scope and Lifetime of Variables

Scope

The *scope* of a declared element is the portion of the program where the element is visible.

Lifetime

The *lifetime* of a declared element is the period of time during which it is alive.

The lifetime of the variable can be determined by looking at the context in which they're defined.

➢ Java allows variables to be declared within any block.

➢ A block begins with an opening curly brace and ends by a closing curly brace.

- Variables declared inside a scope are not accessible to code outside.

- Scopes can be nested. The outer scope encloses the inner scope.

- Variables declared in the outer scope are visible to the inner scope.

- Variables declared in the inner scope are not visible to the outside scope.

```java
public class Scope
{
    public static void main(String args[]){
            int x;  //known to all code within main
            x=10;
            if(x==10){ // starts new scope
                    int y=20; //Known only to this block
                    //x and y are both known here
                    System.out.println("x and y: "+x+" "+y);
                    x=y+2;
            }
            // y=100; // error ! y not known here
            //x is still known here
            System.out.println("x is "+x);
    }
}
```

CheckPoint: Using Different types of variables

```java
public class HelloWorld {
 static int cvar;
 int ivar;
 public static void main(String[] args) {
        int lvar=10;
        System.out.println("class variable="+cvar+"\nlocal variable="+lvar);
}
}
```

Output:
class variable=0
local variable=10

CheckPoint: accessing instance var

```java
public class HelloWorld {
 static int cvar;
 int ivar;
 public static void main(String[] args) {
          int lvar=10;
          System.out.println("class variable="+cvar+"\nlocal variable="+lvar);
          System.out.println("Instance Variable="+ivar);
}
}
```

Output:
/HelloWorld.java:7: error: non-static variable ivar cannot be referenced from a static context

System.out.println("Instance Variable="+ivar);
                                        ^

1 error

CheckPoint: Creating object

```java
public class HelloWorld {
 static int cvar;
 int ivar;
 public static void main(String[] args) {
         int lvar=10;
         System.out.println("class variable="+cvar+"\nlocal variable="+lvar);

         HelloWorld obj=new HelloWorld();

         System.out.println("Instance Variable="+obj.ivar);
 }
}
```

Output:
class variable=0
local variable=10
Instance Variable=0

CheckPoint: Using method

```java
public class HelloWorld {
 static int cvar;
 int ivar;
//Method added
void display()    {
        System.out.println("Instance Variable (from display() method)="+ ivar);
}

public static void main(String[] args) {
        int lvar=10;
        System.out.println("class variable="+cvar+"\nlocal variable="+lvar);
        HelloWorld obj=new HelloWorld();

        System.out.println("Instance Variable (from main)="+ obj.ivar);
        obj.display();   //Method Invoked
}
}
```

Output:
class variable=0
local variable=10
Instance Variable (from main)=0
Instance Variable (from display() method)=0

CheckPoint: Using Constructor

```java
public class HelloWorld {
 static int cvar;
 int ivar;
//Method added
void display()    {
        System.out.println("Instance Variable (from display() method)="+ ivar);    }
//Constructor
HelloWorld() {
        ivar=5638;

}
public static void main(String[] args) {
        int lvar=10;
        System.out.println("class variable="+cvar+"\nlocal variable="+lvar);
        HelloWorld obj=new HelloWorld();
        System.out.println("Instance Variable (from main)="+ obj.ivar);
        obj.display();   //Method Invoked
}
}
```

Output:
class variable=0
local variable=10
Instance Variable (from main)=5638
Instance Variable (from display() method)=5638

CheckPoint: Creating Multiple classes

class HelloWorld {
static int cvar;
int ivar;
void display()   {
        System.out.println("Instance Variable (from display() method)="+ ivar);   }
HelloWorld()   {   ivar=5638;   }
}

public class Check{
     public static void main(String[] args) {
     int lvar=10;
     System.out.println("class variable="+cvar+"\nlocal variable="+lvar);
     HelloWorld obj=new HelloWorld();
     System.out.println("Instance Variable (from main)="+ obj.ivar);
     obj.display();
}
}

```
Output:
/Check.java:17: error: cannot find symbol
System.out.println("class variable="+cvar+"\nlocal variable="+lvar);
                                       ^
symbol:   variable cvar
location: class Check
1 error
```

CheckPoint: Creating Multiple classes

```java
class HelloWorld {
static int cvar;
int ivar;
void display()    {
        System.out.println("Instance Variable (from display() method)="+ ivar);    }
HelloWorld()    {       ivar=5638;    }
}


public class Check{
    public static void main(String[] args) {
    int lvar=10;
    System.out.println("class variable="+HelloWorld.cvar+"\nlocal variable="+lvar);
    HelloWorld obj=new HelloWorld();
    System.out.println("Instance Variable (from main)="+ obj.ivar);
    obj.display();
}
}
```

Output:
class variable=0
local variable=10
Instance Variable (from main)=5638
Instance Variable (from display() method)=5638

Java Packages

- A **java package** is a group of similar types of classes, interfaces and sub-packages.

