



Class Note of OBJECT ORIENTED PROGRAMMING USING JAVA

Course Code: 18ES1T06



Prepared
by

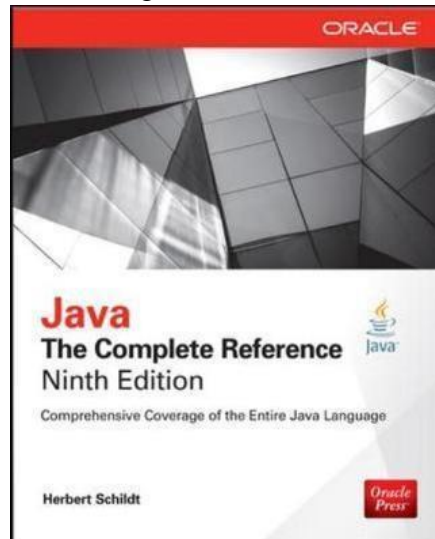
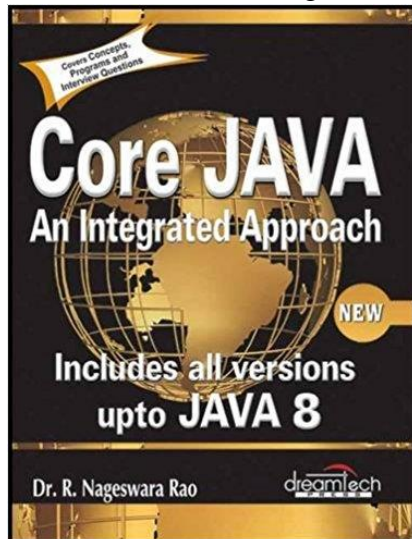
Jasaswi Prasad Mohanty
Department of Computer Science & Engineering
Silicon Institute of Technology
Bhubaneswar

Syllabus:

Module I	Object oriented concepts: Object oriented systems development life cycle, Unified Modeling Language, UML class diagram, Use-case diagram; Java Overview: Java Virtual Machine, Java buzz words, Data types, Operators, Control statements, Class fundamentals, Objects, Methods, Constructors, Overloading, Access modifiers.
Module II	Inheritance: Basics of Inheritance, using super and final keyword, method overriding, Abstract classes, defining and importing packages, access protection, interfaces; Exception handling: Exception fundamentals, types, understanding different keywords (try, catch, finally, throw, throws), User defined exception handling.
Module III	Input/Output: Files, stream classes, reading console input; Threads: thread model, use of Thread class and Runnable interface, thread synchronization, multithreading, inter thread communication.
Module IV	String manipulation: Basics of String handling, String class, StringBuilder, StringBuffer, StringTokenizer. Applet basics and life cycle; Event Handling: delegation event model, event classes, sources, listeners, Adapter class.
Module V	Introduction to GUI Programming: working with windows, frames, graphics, color, and font. AWT Control fundamentals. Swing overview; JavaFX overview; Java database connectivity: JDBC overview, creating and executing queries, dynamic queries.

Books:

1. JAVA Complete Reference (9th Edition), Herbalt Schelidt.
2. Core JAVA – An Integrated Approach, Dr. R. Nageswara Rao

**Web Resources:**

1. For Java Tutorial: <https://www.javatpoint.com/history-of-java>
2. For Java API: <http://docs.oracle.com/javase/7/docs/api/>.

Scope & Objective:

The objective of this course is to introduce the key concepts of object-oriented programming (OOP) using Java as the programming language.

Pre-Requirement:

- Knowledge of any one programming language - preferably C
- Knowledge of Data Structure.
- Some familiarity with Object Oriented Programming is desired but not required.

Outcome:

Upon completion of this course, students will be able to:

1. Apply object oriented principles in software design process to develop Java programs for real life applications.
2. Employ inheritance and exception handling techniques for developing robust and reusable software.
3. Develop programs using stream classes for various I/O operations and design concurrent programs using threads to maximize the use of processing power.
4. Design applications for text processing using String class and develop user interactive applications using event handling.
5. Design database driven GUI applications using AWT, Swing and JDBC.

Detail Hand-out

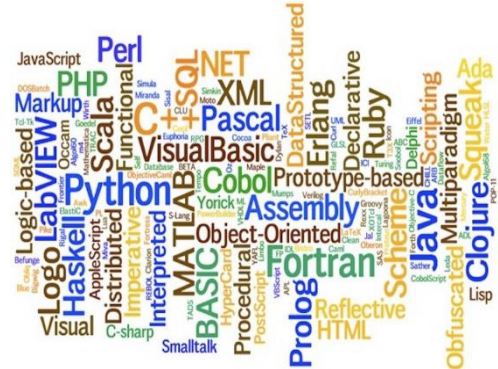
1.	Introduction to Object Oriented Programming (OOP), Features of OOP, Advantages & Disadvantages of OOP	T1/Chapter 2
2.	Introduction to Java (What is Java, Where Java is used, Brief History of Java), Features of Java (Java Buzzwords)	T1/Chapter 1
3.	Java's Magic - The Bytecode, Introduction to Java Virtual Machine, Simple Java Program, Java compilation process, Java libraries, Setting class path	T1/Chapter 1
4.	Data Types: Primitive and non-Primitive data types, Operators, Type Conversion	T1/Chapter 3, 4
5.	Arrays (1D Array, Multi-dimensional Array): Initialization and processing	T1/Chapter 3
6.	Introduction to Class & Objects: Form of a Java Class, Declaring Object, Adding Methods to class, Array of Objects	T1/Chapter 6
7.	Constructor, Types of Constructor (Default & Parameterized), Difference between Constructor & Method	T1/Chapter 6
8.	Method Overloading, Constructor Overloading, Argument Passing, Returning Objects	T1/Chapter 7
9.	Static keyword (static instance variable, static method, static block, static class), this keyword	T1/Chapter 6, 7
10.	Access Specifiers: public, private, protected, default	T1/Chapter 7
11.	Inheritance: Understanding inheritance with its advantage and disadvantage, Types of Inheritance	T1/Chapter 8
12.	Using super and final keyword	T1/Chapter 8
13.	Polymorphism, Compile time vs run time polymorphism, Method overriding, Overloading vs overriding, dynamic method dispatch	T1/Chapter 8
14.	Wrapper Class, Autoboxing, Unboxing, Abstract Class, Abstract method	T1/Chapter 12

15.	Interface, Interface vs Abstract class	T1/Chapter 9
16.	Packages: Definition, finding packages and CLASSPATH, importing packages	T1/Chapter 9
17.	Exception Handling: Exception fundamentals, types, using try & catch, multiple catch clauses, nested try, throw, throws, finally	T1/Chapter 10
18.	User defined exception	T1/Chapter 10
19.	Nested class and Inner class, static nested class, non-static nested class, Anonymous Inner Class	T1/Chapter 7
20.	Multithreading, Thread vs Process, Thread Life Cycle (States of Thread), The main thread	T1/Chapter 11
21.	Creating a thread by implementing Runnable interface	T1/Chapter 11
22.	Creating a thread by extending the Thread class	T1/Chapter 11
23.	Creating multiple threads, Using isAlive() and join(), Thread Priorities Thread Synchronization	T1/Chapter 11
24.	Input/Output: File, Byte Stream classes (FileOutputStream, FileInputStream, BufferedOutputStream, BufferedInputStream, RandomAccessFile), Reading/Writing in a file	T1/Chapter 13
25.	Character Stream classes (FileReader, FileWriter, BufferedReader, BufferedWriter)	T1/Chapter 13
26.	Reading Console Input, Writing Console Output	T1/Chapter 13
27.	String: Creating & initializing a string, Finding length of string, string concatenation, String conversion and toString(), Character extraction, String comparison, Searching a string, Modifying strings, trimming a string	T1/Chapter 16
28.	StringBuffer, StringBuilder, StringTokenizer class. Difference between String, StringBuffer & StringBuilder class	T1/Chapter 16
29.	Applet: Introduction, Life cycle of an Applet, Creating an Applet, Applet Class	T1/Chapter 23
30.	Applet display methods, Requesting Repainting, Passing Parameters to Applets	T1/Chapter 23
31.	Working with Graphics, Graphics Class' Drawing Methods, Colors & Font class	T1/Chapter 25
32.	Event Handling: Introduction	T1/Chapter 24
33.	Handling Mouse Events	T1/Chapter 24
34.	Handling Keyboard Events	T1/Chapter 24
35.	Abstract Window Toolkit (AWT), AWT Hierarchy, Container & Components: Window, Frame, Panel	T1/Chapter 25
36.	Creating Window Programming, Creating a Frame	T1/Chapter 25
37.	Java LayoutManagers (BorderLayout, FlowLayout, GridLayout, CardLayout), Label, Push Button, Check Box, Text Field, Radio Button, TextArea, Choice, List, ScrollBar	T1/Chapter 26
38.	Swing, Difference between AWT & Swing, MVC Model, Components and Container	T1/Chapter 31
39.	Creating Frame in Swing, JLabel, JButton, JTextField, JTextArea Class	T1/Chapter 32
40.	JDBC: Types of Driver, Driver Class, Connection Class, ResultSet	T1/Chapter
41.	Connecting MySQL database	T1/Chapter
42.	Collection classes: ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap Legacy Classes: Vector, Stack,	T1/Chapter 18/19

MODULE I

1. PROGRAMMING LANGUGES

- A programming language is a notation designed to connect instructions to a machine or a computer. Programming languages are mainly used to control the performance of a machine or to express algorithms.
- A programming language is a vocabulary and set of grammatical rules for instructing a computer or computing device to perform specific tasks. The term programming language usually refers to high-level languages, such as BASIC, C, C++, COBOL, Java, FORTRAN, Ada, and Pascal.
- Each programming language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.



1.1. Types of Programming Languages

Different kinds of languages have been developed to perform different types of work on the computer. Basically, languages can be divided into different categories according to **how the computer understands** them. Programming languages are mainly classified into the following types:

- **Procedural Programming Language:** A procedural language is a computer programming language that **specifies a series of well-structured steps and procedures** within its programming context procedures with its programming context to compose a program. It is known as imperative language. **A program written in a procedural language contains one or more procedures.** It determines how and what a process should be done. To execute the program is procedural language stepwise procedure must be followed. **The fundamental building blocks of procedural programming are instructions.** Examples of procedural programming languages include C, FORTRAN, ALGOL, COBOL and BASIC.
- **Functional Programming Language:** In functional programs, data passes through your code like water through a pipe (from function to function to function) to get a result. Also, functions are treated as data to be worked on just like anything else in our program. We can take functions as a parameter, return them, build custom functions, and build functions from other functions. **The fundamental building blocks of functional programming are functions.** Examples of functional programming languages include Lisp, Scheme, Clojure, Wolfram Language (also known as Mathematica), Racket, Erlang, OCaml, Haskell, and F#.
- **Object Oriented Programming Language:** Object-oriented programming, as it's practiced by most developers, is pretty much like procedural programming, but it encapsulates data and behavior into objects. Object-oriented programmers can also define hierarchies of types and subtypes. This is meant to help with code reuse. **The fundamental building blocks of object-oriented programming is object.** Examples of object-oriented programming languages include Java, C++, C#, Python, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, and Smalltalk.