- Package in java can be categorized in two form, built-in package and user-defined package.

- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.
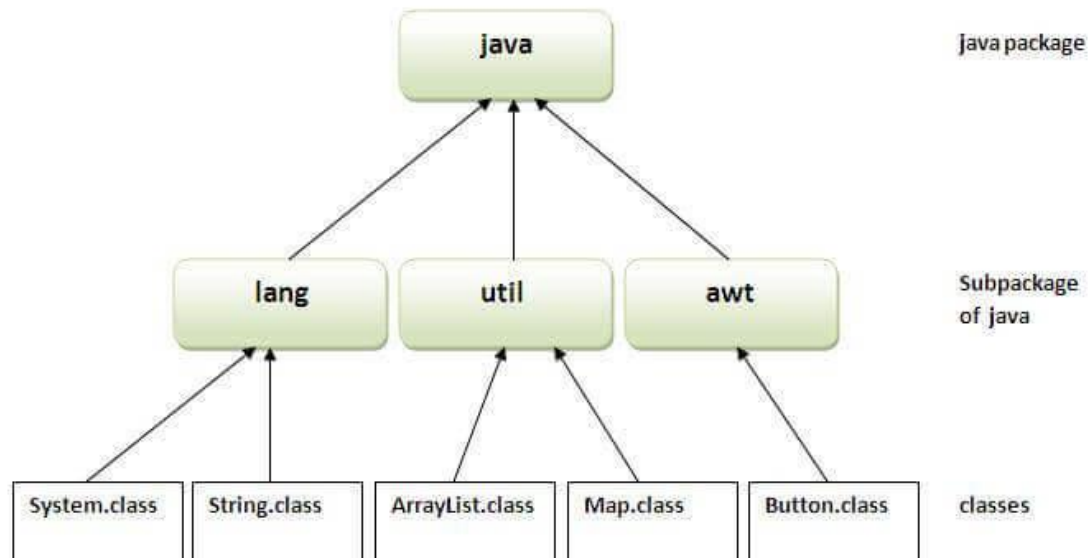
Example: Java Package

The **package keyword** is used to create a package in java.

```java
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

- import package.*;
- import package.classname;
- fully qualified name

How to access package from another package?

1) Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
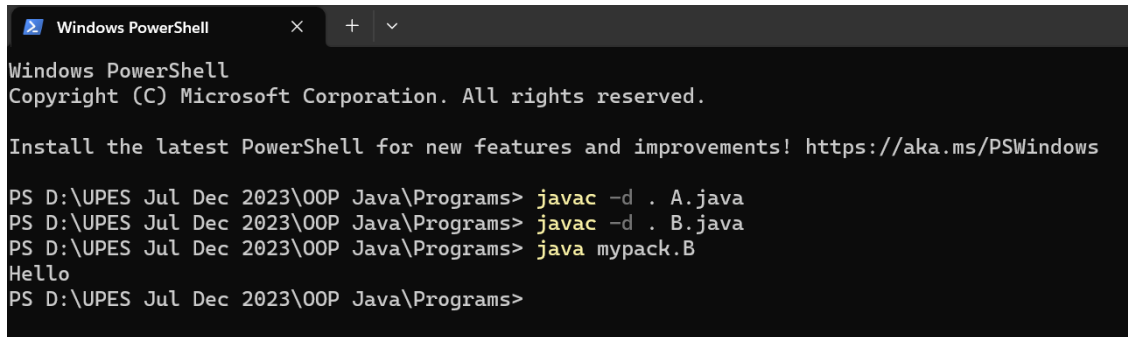
```java
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```java
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
  A obj = new A();
  obj.msg();
}
 }
```

```
Windows PowerShell        ×    +   ∨

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS D:\UPES Jul Dec 2023\OOP Java\Programs> javac -d . A.java
PS D:\UPES Jul Dec 2023\OOP Java\Programs> javac -d . B.java
PS D:\UPES Jul Dec 2023\OOP Java\Programs> java mypack.B
Hello
PS D:\UPES Jul Dec 2023\OOP Java\Programs>
```

How to access package from another package?

2) Using packagename.classname

- If you import package.classname then only declared class of this package will be accessible.

```java
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}


//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:Hello

Deepak Sharma, UPES Dehradun

How to access package from another package?

3) Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

```java
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Note: If you import a package, subpackages will not be imported.

Access Modifiers in Java

- There are two types of modifiers in Java:
  - **access modifiers**
  - **non-access modifiers**.

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

- There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

Four types of Java access modifiers:

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Understanding Java Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Private

The private access modifier is accessible only within the class.

```java
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);//Compile Time Error
  obj.msg();//Compile Time Error
  }
}
```

Role of Private Constructor

- If you make any class constructor private, you cannot create the instance of that class from outside the class.