Generation of Programming Languages:

- a) **Machine Level Language / First Generation Language:** A machine language consists of the numeric codes for the operations that a particular computer can execute directly. **The codes are strings of 0s and 1s, or binary digits ("bits").** Machine level language instructions typically use some bits to represent operations, such as addition, and some to represent operands, or perhaps the location of the next instruction. A computer can understand only

this language. This language is difficult to read and write, since it does not resemble conventional mathematical notation or human language, and its codes vary from computer to computer. Each different type of CPU has its own unique machine level language.

- **Advantages:** Directly interact with computer system, takes less time to execute the program.
 - **Disadvantages:** Hard to understand and debug program, Time consuming to develop new programs, Not portable.
- b) **Assembly Level Language / Second Generation Language:** Assembly language is one level above machine language. It uses short mnemonic codes for instructions and allows the programmer to introduce names for blocks of memory that hold data. One might thus write “ADD PAY, TOTAL” instead of “0110101100101000” for an instruction that adds two numbers. Assembly language is designed to be easily translated into machine language. Although blocks of data may be referred to by name instead of by their machine addresses, assembly language does not provide more sophisticated means of organizing complex information. Like machine language, assembly language requires detailed knowledge of internal computer architecture. We need an assembler (a system software) to translate assembly level language into machine-level language.
- **Advantages:** Easily understandable since it uses statements instead of binary digits, Debugging and troubleshoot is easy. Developing a program takes less time. Portable.
 - **Disadvantages:** It's a machine dependent language due to that program design for one machine no use of other machine, sometime it's hard to understand the statement or command use.
- c) **High Level Language / Third Generation Language:** It is the upper level language because the language comes under this category is closer to human languages. Hence, this is highly understood programming language by human. We need a compiler and/or interpreter system software) to translate high level language into machine level language. There are many examples of high level languages such as FORTRAN, C, C++, JAVA, COBOL, LISP, Prolog etc.
- **Advantages:** Instructions are easier to remember, Debugging is easier, Program development is less time taking, Portable
 - **Disadvantages:** Takes more space compared to other generation languages, Execute slowly.

There are some other types of programming languages also. Some of them are listed below:

- **System languages:** that are designed for writing low-level tasks, like memory and process management, data transfer etc.
- **Scripting languages:** Scripting languages are used for wiring together systems and applications at a very high level. They are almost always extremely expressive (they do a lot with very little code) and usually dynamic (meaning the compiler does very little, while the run-time system does almost everything). These are generally extremely high-level and powerful.
- **Domain-specific languages:** that are used in highly special-purpose areas only

2. COMPILER AND INTERPRETER:

- There are two ways to run programs written in a high-level language. The most common is to compile the program; the other method is to pass the program through an interpreter. Hence, a compiler or an interpreter is a program that converts program written in high-level language into machine code understood by the computer.
- Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which

machine instructions are generated. This process can be time-consuming if the program is long. The interpreter, on the other hand, can immediately execute high-level programs.

- For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly. In addition, interpreters are often used in education because they allow students to program interactively.
- The basic difference is that a compiler system, including a (built in or separate) linker, generates a stand-alone machine code program, while an interpreter system instead performs the actions described by the high level program.
- Once a program is compiled, its source code is not useful for running the code. For interpreted programs, the source code is needed to run the program every time.
- Java programs are first compiled to an intermediate form (.class file), then interpreted by the interpreter.
- Major differences between Interpreter and Compiler are listed below:

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyse the source code but the overall execution time is slower.	It takes large amount of time to analyse the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby uses interpreters.	Programming language like C, C++ uses compilers.

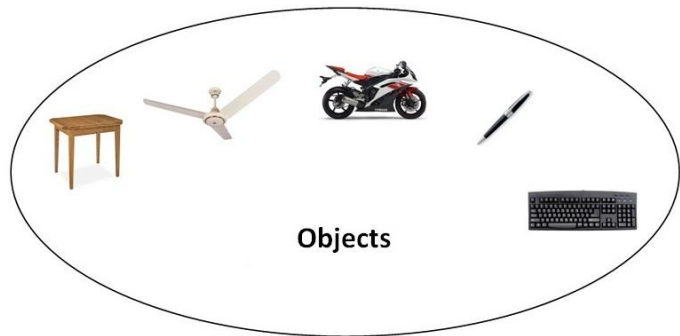
3. OBJECT-ORIENTED PROGRAMMING:

- Object-oriented programming (OOP) refers to a type of computer programming (software design) in which **programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.**
- In this way, the **data structure becomes an object that includes both data and functions.** In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.
- One of the principal **advantages** of object-oriented programming techniques over procedural programming techniques is that they **enable programmers to create modules that do not need to be changed when a new type of object is added.** A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.
- The programming challenge was seen as how to write the logic, not how to define the data. **Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.**
- The first step in OOP is to **identify all the objects** the programmer wants to manipulate and **how they relate to each other**, an exercise often known as *data modeling*. Once an object has been identified, it is generalized as a class of objects which defines the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. Objects communicate with well-defined interfaces called messages.
- **Simula** is considered as the first object-oriented programming language.

- The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

3.1. Features of object-oriented programming:

- **Object:** Object means a real word entity such as pen, chair, table etc. Any entity that has state and behavior is known as an object. Example: chair, pen, table, keyboard, bike etc. It can be physical and logical.



- **Class:** Collection of similar objects is called class. It is a logical entity. These contain data and functions bundled together under a unit. So, object is an instance of a class.

- **Abstraction:** Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing. In JAVA, we use abstract class and interface to achieve abstraction.



- **Encapsulation:** Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines. A java class is the example of encapsulation.

- **Polymorphism:** Poly and morphism refer many and form respectively. So Polymorphism as the name suggests is a certain item appearing in different forms or ways. When one task is performed by different ways i.e. known as polymorphism. An example can be to speak something e.g. cat speaks meow, dog barks woof etc. That is making a function to act in different forms depending on the place they are present is called Polymorphism. In java, we use method overloading and method overriding to achieve



polymorphism. This is a very important feature of object oriented programming methodology which has extended the handling of data type and operations.

- **Inheritance:** Inheritance is the concept of inheriting or deriving properties of an existing class to get new class or classes. In other words we may have common features or characteristics that may be needed by number of classes. So those features can be placed in a common tree class called base class and the other classes which have these characteristics can take the tree class and define only the new things that they have on their own in their classes. These classes are called derived class. The main advantage of using this concept of inheritance in Object oriented programming is it helps in reducing the code size since the common characteristic is placed separately called as base class and it is just referred in the derived class. This provide the users the important usage of terminology called as reusability.

3.2. Advantages of object-oriented programming:

- OOPs provide ability to **simulate real-world event** much more effectively.
- **Improved software-development productivity:** Object-oriented programming is **modular**, as it provides separation of duties in object-based program development. It is also **extensible**, as objects can be extended to include new attributes and behaviors. Objects can also be **reused** within an across applications. Because of these three factors – modularity, extensibility, and reusability – object-oriented programming provides improved software-development productivity over traditional procedure-based programming techniques.
- **Improved software maintainability:** For the reasons mentioned above, object oriented software is also **easier to maintain**. Since the design is modular, **part of the system can be updated in case of issues without a need to make large-scale changes**.
- **Faster development:** **Reuse enables faster development**. Object-oriented programming languages come with **rich libraries** of objects, and code developed during projects is also reusable in future projects.
- **Lower cost of development:** The **reuse** of software also lowers **the cost of development**. Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.
- **Higher-quality software:** Faster development of software and lower cost of development **allows more time and resources to be used in the verification of the software**. Although quality is dependent upon the experience of the teams, object oriented programming tends to result in higher-quality software.
- **Data Protection or security** of data is achieved by concept of **data hiding**.

3.3. Disadvantages of object-oriented programming:

- **Steep learning curve:** The **thought process involved** in object-oriented programming may not be natural for some people, and it **can take time to get used to it**. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as **inheritance** and **polymorphism**, **can be challenging** to comprehend initially.
- **Larger program size:** Object-oriented programs typically **involve more lines of code** than procedural programs.
- **Slower programs:** Object-oriented programs are typically slower than procedure based programs, as they typically **require more instructions to be executed**.
- **Not suitable for all types of problems:** There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

4. INTRODUCTION TO JAVA:

4.1. What is JAVA:

1. Java is a **programming language** and a **platform**. (**Platform:** Any hardware or software environment in which a program runs is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.)
2. Java is a **high level**, **robust**, **secured** and **object-oriented** programming language.

4.2. Where JAVA is used:

3. **Desktop Applications** such as acrobat reader, media player, antivirus etc.
4. **Web Applications** such as irctc.co.in, javatpoint.com etc.
5. **Enterprise Applications** such as banking applications.
6. Mobile
7. Embedded System
8. Smart Card

9. Robotics
10. Games etc.

4.3. Types of JAVA Applications:

1. **Standalone Application:** Standalone applications are also known as desktop applications or window-based applications. These are traditional **software that we need to install on every machine**. Example of standalone applications are: Media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.
2. **Web Applications:** An application that **runs on the server side and creates dynamic page**, is called web application. Currently, servlet, jsp, struts, spring, hibernate, jsf etc. technologies are used for creating web applications in java.
3. **Enterprise Application:** An application that is **distributed in nature**, such as banking applications etc. is called enterprise application. It has the advantage of **high level security, load balancing and clustering**. In java, EJB is used for creating enterprise applications.
4. **Mobile Application:** An application that is **created for mobile devices**. Currently Android and Java ME are used for creating mobile applications.

4.4. JAVA Platforms / Editions:

1. **Java SE (Java Standard Edition):** It is a java programming platform. It **includes Java programming APIs** such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, and Collection etc.
2. **Java EE (Java Enterprise Edition):** It is an enterprise platform which is mainly **used to develop web and enterprise applications**. It is built on the top of Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA etc.
3. **Java ME (Java Micro Edition):** It is a **micro platform which is mainly used to develop mobile applications**.
4. **JavaFx:** It is **used to develop rich internet applications**. It uses light-weight user interface API.

4.5. Why JAVA:

Java is the best programming language in terms of opportunities, development and community support because of the following reasons:

1. **Java is Easy to learn:** Java has fluent **English like syntax** with minimum magic characters e.g. Generics angle brackets, which makes it easy to read Java program and learn quickly. Once a programmer is familiar with initial hurdles with installing JDK and setting up PATH and understands How Classpath works, it's pretty easy to write a program in Java.
2. **Java is an Object Oriented Programming Language:** **Developing OOP application is much easier, and it also helps to keep system modular, flexible and extensible**. Once you have knowledge of key OOP concepts like Abstraction, Encapsulation, Polymorphism, and Inheritance, you can use all those with Java.
3. **Java has rich API:** An **API is a .html file that contains description of all the features of a software, a product, or a technology**. API document is helpful for the user to understand how to use the software or technology.

Java application programming interface (API) is a list of all classes that are part of the Java development kit (JDK). It **includes all Java packages, classes, and interfaces, along with their methods, fields, and constructors**. These prewritten classes provide a tremendous amount of functionality to a programmer. A programmer should be aware of these classes and should know how to use them.

4. **Powerful development tools:** **Eclipse and Netbeans** have played a huge role to make Java one of the best programming languages. Coding in Integrated Development Environment

(IDE) not only helps in code completion but also provides powerful debugging capability, which is essential for real-world development. IDE made Java development much easier, faster and fluent. It's easy to search, re-factor and read code using IDEs.

5. **Great collection of Open Source Libraries:** Open source libraries ensure that Java should be used everywhere. **Apache, Google**, and other organization have contributed a lot of great libraries, which makes Java development easy, faster and cost-effective.
6. **Wonderful Community Support:** A strong and thriving community is the biggest strength of Java programming language and platform. No matter, how good a language is, it wouldn't survive, if there is no community to support, help and share their knowledge. Java has been very lucky, it has lots of active forums, **StackOverflow**, open source organizations and several Java user groups to help everything. There is the **community to help beginners, advanced and even expert Java programmers**.
7. **Java is FREE:** Since Java is free from the start, i.e. you **don't need to pay anything to create Java application**. This FREE thing also helped Java to become popular among individual programmers, and among large organizations. Availability of Java programmers is another big thing, which makes an organization to choose Java for their strategic development.
8. **Excellent documentation support:** **Javadoc**, is a great piece of documentation, which **tells a lot of things about Java API**. Java wouldn't be as popular, without Javadoc. Javadoc made learning easy, and provide an excellent reference while coding in Java. With the advent of Integrated Development Environment (IDE)s e.g. Eclipse, we don't even need to look Javadoc explicitly in the browser, but we can get all information in our IDE window itself.
9. **Java is Platform Independent:** In the 1990s, this was the main reason for Java's popularity. The idea of platform independence is great, and Java's tagline "**write once run anywhere (WORA)**" was enticing enough to attract lots of new development in Java.
10. **Java is everywhere:** Java is everywhere, it's on the desktop, mobile, card, almost everywhere and so is Java programmers. This huge availability of Java programmers is another reason, why **organization prefers to choose Java for new development than any other programming language**. Also, OOP is one of the best programming paradigms, and as long as it will be there Java will remain solid.

4.6. History behind JAVA:

Java was originally **designed for interactive television**, but it was too advanced technology for the digital cable television industry at the time. Java team members (also known as **Green Team**), initiated this project to develop a language for digital devices such as set-top boxes, televisions etc. But, it was suited for **internet programming**. Later, Java technology was incorporated by Netscape. There are given the major points that describes the history of java

- 1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
- 2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.
- 3) Firstly, it was called "**Greentalk**" by James Gosling and file extension was .gt.
- 4) After that, it was called **Oak** and was developed as a part of the Green project.
- 5) Why Oak? **Oak is a symbol of strength** and chosen as a **national tree** of many countries like **U.S.A., France, Germany, Romania** etc.



- 6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.
- 7) Why had they chosen java name for java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.
- 8) Java is an island of Indonesia where first coffee was produced (called java coffee).
- 9) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- 10) In 1995, Time magazine called Java one of the ten best Products of 1995.
- 11) JDK 1.0 released in January 23, 1996.



4.7. History of JAVA versions:

The following are many java versions that have been released.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)
11. Java SE 9 (21st Sep, 2017)
12. Java SE 10 (March, 2018)
13. Java SE 11 (September, 2018)
14. Java SE 12 (March, 2019)

Current stable release of Java is Java SE 9.

4.8. Difference between C, C++ and Java:

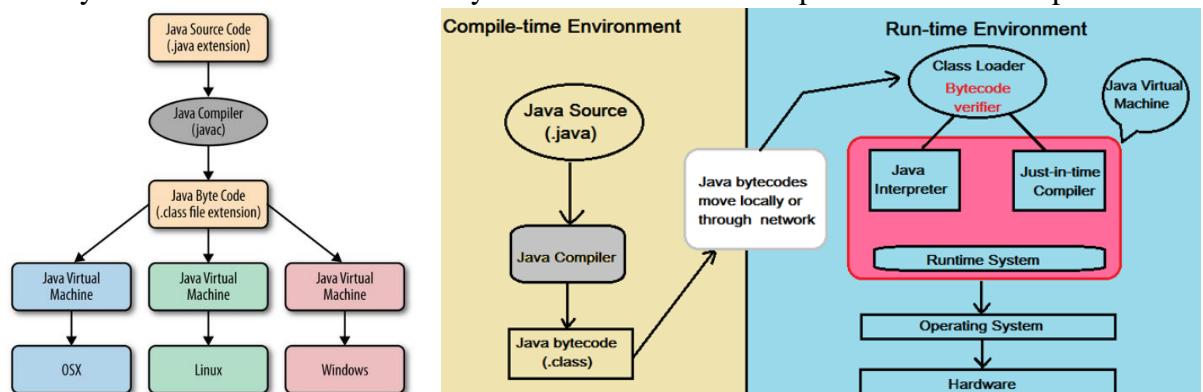
The following are the major differences between C, C++ and Java.

Aspects	C	C++	Java
Developed Year	1972	1979	1991
Developed By	Dennis Ritchie	Bjarne Stroustrup	James Gosling
Successor of	BCPL	C	C(Syntax) & C++ (Structure)
Paradigms	Procedural	Object Oriented	Object Oriented
Platform Dependency	Dependent	Dependent	Independent
Keywords	32	63	50
Datatypes : union, structure	Supported	Supported	Not Supported
Pre-processor directives	Supported (#include, #define)	Supported (#include, #define)	Not Supported
Header files	Supported	Supported	Use Packages (import)
Inheritance	No Inheritance	Supported	Multiple Inheritance not Supported

Overloading	No Overloading	Supported	Operator Overloading not Supported
Pointers	Supported	Supported	No Pointers
Code Translation	Compiled	Compiled	Interpreted
Storage Allocation	Uses malloc, calloc	Uses new, delete	uses garbage collector
Multi-threading and Interfaces	Not Supported	Not Supported	Supported
Exception Handling	No Exception handling	Supported	Supported
Templates	Not Supported	Supported	Not Supported
Storage class: auto, extern	Supported	Supported	Not Supported
Destructors	No Constructor or Destructor	Supported	Not Supported
Database Connectivity	Not Supported	Not Supported	Supported

4.9. Java's Magic - The Bytecode:

- The output of a Java compiler is not executable code rather it is **bytecode**.
- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system which is called **Java Virtual Machine (JVM)**.
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.



- Although the details of the **JVM differ from platform to platform**, all understands the same Java Bytecode. The execution of bytecode by the JVM is the easiest way to create truly portable programs.
- JVM prevents the program from generating side effects outside of the system and hence helps to make it secure.
- As Java program is **compiled to an intermediate form and then interpreted by a JVM** it runs slower than programs of other languages. However, the Java Bytecode is highly optimized, and enables the JVM to execute programs in a much faster rate.
- The **HotSpot technology** of java provides a **Just-In-Time (JIT)** compiler for bytecode. The JIT compiler is part of the JVM. It **compiles the selected portion of the bytecode into executable code in real-time, on a piece-by-piece, as it is needed, during execution**. This increases the performance of execution of Java Programs.

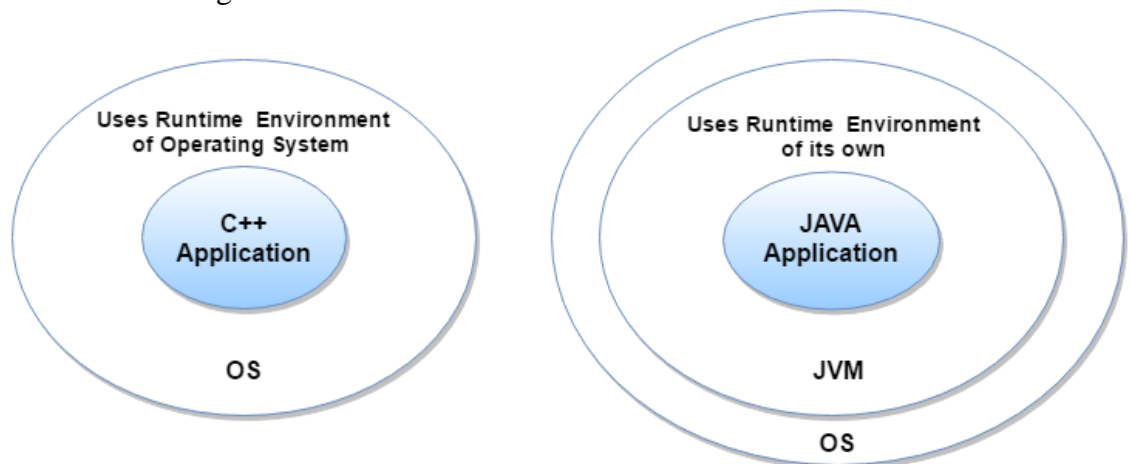
4.10. Features of JAVA:

The **main objective** of Java programming language creation was to make it **portable, simple and secure programming language**. Apart from this, there are also some awesome features which play important role in the popularity of this language. The features of Java are also

known as **java buzzwords**. Following is a list of most important features of Java language. The Java Features given below are simple and easy to understand.

1. **Simple:** Java is very **easy to learn** and its **syntax is simple**, clean and **easy to understand**. According to Sun, Java language is simple because:
 - Syntax is based on C/C++ (easier for programmers to learn it after C++).
 - Removed many confusing and/or rarely-used features e.g., **explicit pointers**, **operator overloading** etc.
 - No need to remove unreferenced objects because there is **Automatic Garbage Collection** in java.
2. **Object-Oriented:** Java is Object-oriented programming language. **Everything in Java is an object**. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules. The **object model in Java is simple and easy to extend**, while primitive types, such as integers, are kept high-performance non objects.
3. **Robust:** Robust simply means strong. Java is robust because:
 - It uses **strong memory management** to allocate and deallocate memory.
 - There is **automatic garbage collection** in java.
 - There is **lack of pointers that avoids security problem**.
 - There is object-oriented **exception handling** and type checking mechanism in java.
4. **Multithreaded:** A thread is like a separate program, executing concurrently. Java supports multithreaded programming, which allows you to write programs that do **many things simultaneously**. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it **doesn't occupy memory for each thread**. It **shares a common memory area**. Threads are important for multi-media, Web applications etc.
5. **Architecture-neutral:** Java is architecture neutral because there is **no implementation dependent features** e.g. size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.
6. **Interpreted and High Performance:** Java enables the creation of cross-platform programs by compiling into an intermediate representation called **Java bytecode**. This code can be executed on any system that implements the **Java Virtual Machine**. Java bytecode is carefully designed so that it would be **easy to translate directly into native machine code** for high performance without losing any benefits of the platform-independent code. Java is **faster than traditional interpretation** since bytecode is "close" to native code still somewhat slower than a compiled language (e.g., C++). Java is an interpreted language, so it is also a reason that why it is slower than compiled language C, C++.
7. **Distributed:** Java is distributed because it **facilitates us to create distributed applications** in java. It is designed for the distributed environment of the Internet because it handles **TCP / IP** protocols. **RMI and EJB** are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.
8. **Secured:** Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
 - It has no explicit pointer

- Java Programs run inside virtual machine sandbox



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to dynamically load Java classes into the Java Virtual Machine. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
 - **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
 - **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.
9. **Platform independent:** Java is platform independent because it is different from other languages like C, C++ etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components: Runtime Environment, Application Programming Interface (API). Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac / OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere (WORA).