```java
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
  A obj=new A();//Compile Time Error
 }
}
```

Note: A class cannot be private or protected except nested class.

Default

- If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

The scope of class A and its method msg() is default so it cannot be accessed from outside the package.

Deepak Sharma, UPES Dehradun

Protected

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

```java
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```java
//save by B.java
package mypack;
import pack.*;

class B extends A{
 public static void main(String args[]){
  B obj = new B();
  obj.msg();
 }
}
```

Public

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

```java
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}



//save by B.java
package mypack;
import pack.*;

class B{
 public static void main(String args[]){
  A obj = new A();
  obj.msg();
 }
}
```

//Example for access control

```
class Test {
    int a;          // default access
    public int b;   // public access
    private int c;  // private access
    /*protected applies only
    when inheritance is involved*/


    // methods to access c

    void setc(int i){
            c = i;     // set c's value

    }
    int getc() {
            return c;   // get c's value
    }
}
```

```
class AccessTest {

        public static void main(String args[]) {

        Test ob = new Test();


// These are OK, a and b may be accessed directly
            ob.a = 10;
            ob.b = 20;


// This is not OK and will cause an error
            //ob.c = 100; // Error!


// You must access c through its methods
            ob.setc(100); // OK

System.out.println(ob.a + " " +ob.b + " " + ob.getc());
            }
}
```

Scanner Class in Java

- **Scanner**: Helps in reading inputs from many sources.
  - Communicates with `System.in`
  - Can also read from files, web sites, databases, …

- The `Scanner` class is found in the `java.util` package.
  `import java.util.Scanner;`

- Constructing a `Scanner` object to read console input:

  `Scanner` **name** `= new Scanner(System.in);`

  - Example:
  `Scanner console = new Scanner(System.in);`

Scanner Class- Input types

Method	Description
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

`next()`	reads a one-word `String` from the user Avoid when Scanner connected to System.in

Scanner example 1

```java
import java.util.Scanner;

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

        System.out.print("How old are you? ");
        int age = console.nextInt();

        int years = 65 - age;
        System.out.println(years + " years until
   retirement!");
    }
}
```

age 29

years 36

29

- Console (user input underlined):

```
How old are you?
36 years until retirement!
```

Scanner example 2

- The `Scanner` can read multiple values from one line.

```java
import java.util.Scanner;
public class ScannerMultiply {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Please type two numbers: ");
        int num1 = console.nextInt();
        int num2 = console.nextInt();

        int product = num1 * num2;
        System.out.println("The product is " + product);
    }
}
```

- Output (user input underlined):

```
Please type two numbers: 8 6
The product is 48
```

Scanner Example3

```java
import java.util.Scanner;
public class ScannerDemo1
{
    public static void main(String[] args)
    {
        // Declare the object and initialize with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // String input
        String name = sc.nextLine();

        // Character input
        char gender = sc.next().charAt(0);

        int age = sc.nextInt();
        long mobileNo = sc.nextLong();
        double cgpa = sc.nextDouble();

        System.out.println("Name: "+name);
        System.out.println("Gender: "+gender);
        System.out.println("Age: "+age);
        System.out.println("Mobile Number: "+mobileNo);
        System.out.println("CGPA: "+cgpa);
    }
}
```

Concepts of Classes, Objects

General Form of Class

class *classname {*

      type instance-variable1;

      //…

      type instance-variableN;

      static type variable1;

      type methodname1(parameter-list) {

            // body of method

      }

      // …

      type methodnameN(parameter-list){

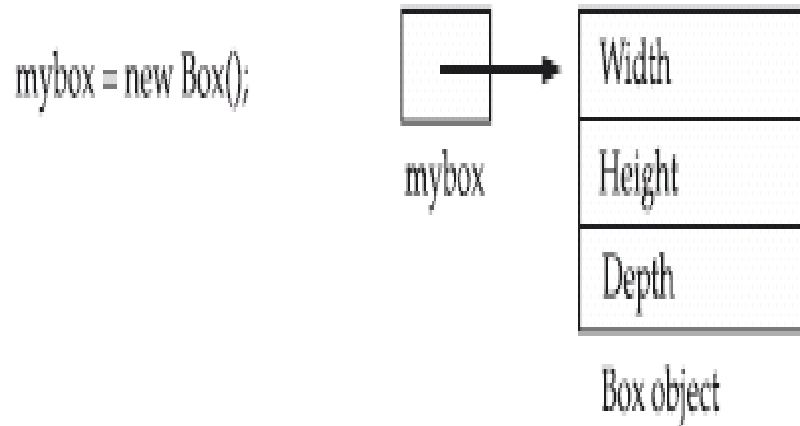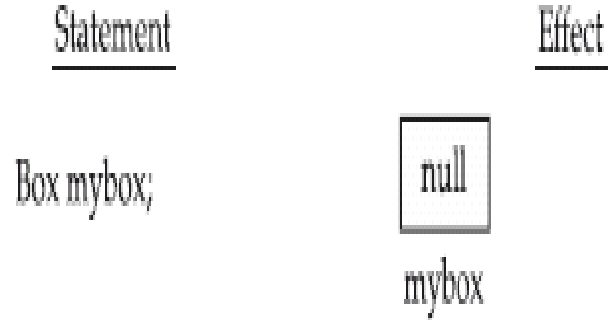   // body of method

      }

}

Declaring an Object