10. **Portable:** Java is portable because it facilitates you to carry the java byte code to any platform.
11. **Dynamic:** Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

4.11. First Java Program:

Requirement for Running a Java Program:

For executing any java program, we need to:

- Download the JDK and install it.
- Set path of the jdk/bin directory.
- Create the java program
- Compile and run the java program

Program:

To write a simple Java program, open notepad or (Notepad++) and write simple program as displayed below:

```
/*
    My First Java Program
```

```

        Save this file with name "First.java"
*/
class First{
    public static void main(String args[]){
        System.out.println("This is my first java program.");
    }
}

```

Save as: First.java

Compile: javac First.java

Run: java First

Output: This is my first java program.

To compile and run this program, you need to open command prompt by either of the following two ways:

- From Notepad++: File → Open Containing Folder → cmd
- From start button: Start menu → All Programs → Accessories → command prompt.

NOTE:

- **Extension** of a java file should be **.java**.
- In java all **code must reside inside a class**.
- **Name of the class** should **match** the **name of the file** that holds the program.
- Java is **case sensitive**.
- After compilation the java compiler called **javac** converts the source code First.java into its bytecode First.class.
- When Java source code is compiled, **each individual class** is put into its own output file named after the class and uses the .class extension.
- The java interpreter called **java** uses the class name as a command-line argument. The interpreter searches for a file by that name that has the .class extension.
- The first four lines are for multiline commenting.
- **class** keyword is used to declare a class in java.
- **First** is the class name. In Java all program activity occurs within class. That is why all Java programs are object-oriented.
- Program execution begins by calling **main()**.
- The **public** keyword is an **access modifier**. The other type of access specifier is **private**. Access specifier allows the programmer to **control the visibility** of class members. Public members can be accessed by code outside the class in which it is declared. That means **public** members are visible to all. The function **main()** must be declared as public, since it must be called by code outside of its class when the program is started.
- **static** is a keyword, if we declare any method as static, it is known as static method. The keyword **static** allows **main()** to be called without having to instantiate a particular instance of the class. This is a necessary requirement since **main()** is called by the *Java Virtual Machine* before any objects are made.
- The keyword **void** simply tells the compiler that **main()** does not return a value.
- A Java program will be compiled with no error even if it does not contain a **main()** method in any of the classes present in it. However, the interpreter **java** would report an error because it would be unable to find the **main()** method.
- In **main()**, there is only one parameter **String args[]**. Here **String** is a class and **args** is array of instances of the class **String**. Here, **args** receives any command-line arguments present when the program is executed.

- A program may have many classes, out of which only one of which will need to have a `main()` method to get things started.
- `System` is a class in the `java.lang` package.
- `out` is a static member of the `System` class, and is an instance of `java.io.PrintStream`. `out` is the output stream connected to the console.
- `println()` is a method of `java.io.PrintStream`. This method is overloaded to print message to output destination, which is typically a console or file.
- The relation between `System` class and `PrintStream` class is HAS-A relation. Here `System` class HAS-A `PrintStream` class. To understand the relation we need to understand the following program.

```
class A
{
    void display()
    {
        System.out.println("this is display method");
    }
}

class B
{
    static A ob=new A();
}

class demo
{
    public static void main(String args[])
    {
        B.ob.display();
    }
}
```

- It prints this is display method. `B.ob.display()` is just like `System.out.println()`. A object is created in B class. `PrintStream` class object is created in `System` class. `ob` is static object reference of A class. `out` also static reference of `PrintStream` class.
- All statements in Java end with a semicolon.

Different ways of writing a java program:

There are many ways to write a java program. The modifications that can be done in a java program are given below:

- 1) By changing sequence of the modifiers, method prototype is not changed.

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

- 2) Subscript notation in java array can be used after type, before variable or after variable:

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

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

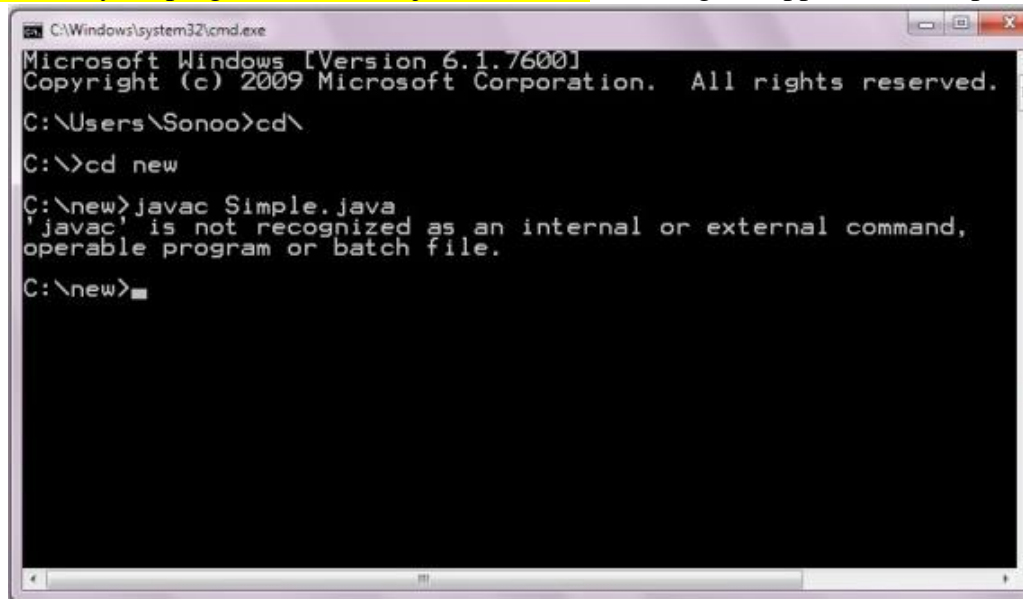
- 3) We can provide variable arguments support to main method by passing 3 ellipses (dots).

```
public static void main(String... args)
```

- 4) Having semicolon at the end of class in java is optional.

Resolving an error "javac is not recognized as an internal or external command"

If there occurs a problem like displayed in the below figure, you need to set path. Since DOS doesn't know javac or java, we need to set path. **Path is not required in such a case if you save your program inside the jdk/bin folder.** But its good approach to set path.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.
C:\new>
```

How to set path in Java:

There are 2 ways to set java path:

1) How to set Temporary Path of JDK in Windows

To set the temporary path of JDK, you need to follow following steps:

- Open command prompt
- Copy the path of jdk/bin directory
- Write in command prompt: set path=copied_path.

For example:

```
set path=C:\Program Files\Java\jdk1.6.0_23\bin
```

2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

Go to MyComputer properties → advanced tab → environment variables → new tab of user variable → write path in variable name → write path of bin folder in variable value → ok → ok → ok

3) Setting Java Path in Linux

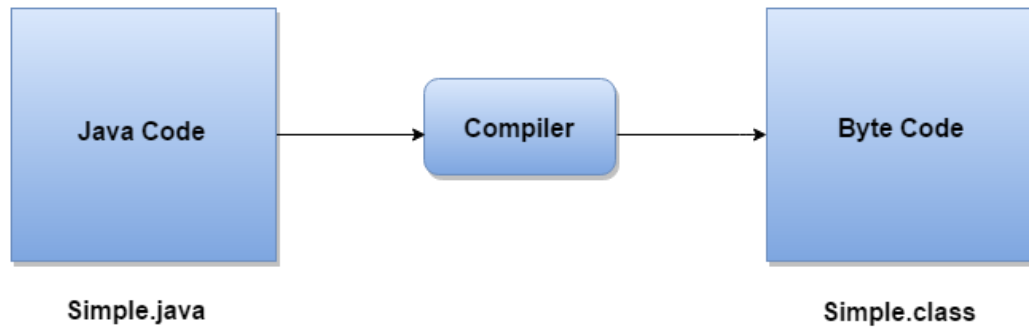
Setting path in Linux OS is same as setting path in the Windows OS. But here we use export tool rather than set. To set path in Linux OS:

```
export PATH=$PATH:/home/jdk1.6.01/bin/
```

Note that, we have installed the JDK in the home directory under Root (/home).

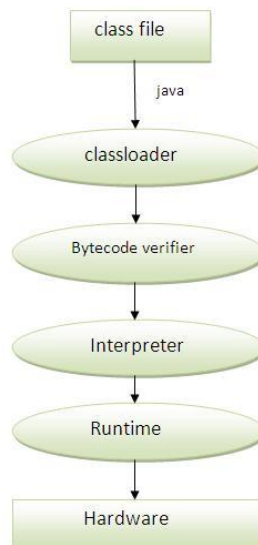
What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



What happens at run time?

At runtime, following steps are performed:



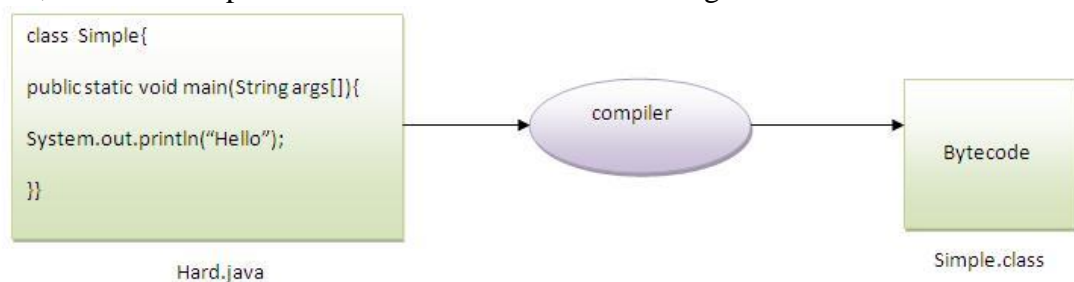
Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions.

Can you save a java source file by a name other than the class name?

Yes, but after compilation it will create a .class file using the class name.

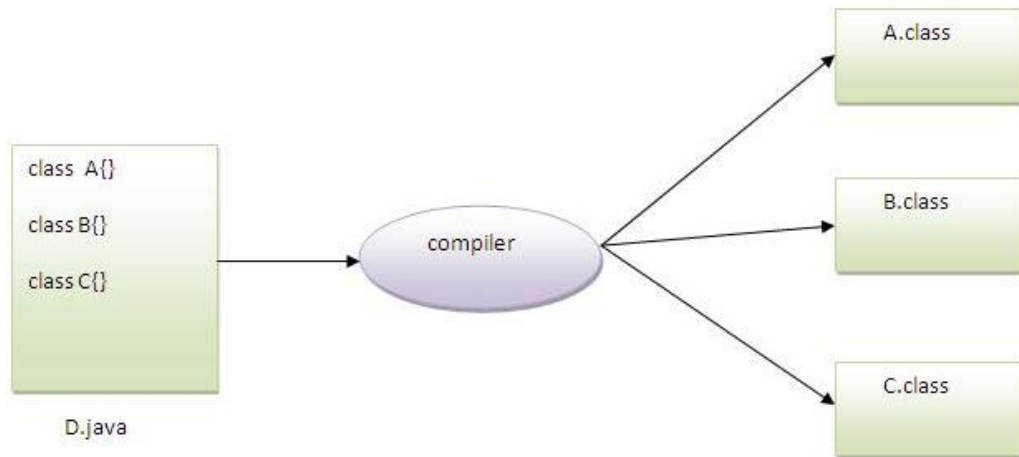


To compile: javac Hard.java

To run: java Simple

Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



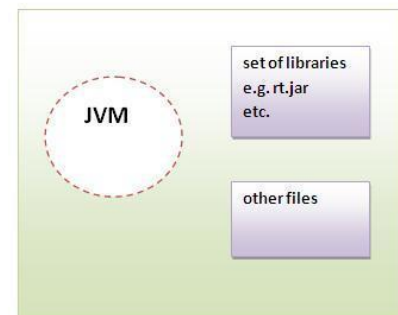
To compile: `javac D.java`

To run: `java C` [Assuming class C contains the main method.]

4.12. Difference between JDK, JRE and JVM:

JVM:

- **JVM** (Java Virtual Machine) is not a machine, **it is a program.**
- JVM is an abstract machine. It is a **specification that provides runtime environment** in which java bytecode can be executed.
- JVMs are available for many hardware and software platforms (i.e. **JVM is platform dependent**).
- JVMs are available for many hardware and software platforms. **JVM, JRE and JDK are platform dependent** because configuration of each OS differs. But, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.
- The JVM performs following main tasks:
 1. Loads code
 2. Verifies code
 3. Executes code
 4. Provides runtime environment
- JVM provides definitions for the:
 1. Memory area
 2. Class file format
 3. Register set
 4. Garbage-collected heap
 5. Fatal error reporting etc.



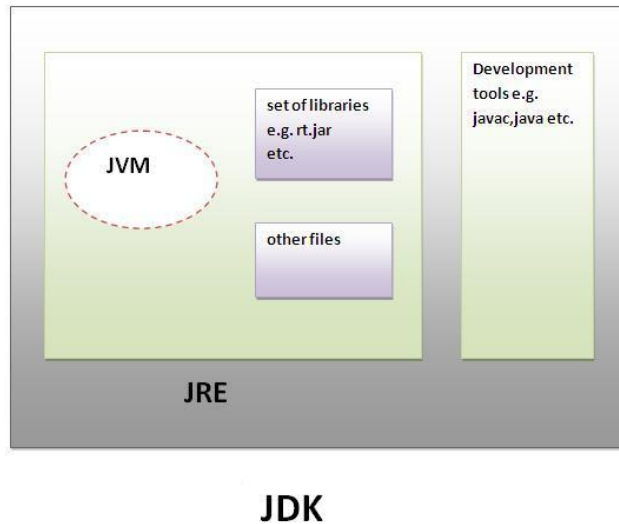
JRE

JRE:

- JRE is an acronym for **Java Runtime Environment**. It is used to provide runtime environment. It is the **implementation of JVM**. It physically exists. It **contains set of libraries plus other files that JVM uses at runtime.**
- Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.

JDK:

- JDK is an acronym for Java Development Kit. It physically exists. It contains JRE plus development tools.



4.13. Second Short Java Program:

Program:

To write another simple Java program, open notepad or (Notepad++) and write the following simple program as displayed below:

```

class Second{
    public static void main(String args[]){
        int num;    // declaration of variable num
        num = 100;  // variable num is assigned with a value 100
        System.out.println("This is num: " + num);
        num = num * 2;
        System.out.print("The value of num * 2 is ");
        System.out.println(num);
    }
}

```

Output: This is num: 100
 The value of num * 2 is 200

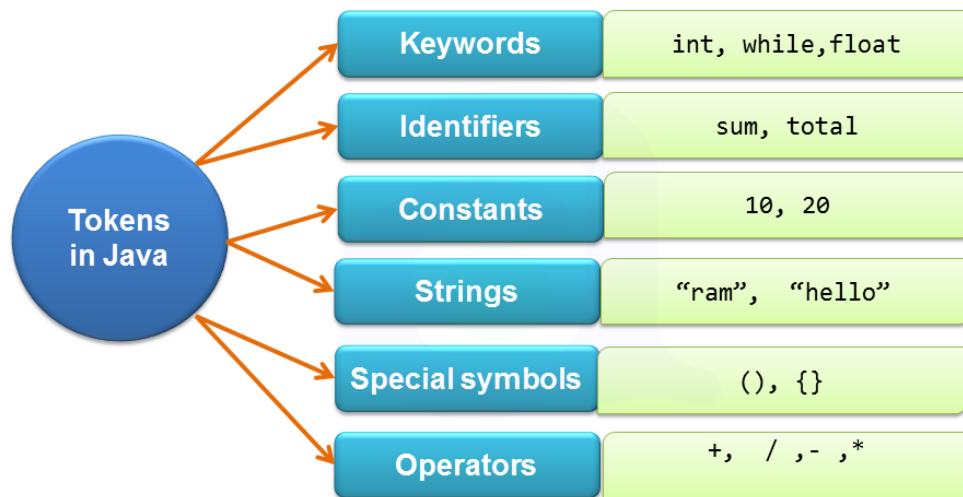
NOTE:

- **Variable declaration:**
 Syntax: data-type var-name.
 Example: int num;
- In the first println() statement num is first converted from an integer into its string equivalent and then concatenated with the string that precedes it. We can join together as many items as we want within a single println() statement.
- The print() method is just like println(), except that it does not output a new line character after each call.

5. UNDERSTANDING EVERY TERM OF JAVA PROGRAM:

5.1. Tokens

Tokens are the **various Java program elements which are identified by the compiler**. A token is the smallest element of a program that is meaningful to the compiler. Tokens supported in Java include keywords, variables, constants, special characters, operators etc.



When you compile a program, the compiler scans the text in your source code and extracts individual tokens. While tokenizing the source file, the compiler recognizes and subsequently removes white spaces (spaces, tabs, newline and form feeds) and the text enclosed within comments.

5.2 Keywords: Keywords are the reserve words whose meanings cannot be changed. Here is a list of keywords in the Java programming language:

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

You cannot use any of the following as identifiers in your programs. The keywords **const** and **goto** are **reserved, even though they are not currently used**. **True**, **false**, and **null** might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

5.3 Identifiers: Identifiers are the names of variables, methods, classes, packages and interfaces. Unlike literals they are not the things themselves, just ways of referring to them. In the First.java program, First, String, args, main and println are identifiers.

Rules for creating identifiers:

- Should be **single word**. That is spaces are not allowed.
Example: mangoprice is valid but mango price is not valid.
- Should **start with a letter (alphabet) or underscore or \$ symbol**.
Example: price, _price and \$price are valid identifiers.
- Should not be a keyword** of Java as keyword carries special meaning to the compiler.
Example: class or void etc.
- Should not start with a digit but digit can be in the middle or at the end.
Example: 5mangoescost is not valid and mango5cost and mangocost5 are valid.
- Length** of an identifier in Java can be of **65,535 characters** and all are significant.
- Identifiers are **case-sensitive**. That is both mango and Mango are treated differently.

- Can contain all uppercase letters or lowercase letters or a mixture.

Note: Always choose the identifier a meaningful word. Suppose you would like to store the number of mangoes in the code, write as

`int numberOfMangoes = 10;` instead of `int x =10;`

5.4 Variables: Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

`int data=50; //Here data is variable`

There are three types of variables in java:

1. **local variable:** A variable declared inside the method is called local variable.
2. **instance variable:** A variable declared inside the class but outside the method, is called instance variable. It is not declared as static.
3. **static variable:** A variable which is declared as static is called static variable. It cannot be local.

Example:

```
class A{
    int data=50; //instance variable
    static int m=100; //static variable
    void method(){
        int n=90; //local variable
    }
} //end of class
```

5.5 Literals: Any constant value which can be assigned to the variable is called as literal/constant.

`// Here 100 is a constant/literal.`
`int x = 100;`

The following are different types of literals:

1. **Integer Literal:** For Integral data types (byte, short, int, long), we can specify literals in 4 ways:

- a) **Decimal literals (Base 10):** In this form the allowed digits are 0-9.

`int x = 107;`

- b) **Octal literals (Base 8):** In this form the allowed digits are 0-7. The octal number should be prefix with 0.

`int x = 0146;`

- c) **Hexa-decimal literals (Base 16):** In this form the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that java is a case-sensitive programming language but here java is not case-sensitive. The hexa-decimal number should be prefix with 0X or 0x.

`int x = 0X123Face;`

- d) **Binary literals:** From 1.7 onward we can specify literals value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.

`int x = 0b1111;`

2. **Character Literal:** For char data types we can specify literals in 4 ways:

- a) **Single quote:** We can specify literal to char data type as single character within single quote.

`char ch = 'a';`

- b) **Char literal as Integral literal:** We can specify char literal as integral literal which represents Unicode value of the character and that integral literals can be specified either in Decimal, Octal and Hexadecimal forms. But the allowed range is 0 to 65535


```
char ch = 062;
```

- c) **Unicode Representation:** We can specify char literals in Unicode representation `'\uxxxx'`. Here xxxx represents 4 hexadecimal numbers.

```
char ch = '\u0061'; // Here \u0061 represent a.
```

- d) **Escape Sequence:** Every escape character can be specify as char literals

```
char ch = '\n';
```

3. **String literal:** Any sequence of characters within double quotes is treated as String literals.

```
String s = "Hello";
```

String literals may not contain un-escaped newline or linefeed characters. However, the Java compiler will evaluate compile time expressions, so the following String expression results in a string with three lines of text:

Example: String text = "This is a String literal\n" + "which spans not one and not two\n" + "but three lines of text.\n";

4. **boolean literal:** Only two values are allowed for Boolean literals i.e. `true` and `false`.