```
class Box{
    double width;
    double height;
    double depth;
}
```

Representation 1:
Box mybox;
mybox=new Box();

Representation 2:
Box mybox=new Box();

Statement	Effect
Box mybox;	null mybox
mybox = new Box();	mybox → [Width / Height / Depth] Box object

Representation 3: Assigning Object Reference Variables
Box b1 = new Box();
Box b2 = b1;

```java
package packagename;
import statement;
class classname{
    //instance variables;
    //class or static variables;
    /*methods(parameters){
        //local variables;
        //object creation
        //statements;
    }*/
}
```

```java
import java.util.Scanner;
class Demon{
    int i;
    static double d;
public static void main(String args[]){
        char c='a';
        Demon obj=new Demon();
Scanner s=new Scanner(System.in);
        d=s.nextDouble();
        System.out.println(obj.i);
        System.out.println(d);
        System.out.println(c);
    }
}
```

Deepak Sharma, UPES Dehradun

```java
class classname{
    //instance variables;
    //class or static variables;
    /*methods(parameters){
        //local variables;
        //statements;
    }*/
}
class  MainCLass{
public static void main(String args[])
{
        //object creation
        //invoking methods
        //statements
    }
}
```

```java
class Test{
    char c='a';
    static float f;
    void display(){
        int i=10;
        System.out.println("Test:display()");
        System.out.println("c value: "+c);
        System.out.println("i value: "+i);
        System.out.println("f value: "+f);
    }
}
class Demo{
    public static void main(String args[]){
    Test t=new Test();
     t.display();
    System.out.println("Demo:main()");
    }
}
```

Constructors and Methods

- A constructor is a special member function whose task is to **initialize** an object immediately upon creation.

- The constructor is **automatically** called immediately after the object is created.

- A constructor has the **same name as the class** in which it resides and is syntactically similar to a method.

- If no constructor in program .System provides its own constructor called as **default** constructor.

- Constructors **doesn't** have any return type.

- A constructor which accepts parameters is called as **parameterized** constructor.

- Constructors can be **overloaded**.

Default Constructor:

- A constructor that accepts no parameters is called Default constructor.
- If not defined, provided by the compiler.
- The default constructor is called whenever an object is created without specifying initial values.

Ex: class Box {

       double width;

       double height;

       double depth;

      Box() *{*

           width = 10;

           height = 10;

           depth = 10;

        }

    }

  // declare, allocate, and initialize Box objects

Box mybox1 = new Box();

// Non parameterized Default Constructor

```java
class Test{
    char c='a';
    static float f;
    Test(){
        int i=10;
        System.out.println("Test:Test()");
        System.out.println("c value: "+c);
        System.out.println("i value: "+i);
        System.out.println("f value: "+f);
    }
}
class ConDemo{
    public static void main(String args[]){
    Test t=new Test();
     //t.Test();
System.out.println("ConDemo:main()");
    }
}
```

//Parameterized Constructor

```java
class Test{
    int x,y;
    Test(int a, int b){
        x=a;
        y=b;
    System.out.println("x value: "+x);
    System.out.println("y value: "+y);
    }
}
class PConDemo{
public static void main(String args[]){
Test t=new Test(10,20);
System.out.println("PConDemo:main()");
    }
}
```

Checkpoint

```java
class Test{
        int x,y;
        Test(int a, int b){
            x=a;
            y=b;
  System.out.println("x value: "+x);
  System.out.println("y value: "+y);
        }
}
public class PConDemo{
public static void main(String args[]){
Test t=new Test(10,20);
Test t1=new Test();
System.out.println("PConDemo:main()");
        }
}
```

```
/PConDemo.java:13: error: constructor Test in class Test cannot be applied to given types;
Test t1=new Test();
        ^
  required: int,int
  found:    no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Methods

General Form:

type name(parameter-list) {

    // body of method

}

- The type of data returned by a method must be compatible with the return type specified by the method.

- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

    return *value*;   //Here, *value* is the value returned.

Ex:

double volume(double w, double h, double d) {

    return w*h*d;

}

Non parameterized or default Constructor

```
class Box{
    double width;
    double height;
    double depth;

    Box(){
        width=10;
        height=10;
        depth=10;
    }
    double volume(){
    return width*height*depth;
    }
}
```

```
class BoxDemo6{
public static void main(String args[]){

        Box mybox1=new Box();
        Box mybox2=new Box();

        double vol;

        vol=mybox1.volume();
    System.out.println("Volume is: "+vol);

        vol=mybox2.volume();
    System.out.println("Volume is: "+vol);
        }
}
```

Parameterized Constructor

```java
class Box{
    double width;
    double height;
    double depth;

Box(double w,double h,double d){
        width=w;
        height=h;
        depth=d;
}
    double volume(){
        return width*height*depth;
    }

}
```

```java
class BoxDemo7{
    public static void main(String args[]){

        Box mybox1=new Box(10,20,15);
        Box mybox2=new Box(3,6,9);

        double vol;

        vol=mybox1.volume();

System.out.println("Volume is: "+vol);

        vol=mybox2.volume();

System.out.println("Volume is: "+vol);
    }
}
```

Java Arrays

- An array is a collection of similar type of elements which has contiguous memory location.