```
boolean b = true;
```

5.6 Data Types: Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- a) **Primitive data types:** There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. These primitive data types are put into the following four groups:

- **Integers:** This group includes `byte`, `short`, `int`, `long`, which are whole valued signed (no unsigned in Java) numbers.
- **Floating-point numbers:** This group includes `float` and `double`, which represent numbers with fractional precision.
- **Characters:** This group includes `char`, which represents symbols in a character set like letters and numbers.
- **Boolean:** This group includes `boolean`, which is a special type for representing true/false values.

The eight primitive data types in Java are:

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
- The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

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

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

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

float

- Float data type is a single-precision 32-bit IEEE-754 floating point
- Minimum value is $1.4e-045$
- Maximum value is $3.4e+038$
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE-754 floating point
- Minimum value is $4.9e-324$
- Maximum value is $1.8e+308$
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Default value is 0.0d
- Example: double d1 = 123.4

boolean

- boolean data type represents one bit of information
- The values of a boolean type variable are either true or false
- 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

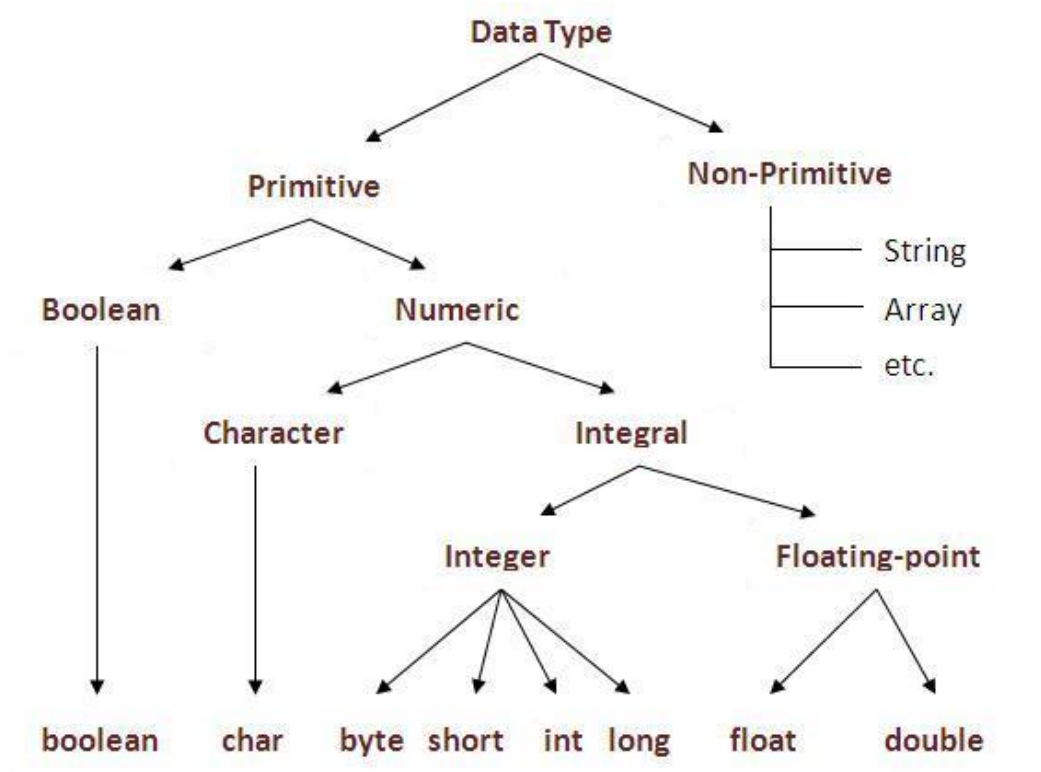
char

- 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'.
- Although char is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example you can add two character variables, or increment the value of a character variable.

NOTE: Because of Java's portability requirement, all data types have a strictly defined range. For example int is always 32 bits, regardless of the particular platform.

b) Non-primitive (Reference/Object Data Types) data types

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: `Animal animal = new Animal("giraffe");`



Type	Description	Default	Size	Example Literals
boolean	true or false	false	1 bit	true, false
byte	twos complement integer	0	8 bits	(none)
char	Unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'', '\n', '\B'
short	twos complement integer	0	16 bits	(none)
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d

5.7 Escape Sequences: Java language supports few special escape sequences for String and char literals as well. They are:

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)

5.8 Unicode System: Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Before Unicode, there were many language standards:

- ASCII (American Standard Code for Information Interchange) for the United States.
- ISO 8859-1 for Western European Language.
- KOI-8 for Russian.
- GB18030 and BIG-5 for chinese, and so on.

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, other require two or more byte.

To solve these problems, a new language standard was developed i.e. Unicode System. In unicode, character holds 2 byte, so java also uses 2 byte for characters.

Lowest value: \u0000, Highest value: \uFFFF

5.9 Operators: Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

5.9.1 Arithmetic Operators: Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

NOTE:

1. Unlike in C, the modulus operator can be applied to floating-point types as well as integer types.

Example:

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

        int x=42;
        float y=42.25f;

        System.out.println("x mod 10: "+ x%10);
        System.out.println("y mod 10: "+ y%10);
    }
}
```

Output:

```
X mod 10: 2
Y mod 10: 2.25
```

2. Operator + is also used for string concatenation.

Example:

```
class TestStringConcatenation{
    public static void main(String args[]){
        String s="Sachin"+" Tendulkar";
        System.out.println(s);
    }
}
```



```
}
Output:
Sachin Tendulkar
```

3. The string concatenation operator can concat not only string but primitive values also.

Example:

```
class TestStringConcatenation{
    public static void main(String args[]){
        String s=40+40+"Sachin"+40+40;
        System.out.println(s);
    }
}
```

Output:
80Sachin4040

Example: Unary Operators ++, --

```
class OperatorExample{
    public static void main(String args[]){
        int x=10;
        System.out.println(x++); //10 (11)
        System.out.println(++x); //12
        System.out.println(x--); //12 (11)
        System.out.println(--x); //10
    }
}
```

Output:
10
12
12
10

5.9.2 Relational Operators: There following are the relational operators supported by Java language:

Assume integer variable A holds 10 and variable B holds 20, then

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.

>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

5.9.3 Bitwise Operators: Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

a & b = 0000 1100

a | b = 0011 1101

a ^ b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000

>> (right shift) arithmetic shift right	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift) logical shift right Unsigned right shift	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. For positive number, >> and >>> works same. For negative number, >>> changes parity bit (MSB) to 0	A >>>2 will give 15 which is 0000 1111

Example: Operator ~

```
class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=-10;
        System.out.println(~a); //-11 (minus of total positive value
        which starts from 0)
        System.out.println(~b); //9 (positive of total minus, positive
        starts from 0)    }
    }
```

Output:

```
-11
9
```

Example: Operator <<

```
class OperatorExample{
    public static void main(String args[]){
        System.out.println(10<<2); //10*2^2=10*4=40
        System.out.println(10<<3); //10*2^3=10*8=80
        System.out.println(20<<2); //20*2^2=20*4=80
        System.out.println(15<<4); //15*2^4=15*16=240}
    }
```

Output:

```
40
80
80
240
```

Example: Operator >> vs >>>

```
class OperatorExample{
    public static void main(String args[]){
        //For positive number, >> and >>> works same
        System.out.println(20>>2);
        System.out.println(20>>>2);
        //For negative number, >>> changes parity bit (MSB) to 0
        System.out.println(-20>>2);
        System.out.println(-20>>>2);
    }
}
```

Output:

5
5
-5

1073741819

5.9.4 Logical Operators: The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
(logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Example: Operator Logical || and Bitwise |

```

class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=5;
        int c=20;
        System.out.println(a>b||a<c);//true || true = true
        System.out.println(a>b|a<c);//true | true = true
        ///| vs |
        System.out.println(a>b||a++<c);//true || true = true
        System.out.println(a);//10 because second condition is not
        checked
        System.out.println(a>b|a++<c);//true | true = true
        System.out.println(a);//11 because second condition is
        checked
    }
}

```

Output:

true
true
true
10
true
11

NOTE: The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false. The bitwise | operator always checks both conditions whether first condition is true or false.

5.9.5 Assignment Operators: Following are the assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	$C \% = A$ is equivalent to $C = C \% A$
<<=	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
>>=	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
&=	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
^=	bitwise exclusive OR and assignment operator.	$C \wedge= 2$ is same as $C = C \wedge 2$
=	bitwise inclusive OR and assignment operator.	$C = 2$ is same as $C = C 2$

Example: Adding short

```
public class Test {  
    public static void main(String args[]) {  
        short a=10;
```

```

        short b=10;
        a=a+b;//Compile time error because 10+10=20 now int
        System.out.println(a);
    }
}

```

Output:

Compile time error

Example: Adding short

```

public class Test {
    public static void main(String args[]) {
        short a=10;
        short b=10;
        a=(short)(a+b);//20 which is int now converted to short
        System.out.println(a);
    }
}

```

Output:

20

NOTE: Assignment operator (=) allows us to create a chain of assignments.

Example: int x, y, z;
 x = y = z =10; // set x, y, and z to 10.

5.9.6 Miscellaneous Operators: There are few other operators supported by Java Language.

a) Conditional Operator (? :): Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

Example:

```

public class Test {
    public static void main(String args[]) {
        int a, b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "value of b is : " + b );

        b = (a == 10) ? 20: 30;
        System.out.println( "value of b is : " + b );
    }
}

```

Output:

Value of b is : 30

Value of b is : 20

b) instanceof Operator: This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). *instanceof* operator is written as:

(Object/reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true.

Example:

```

public class Test {
    public static void main(String args[]) {
        String name = "James";
        // following will return true since name is type of String
        boolean result = name instanceof String;
        System.out.println( result );
    }
}

```

Output:

```

true

```

5.9.7 Precedence of Java Operators: Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Separators	(), [], . (dot operator)	Left to right
Postfix	++, --	Right to left
Prefix, Unary	++, --, ~, !, +(unary), -(unary), type-cast	Right to left
Multiplicative	*, /, %	Left to right
Additive	+, -	Left to right
Shift	>>, >>>, <<	Left to right
Relational, Special	>, >=, <, <=, instanceof	Left to right
Equality	==, !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Arrow	->	Left to right
Assignment	>= += -= *= /= %= >>= <<= &= ^= =	Right to left

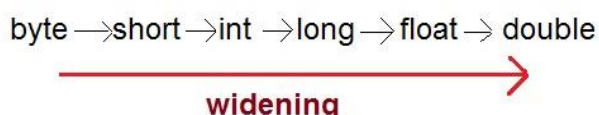
5.10 Type Conversion: When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.

a) Widening or Automatic (Implicit) Type Conversion or coercion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- The destination type is larger than the source type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.



Example:

```
class Test{
    public static void main(String[] args){
        int i = 100;
        //automatic type conversion
        long l = i;
        //automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Output:

```
Int value 100
Long value 100
Float value 100.0
```

b) Narrowing or Explicit Conversion.

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.



char and number are not compatible with each other. Let's see when we try to convert one into other.

Example:

```
//Java program to illustrate incompatible data
// type for explicit type conversion
class Test
{
    public static void main(String[] argv)
    {
        char ch = 'c';
        int num = 88;
        ch = num;          //Correct code: ch = (char)num;
        System.out.println(ch);
    }
}
```

Output:

```
7: error: incompatible types: possible lossy conversion from int to char
ch = num;
    ^
1 error
```

Example:

```
//Java program to illustrate incompatible data
// type for explicit type conversion
class Test
{
    public static void main(String[] argv)
    {
        double d = 100.04;
        //explicit type casting
        int i = (int)d;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:

```
Double value 100.04
Int value 100
```

While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).