- Since arrays are objects in Java, we can find their length using the object property *length. [e.g. array_var.length]*

- The variables in the array are ordered, and each has an index beginning from 0.

- The **size** of an array must be specified by int or short value and not long.

- The direct superclass of an array type is Object.

- This storage of arrays helps us in randomly accessing the elements of an array [Support Random Access].

- The size of the array cannot be altered(once initialized). However, an array reference can be made to point to another array.

Java Arrays
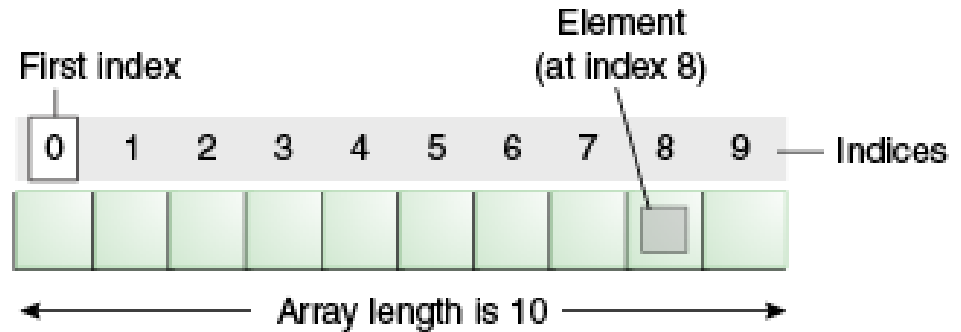
Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.

- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array and Multidimensional Array

Single Dimensional Array in Java

  Syntax to Declare an Array in Java

    dataType[] arr; (or)

    dataType []arr; (or)

    dataType arr[];

  Instantiation of an Array in Java

    arr=**new** datatype[size];

int a[]={33,3,4,5};//declaration, instantiation and initialization

Java Array Example

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

```java
import java.util.Scanner;
class ArrayEx{
    public static void main(String args[]){
        Scanner input=new Scanner(System.in);
        int a[]={10,20,30,40,50};
        char []c={'a','b','c','d','e'};
        int b[]=new int[5];
        for(int i=0;i<5;i++){
            System.out.print(a[i]+" ");
            System.out.println(c[i]+" ");
        }
        for(int i=0;i<5;i++){
            b[i]=input.nextInt();
        }
        for(int i=0;i<5;i++){
            System.out.print(b[i]+" ");
        }
    }
}
```

For-each Loop for Java Array

class Testarray1{

public static void main(String args[]){

int arr[]={33,3,4,5};

//printing array using for-each loop

for(**int** i:arr)

System.out.println(i);

}}

Passing Array to a Method in Java

```java
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

Returning Array from the Method

```java
class TestReturnArray{
//creating method which returns an array
static int[] get(){
return new int[]{10,30,50,90,60};
}

public static void main(String args[]){
//calling method which returns an array
int arr[]=get();
//printing the values of an array
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}}
```

Multidimensional Array in Java

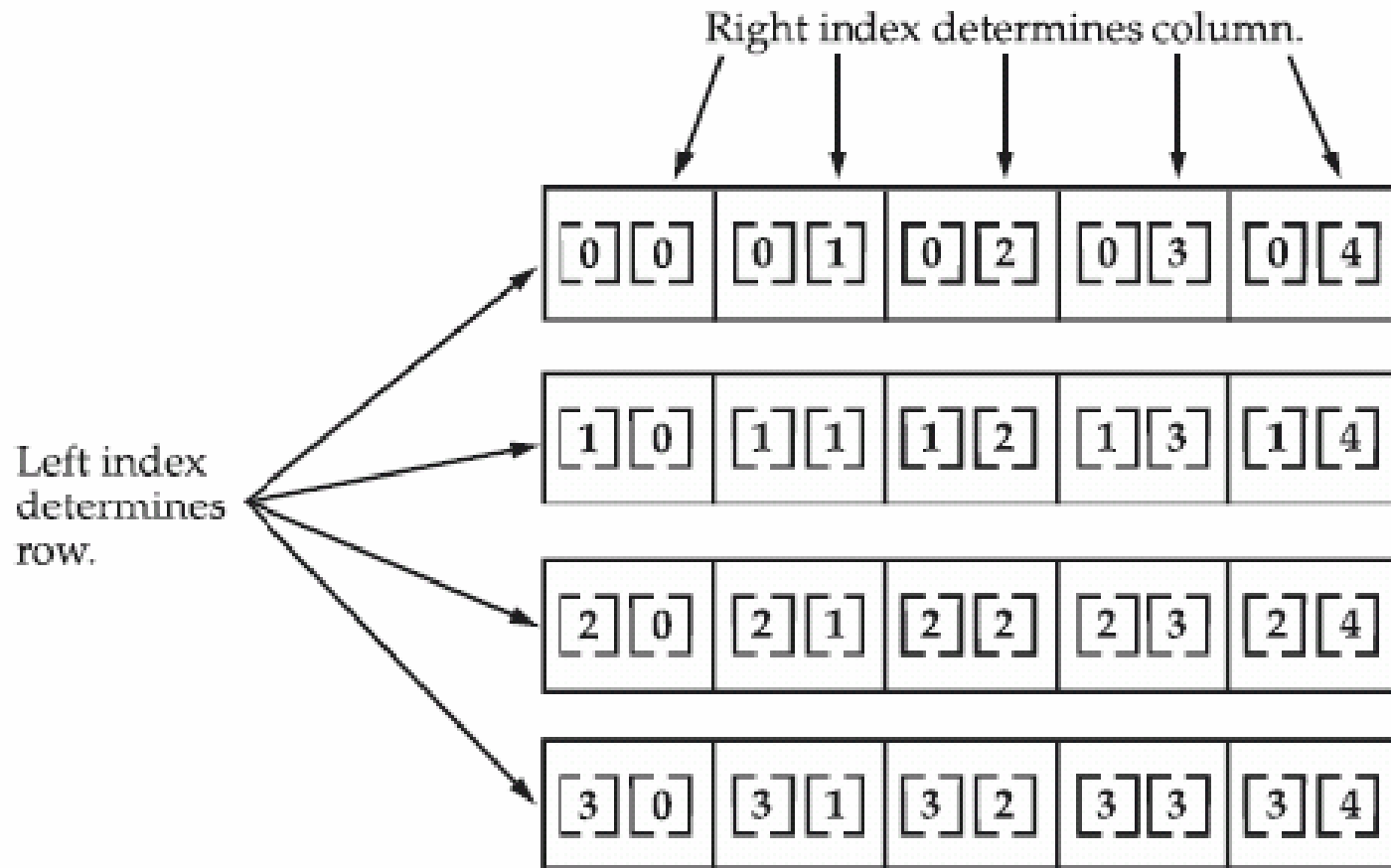**Syntax to Declare Multidimensional Array in Java**

dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

int[][] arr=new int[3][3];//3 row and 3 column

Multidimensional Arrays

int twoD[][]=new int[4][5];

Right index determines column.

| 0 0 | 0 1 | 0 2 | 0 3 | 0 4 |

Left index determines row.

| 1 0 | 1 1 | 1 2 | 1 3 | 1 4 |

| 2 0 | 2 1 | 2 2 | 2 3 | 2 4 |

| 3 0 | 3 1 | 3 2 | 3 3 | 3 4 |

Example of Multidimensional Java Array

```java
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

How to Create Array of Objects in Java

- An array of objects stores OBJECTS.

- The array elements store the location of the reference variables of the object.

Syntax:

Class obj[]= new Class[array_length]

Example:

```java
class ObjectArray{
  public static void main(String args[]){
   Account obj[] = new Account[2] ;
   obj[0] = new Account();
   obj[1] = new Account();
   obj[0].setData(1,2);
   obj[1].setData(3,4);
   System.out.println("For Array Element 0");
   obj[0].showData();
   System.out.println("For Array Element 1");
    obj[1].showData();
  }
}
class Account{
  int a;
  int b;
 public void setData(int c,int d){
   a=c;
   b=d;
 }
 public void showData(){
   System.out.println("Value of a ="+a);
   System.out.println("Value of b ="+b);
 }
}
```

For Array Element 0
Value of a =1
Value of b =2

For Array Element 1
Value of a =3
Value of b =4

Method & Constructor Overloading

➢ Defining two or more methods within the same class that share the same name is called **method overloading**.

➢ Java uses the type and/or number of arguments to determine which version of the overloaded method to call.

➢ Constructors can also be **overloaded** in the same way as method overloading.

//method overloading

```java
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    void test(int a) {
        System.out.println("a: " + a);
    }
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

//Constructor Overloading

```java
class CDemo{
    int  value1;
    int  value2;
    CDemo(){
            value1 = 10;
            value2 = 20;
    System.out.println("Inside 1st
                    Constructor");
    }
    CDemo(int a){
            value1 = a;
            System.out.println("Inside 2nd Constructor");
    }
    CDemo(int a,int b){
            value1 = a;
            value2 = b;
            System.out.println("Inside 3rd Constructor");
    }
    public void display(){
            System.out.println("Value1 === "+value1);
            System.out.println("Value2 === "+value2);
    }
}
```