Example:

```
//Java program to illustrate Conversion of int and double to byte
class Test
{
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
```

```

        System.out.println("Conversion of int to byte.");

        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }
}

```

Output:

Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b = 67

Type promotion in Expressions:

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

- Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

The following code shows error:

```

byte b = 50;
b = b * 2; //error

```

The reason is first b (in bytes) is converted to int before multiplying with 2. After multiplication 100(in integers) cannot be stored in b (in bytes). The correct code would be:

```

b = (byte) (b * 2);

```

Example:

```

//Java program to illustrate Type promotion in Expressions
class Test
{
    public static void main(String args[])
    {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = 0.1234;

        // The Expression
        double result = (f * b) + (i / c) - (d * s);

        //Result after all the promotions are done
        System.out.println("result = " + result);
    }
}

```

```
}  
}
```

Output:

Result = 626.7784146484375

5.11 Java comments:

- **Single Line Comment:** The single line comment is used to comment only one line.

Syntax:

```
// This is single line comment
```

Example:

```
int i=10; //Here, i is a variable
```

- **Multi Line Comment:** The multi-line comment is used to comment multiple lines of code.

Syntax:

```
/*  
  
    This is multi line comment  
  
*/
```

Example:

```
/*  
  
    My First Java Program  
    Save this file with name "First.java"  
  
*/
```

- **Java Documentation Comment:** The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

```
/**  
  
    This is documentation comment  
  
*/
```

Example:

```
/** The Calculator class provides methods to get addition and  
    subtraction of given 2 numbers.*/  
public class Calculator  
{  
    /** The add() method returns addition of given numbers.*/  
    public static int add(int a, int b)  
    {  
        return a+b;  
    }  
  
    /** The sub() method returns subtraction of given numbers.*/  
    public static int sub(int a, int b)  
    {  
        return a-b;  
    }  
}
```

After compilation create Documentation API by javadoc tool:

javadoc Calculator.java

Now, there will be HTML files created for your Calculator class in the current directory. Open the HTML files and see the explanation of Calculator class provided through documentation comment.

5.12 Control Structures: A control structure is a block of programming that analyses variables and chooses a direction in which to go based on given parameters. The term flow control details the direction the program takes (which way program control "flows"). Hence it is the basic decision-

making process in computing; flow control determines how a computer will respond when given certain conditions and parameters.

If-else Statement:

The if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in java.

- if statement
- if-else statement
- if-else-if ladder
- nested if statement

if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

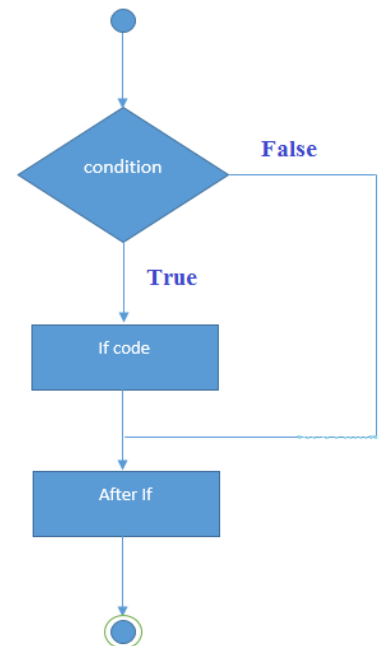
```
if(condition){  
    //code to be executed  
}
```

Example:

```
public class IfExample {  
    public static void main(String[] args) {  
        int age=20;  
        if(age>18){  
            System.out.print("Age is greater than 18");  
        }  
    }  
}
```

Output:

Age is greater than 18



if-else Statement

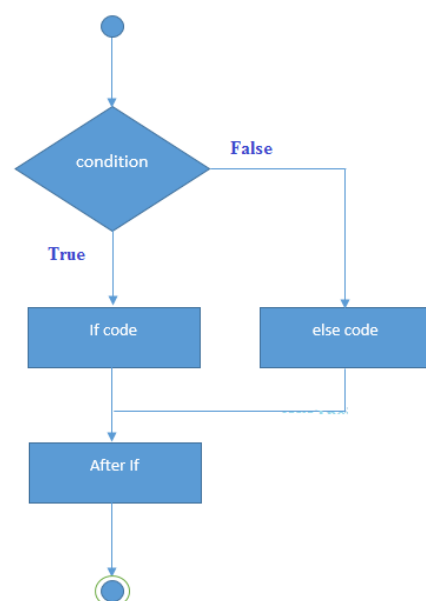
The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){  
    //code if condition is true  
}  
else{  
    //code if condition is false  
}
```

Example:

```
public class IfElseExample {  
    public static void main(String[] args){  
        int number=13;  
        if(number%2==0){  
            System.out.println("even  
number");  
        }  
        else
```




```

        {
            system.out.println("odd number");
        }
    }
}

```

Output:

odd number

if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```

if(condition1){
    //code to be executed if condition1 is true
}
else if(condition2){
    //code to be executed if condition2 is true
}
else if(condition3){
    //code to be executed if condition3 is true
}
...
else{
    //code to be executed if all the conditions are false
}

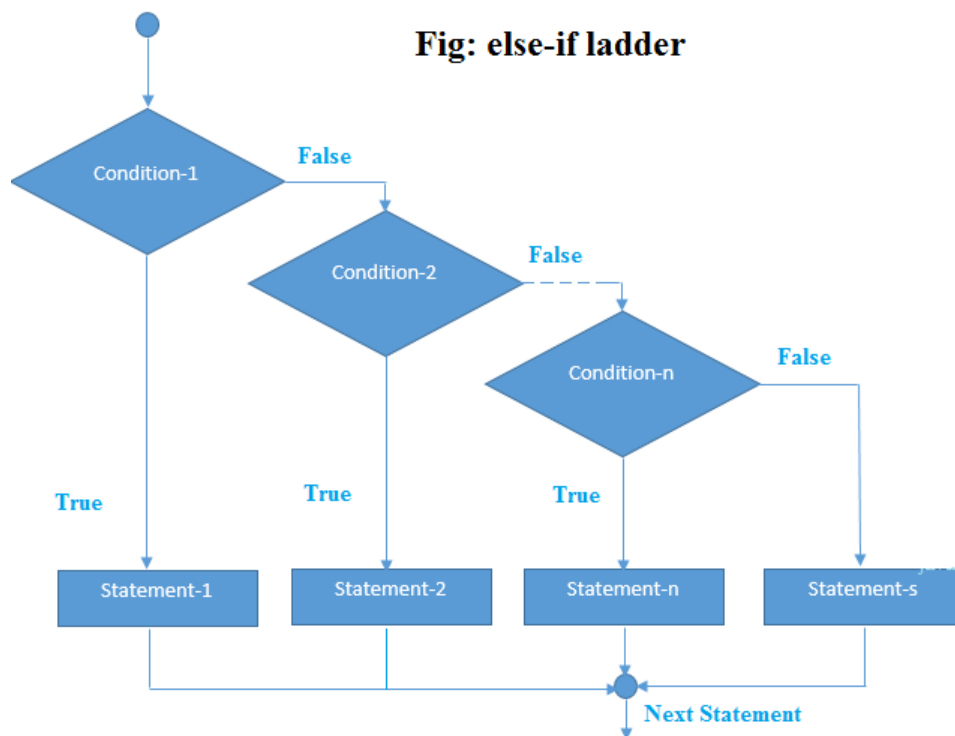
```

Example:

```

public class IfElseIfExample {
    public static void main(String[] args) {
        int marks=65;
    }
}

```



```

if(marks<50){
    system.out.println("fail");
}

```

```

    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }
    else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }
    else{
        System.out.println("Invalid!");
    }
}
}

```

Output:

C grade

Switch Statement:

The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement.

Syntax:

```

switch(expression){
    case value1:
        //code to be executed;
        break; //optional
    case value2:
        //code to be executed;
        break; //optional
    .....
    default:
        code to be executed if all cases are not matched;
}

```

Example:

```

public class SwitchExample
{
    public static void main(String[] args)
    {
        int number=20;
        switch(number)
        {
            case 10: System.out.println("10");break;
            case 20: System.out.println("20");break;
            case 30: System.out.println("30");break;
            default: System.out.println("Not in 10,20or30");
        }
    }
}

```

```

    }
}
}

```

Output:

20

Note: The java switch statement is fall-through. It means it executes all statement after first match if break statement is not used with switch cases.

Example:

```

public class SwitchExample2 {
    public static void main(String[] args) {
        int number=20;
        switch(number){
            case 10: System.out.println("10");
            case 20: System.out.println("20");
            case 30: System.out.println("30");
            default: System.out.println("Not in 10, 20 or 30");
        }
    }
}

```

Output:

20

30

Not in 10, 20 or 30

Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in java.

- for loop
- while loop
- do-while loop

1. for Loop

The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

There are three types of for loops in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

```

for(initialization; condition; incr/decr){
    //code to be executed
}

```

Example:

```

public class ForExample
{
    public static void main(String[] args)
    {
        for(int i=1;i<=10;i++)
        {
            System.out.println(i);
        }
    }
}

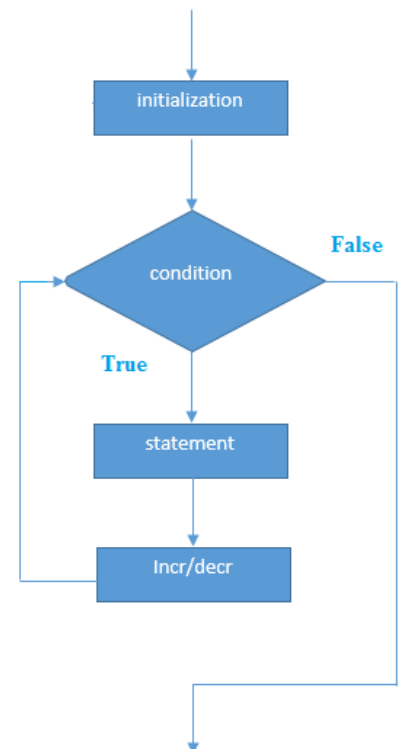
```

Output:

```

1
2
3
4
5
6
7
8
9
10

```

**for-each Loop**

The for-each loop is used to **traverse array or collection in java**. It is easier to use than simple for loop because **we don't need to increment value and use subscript notation**.

It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

```

for(Type var:array){
    //code to be executed
}

```

Example:

```

public class ForEachExample
{
    public static void main(String[] args)
    {
        int arr[]={12,23,44,56,78};
        for(int i:arr)
        {
            System.out.println(i);
        }
    }
}

```

Output:

```

12
23
44
56
78

```

Labeled For Loop

We can have **name of each for loop**. To do so, we use label before the for loop. It is **useful if we have nested for loop so that we can break/continue specific for loop**. Normally, break and continue keywords breaks/continues the inner most for loop only.