```java
public static void main(String args[]){
                CDemo d1 = new CDemo();
                CDemo d2 = new CDemo(30);
                CDemo d3 = new CDemo(30,40);
                d1.display();
                d2.display();
                d3.display();
        }
}
```

this

- In java, it is illegal to declare two local variables with the same name inside the same or enclosing scopes.

- But you can have formal parameters to methods, which overlap with the names of the class' instance variables.

- **this** keyword is used to refer to the **current** object.

- **this** can be used to resolve any name collisions that might occur between instance variables and formal variables.

- When a formal variable has the same name as an instance variable, the formal variable **hides** the instance variable.

- Also used in **method chaining** and **constructor chaining**.

this

Following are various uses of 'this' keyword in Java:

- It can be used to refer instance variable of current class
- It can be used to invoke or initiate current class constructor
- It can be passed as an argument in the method call
- It can be passed as argument in the constructor call
- It can be used to return the current class instance

```java
// instance and formal variables are different
class Box{
    double w=5,h=5,d=5;
    Box(double w1,double h1,double d1){
        w=w1;
        h=h1;
        d=d1;
    }
    double volume(){
        return w*h*d;
    }
}
class BoxTest1{
    public static void main(String args[]){
        Box b=new Box(1,2,3);
        System.out.println("Volume is: "+b.volume());
    }
}
```

Output:
Volume is:6.0

```java
// instance and formal variables are same
class Box{
    double w=5,h=5,d=5;
    Box(double w,double h,double d){
        w=w;
        h=h;
        d=d;
    }
    double volume(){
        return w*h*d;
    }
}
class BoxTest2{
    public static void main(String args[]){
        Box b=new Box(1,2,3);
        System.out.println("Volume is: "+b.volume());
    }
}
```

Output:
Volume is:125.0

```java
//'this' to refer instance variable
class Box{
    double w=5,h=5,d=5;
    Box(double w,double h,double d){
            this.w=w;
            this.h=h;
            this.d=d;
}
    double volume(){
            return w*h*d;
    }
}
class BoxTest2{
    public static void main(String args[]){
            Box b=new Box(1,2,3);
            System.out.println("Volume is: "+b.volume());
    }
}
```

Output:
Volume is:6.0

this: to invoke current class method

- You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example

```java
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

```java
class Fchain{           // method chaining
        int a,b;
        Fchain setValue(int x,int y){
                a=x;
                b=y;
                return this;
        }
        Fchain disp(){
                System.out.println("a value is:"+a);
                System.out.println("b value is:"+b);
                return this;
        }
}
class FchainDemo{
        public static void main(String args[]){
        Fchain f1=new Fchain();
        f1.setValue(10,20).disp().setValue(11,22).disp();
        }
}
```

O/P:
a value is:10
b value is:20
a value is:11
b value is:22

this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for _constructor chaining_.

Calling default constructor from parameterized constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Rule: Call to this() must be the first statement in constructor.

this() : to invoke current class constructor

Calling parameterized constructor from default constructor:

```java
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

```java
//Constructor Chaining
class Test{
        int a,b,c,d;
        Test(int x,int y){
                a=x;
                b=y;
        }
        Test(int x,int y,int z){
                this(x,y);
                c=z;
        }
        Test(int p,int q,int r,int s){
                this(p,q,r);
                d=s;
        }
        void disp(){
   System.out.println(a+" "+b+" "+c+" "+d);
        }
}
```

```java
class TestDemo{
        public static void main(String args[]){
                Test t1=new Test(10,20,30,40);
                t1.disp();
                }
}
```

O/P:10 20 30 40

Parameter Passing

➢ The **call-by-value** copies the *value* of a actual parameter into the formal parameter of the method.

➢ In this method, changes made to the formal parameter of the method have no effect on the actual parameter.

➢ In **call-by-reference**, a reference to an actual parameter (not the value of the argument) is passed to the formal parameter.

➢ In this method, changes made to the actual parameter will affect the actual parameter used to call the method.

```java
// Simple types are passed by value.
class Test {
    void meth(int i, int j) {
            i *= 2;
            j /= 2;
    }
}
class CallByValue {
    public static void main(String args[]) {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a and b before call: " +a + " " + b);
    ob.meth(a, b);
    System.out.println("a and b after call: " +a + " " + b);
    }
}
```

```java
// Objects are passed by reference.
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    void meth(Test o) {                        // pass an object
        o.a *= 2;
        o.b /= 2;
    }
}
class CallByRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
    }
}
```

```java
class Box{    //call by reference using Objects
    double width,height,depth;
    Box(Box ob) {
            width = ob.width;
            height = ob.height;
            depth = ob.depth;
    }
    Box(double w, double h, double d){
            width = w;
            height = h;
            depth = d;
    }
    Box(){
            width = -1;
            height = -1;
            depth = -1;
     }
    Box(double len) {
            width = height = depth = len;
    }
    double volume() {
            return width * height * depth;
    }
}
```

```java
class OverloadCons2 {
        public static void main(String args[]) {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        Box myclone = new Box(mybox1);
        double vol;

        vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

        vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

        vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

        vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
        }
}
```

Recursion

➢ Recursion is the process of defining something in terms of itself.

➢ A method that calls itself is said to be *recursive*.

```java
class Factorial{
    int fact(int n){
        int result;
        if(n==1)
            return 1;
        else
            result = fact(n-1) * n;
            return result;
    }
}
class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects
  - for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

- It makes your program **memory efficient** (i.e., it saves memory).

Java static Keyword

- Mainly used for memory management.
- Can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.

The static can be:

- Variable (also known as a class variable)
- Method (also known as a class method)
- Block
- Nested class

Java static variable

- Without static variable

class Student{

    int rollno;

    String name;

    String college="UPES";

}

- With static variable

class Student{

    int rollno;

    String name;

    static String college="UPES";

}

Assume 120 students in a class.
Which version of the program will you prefer?

Java static property is shared to all objects.

Java static variable

```java
class Student{
  int rollno;String name; //instance variable
  static String college ="UPES";//static variable
  Student(int r,String n){
  rollno = r; name = n;
  }
void display ()
{System.out.println(rollno+" "+name+" "+college);}
}
public class TestStaticVariable{
 public static void main(String args[]){
 Student s1 = new Student(1,"Raj");
 Student s2 = new Student(2,"Rohan");
 s1.display();
 s2.display();
 Student.college="University of petroleum and Energy Studies";
 s1.display();
 s2.display();
 }
}
```

Output:
1 Raj UPES
2 Rohan UPES
1 Raj University of petroleum and Energy Studies
2 Rohan University of petroleum and Energy Studies

Java static variable

- **Count number of instances using static variable**

```java
class Counter2{
        static int count=0;//will get memory only once and retain its value
        Counter2(){
                count++;//incrementing the value of static variable
                System.out.print(count+" ");
        }
        public static void main(String args[]){
        Counter2 c1=new Counter2();
        Counter2 c2=new Counter2();
        Counter2 c3=new Counter2();
        }
}
```

Output: 1 2 3

Point to remember: Static Variable

- The static variables can be accessed in all types of methods: static or non-static.

- We cannot access non-static variables in static methods because non-static variables can only be accessed by creating an instance of the class.

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.

- A static method can be invoked without the need for creating an instance of a class.

- A static method can access static data member and can change the value of it.

Java static method

Example:

```java
class Calculate{
  static int cube(int x){
  return x*x*x;
  }


  public static void main(String args[]){
  int result=Calculate.cube(5);  //or cube(5)
  System.out.println(result);
  }
}
```

Java static method

```java
class Student{
    int rollno;String name;
    static String college = "UPES";
    static void change(){
    college = "UPES Dehradun";
    }
    Student(int r, String n){
    rollno = r; name = n;
    }
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
public class TestStaticMethod{
    public static void main(String args[]){
    Student.change();//calling change method
    Student s1 = new Student(1,"Karan");
    Student s2 = new Student(2,"Aryan");
    //calling display method
    s1.display();
    s2.display();
    }
}
```

Output:
1 Karan UPES Dehradun
2 Aryan UPES Dehradun

Restrictions for the static method

There are two main restrictions for the static method.

- The static method can not use non static data member or call non-static method directly.

- this and super cannot be used in static context.

```java
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Output: Compile Time Error

```java
class A{
 static int a=40;//static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Output: 40

Why is the Java main method static?

Java **main()** method is always static, so that compiler can call it without the creation of an object or before the creation of an object of the class.

- In any Java program, the **main()** method is the starting point from where compiler starts program execution. So, the compiler needs to call the main() method.

- If the **main()** is allowed to be non-static, then while calling the **main()** method JVM has to instantiate its class.

Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

```
class A2{
  A2(){System.out.println("Constructor Called");}

static{
        System.out.println("static block is invoked");
    }

  public static void main(String args[]){
   System.out.println("Hello main");
   A2 a=new A2();
  }
}
```

Output:
static block is invoked
Hello main
Constructor Called

Garbage Collection

- Garbage collection done automatically in java.

- When no reference to an object exist, that object is assumed to be no loner needed, and the memory occupied by the object can be reclaimed.

- Garbage collection only occurs at regular intervals during the execution of your program.

- We can run garbage collection on demand by calling the **gc()** method.

- **public static void gc():** Initiates the garbage collection.

System.gc();

```java
public class GarbageCollector{
    public static void main(String[] args) {
        int SIZE = 200;
        StringBuffer s;
        for (int i = 0; i < SIZE; i++) {
        }
        System.out.println("Garbage Collection started explicitly.");
        System.gc();
    }
}
```

finalize( ) method

- Sometimes an object will need to perform some action when it is destroyed.

  Ex:

  If an object is holding some non-java resource such as a file, then you might want to make sure these resources are freed before an object is destroyed.

- To handle such situations, Java provides a mechanism called *finalization*.

- **The finalize( ) method has this general form:**

  ```
  protected void finalize( ){
              // finalization code here
  }
  ```

Practice Question

To be decided in the class

Complete all topics of Unit 1 before starting Unit 2