Syntax:

```
labelname:
for(initialization;condition;incr/decr){
    //code to be executed
}
```

Example:

```
public class LabeledForExample
{
    public static void main(String[] args)
    {
        aa:
        for(int i=1;i<=3;i++)
        {
            bb:
            for(int j=1;j<=3;j++)
            {
                if(i==2 && j==2)
                {
                    break aa;
                }
                System.out.println(i+" "+j);
            }
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
```

NOTE: If you use `break bb;` instead of `break aa;`, it will break inner loop only which is the default behaviour of any loop. In that case the output will be:

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Infinitive For Loop

If you use two semicolons `;;` in the for loop, it will be infinitive for loop.

Syntax:

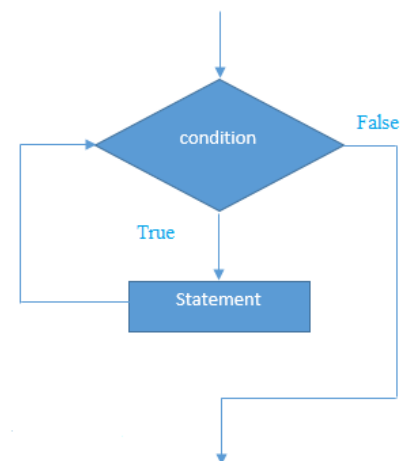
```
for(;;){
    //code to be executed
}
```

Example:

```
public class ForExample
{
    public static void main(String[] args)
    {
        for(;;)
        {
            System.out.println("infinitive loop");
        }
    }
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
...
...
...
infinitive loop
```

**2. while Loop**

The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

```
while(condition){
    //code to be executed
}
```

Example:

```
public class whileExample
{
    public static void main(String[] args)
    {
        int i=1;
        while(i<=10)
        {
            System.out.println(i);
            i++;
        }
    }
}
```

Output:

```
1
2
3
4
5
6
7
```


8
9
10

Infinitive while Loop

If you pass true in the while loop, it will be infinitive while loop.

Syntax:

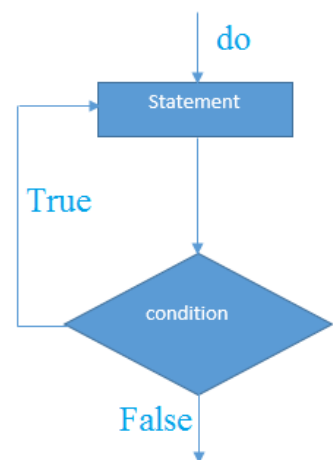
```
while(true){  
    //code to be executed  
}
```

Example:

```
public class whileExample2 {  
    public static void main(String[] args) {  
        while(true){  
            System.out.println("infinitive while loop");  
        }  
    }  
}
```

Output:

Infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c



3. do-while Loop

The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop. The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

```
do{  
    //code to be executed  
}while(condition);
```

Example:

```
public class DowhileExample  
{  
    public static void main(String[] args)  
    {  
        int i=1;  
        do{  
            System.out.println(i);  
            i++;  
        }while(i<=10);  
    }  
}
```

Output:

1
2
3

4
5
6
7
8
9
10

Infinitive do-while Loop

If you pass true in the do-while loop, it will be infinitive do-while loop.

Syntax:

```
do{  
    //code to be executed  
}while(true);
```

Example:

```
public class DowhileExample2  
{  
    public static void main(String[] args)  
    {  
        do{  
            System.out.println("infinitive do while loop");  
        }while(true);  
    }  
}
```

Output:

infinitive do while loop
infinitive do while loop
infinitive do while loop
ctrl+c

break Statement

- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- The Java break is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Syntax:

```
jump-statement;  
break;
```

Example:

```
public class BreakExample  
{  
    public static void main(String[] args)  
    {  
        for(int i=1;i<=10;i++)  
        {  
            if(i==5)  
                break;  
            System.out.println(i);  
        }  
    }  
}
```

Output:

1
2
3
4

Note: It breaks inner loop only if you use break statement inside the inner loop.

Example:

```
public class BreakExample2
{
    public static void main(String[] args)
    {
        for(int i=1;i<=3;i++)
        {
            for(int j=1;j<=3;j++)
            {
                if(i==2&& j==2)
                    break;
                System.out.println(i+" "+j);
            }
        }
    }
}
```

Output:

1 1
1 2
1 3
2 1
3 1
3 2
3 3

continue Statement

- The continue statement is used in loop control structure when you need to immediately jump to the next iteration of the loop. It can be used with for loop or while loop.
- The Java continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax:

```
jump-statement;  
continue;
```

Example:

```
public class BreakExample
{
    public static void main(String[] args)
    {
        for(int i=1;i<=10;i++)
        {
            if(i==5)
                continue;
            System.out.println(i);
        }
    }
}
```

```
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Note: It continues inner loop only if you use continue statement inside the inner loop.

Example:

```
public class BreakExample2
{
    public static void main(String[] args)
    {
        for(int i=1;i<=3;i++)
        {
            for(int j=1;j<=3;j++)
            {
                if(i==2&& j==2)
                    continue;
            }
            System.out.println(i+" "+j);
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

5.11 Arrays: An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Each item in an array is called an element, and each element is accessed by its numerical index. Numbering begins with 0.

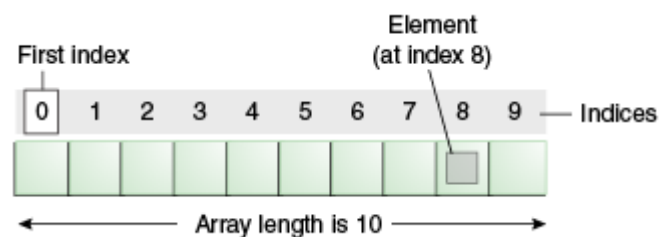
Advantage of Java Array

- **Code Optimization:** It makes the code optimized; we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only 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.

One-Dimensional Arrays:



- To create a one-dimensional array, we have to create an array variable of the required type.
- The general form of array declaration is:


```
type var-name[];
or type[] var-name;
or type []var-name;
```
- Note that **in array declaration, only array variable is created**. At this point no array actually exists.
- To link the array variable with an actual, physical array we need **to allocate memory space using a special operator named as new**.
- The general form of new is as follows:


```
array-var = new type [size];
```
- The element in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean) and null (for reference types).
- **In Java all arrays are dynamically allocated**.
- It is possible to combine the declaration of array variable with the allocation of array itself as shown below:


```
type var-name[] = new type [size];
```
- We can declare, instantiate and initialize the java array together by:


```
type var-name[] = {element1, element2, . . .};
```

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;

        //printing array
        for(int i=0; i<a.length; i++) //length is the property of array
            System.out.println(a[i]);
    }
}
```

Output:

```
10
20
70
40
50
```

Example:

```
class Testarray1
{
    public static void main(String args[])
    {
        //declaration, instantiation and initialization
        int a[]={33,3,4,50};
    }
}
```

```

        //printing array
        for(int i=0;i<a.length;i++)//length is the property of array
            System.out.println(a[i]);
    }
}

```

Output:

```

33
3
4
50

```

Passing Array to method:

We can pass the java array to method so that we can reuse the same logic on any array.

Example:

```

class Testarray2
{
    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};
        min(a);//passing array to method
    }
}

```

Output:

```

3

```

Copying Arrays:

The **System** class has an **arraycopy** method that we can use to efficiently **copy data from one array into another**:

```

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

```

Example:

```

class First
{
    public static void main(String[] args)
    {
        char[] copyFrom = {'d','e','c','a','f','f','e','i','n','a','t','e','d'};
        char[] copyTo = new char[7];
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(copyTo);
    }
}

```



```
}
```

Output:

caffeine

Multidimensional array:

- Multidimensional arrays are actually array of arrays. In such case, data is stored in row and column based index (also known as matrix form).
- The general form of array declaration is:

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

- Example to instantiate Multidimensional Array:
`int twoD[][] = new int[4][5];` //4 row and 5 column

Example:

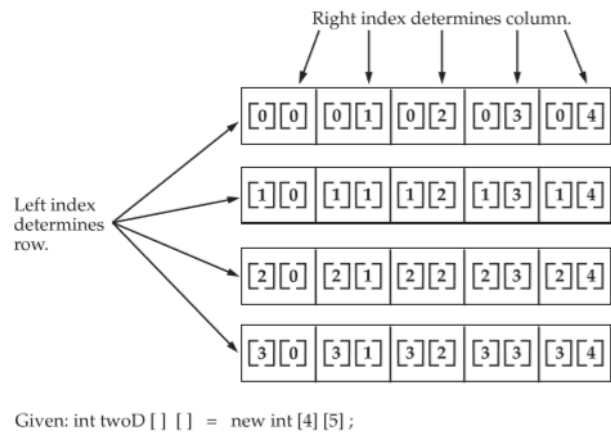
```
class Testarray3  
{  
    public static void main(String args[])  
    {  
        //declaring and initializing 2D array  
        int arr[][]={{1,2,3},{2,4,5},{4,4,5}};  
  
        //printing 2D array  
        for(int i=0;i<3;i++)  
        {  
            for(int j=0;j<3;j++)  
            {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

Output:

```
1 2 3  
2 4 5  
4 4 5
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. We can allocate the remaining dimensions separately.

```
int twoD[][] = new int[4][];  
twoD[0] = new int[5];  
twoD[1] = new int[5];  
twoD[2] = new int[5];  
twoD[3] = new int[5];
```



- In Java we are allowed to allocate different number of columns in the rows. In that case it is known as **Jagged arrays**. Jagged arrays are useful when dealing with a group of arrays of different size.

```
int x[][] = new int[2][];
x[0] = new int[2];
x[1] = new int[3];
```

Class name of Java array:

- In java, **array is an object**. For array object, an proxy class is created whose name can be obtained by **getClass().getName()** method on the object.

```
class First
{
    public static void main(String args[])
    {
        int arr[][]={4,4,5};
        System.out.println(arr.getClass().getName());
    }
}
```

Output:

[I ([is for single dimensional array and I is for integer)

5.12 Taking Input from keyboard: In Java, there are different ways for reading input from the user:

1. **Command Line Argument:** The **command line argument is the argument passed to a program at the time when you run it**. To access the **command-line argument** inside a java program is quite easy, they **are stored as string in String array passed to the args parameter of main()** method.

Example:

```
class cmd
{
    public static void main(String[] args)
    {
        System.out.println("No. of arguments:" + args.length);
        for(int i=0;i< args.length;i++)
        {
            System.out.println(args[i]);
        }
    }
}
```

Output:

```
To run: java cmd 10 20 30
No. of arguments: 3
10
20
30
```

Example:

```
public class Echo {
    public static void main (String[] args) {
        for (String s: args) {
            System.out.println(s);
        }
    }
}
```

```
}  
}
```

Output:

To run: java Echo Silicon Institute of Technology
Silicon
Institute
of
Technology

2. **Scanner Class:** The main purpose of the Scanner class is to parse primitive types and strings using regular expressions. The Java **Scanner class breaks the input into tokens using a delimiter that is whitespace by default**. It provides many methods to read and parse various primitive values. Convenient methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input.

The following is a list of commonly used Scanner class methods:

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.
public double nextDouble()	it scans the next token as a double value.

Example:

```
import java.util.Scanner;  
class First  
{  
    public static void main(String args[])  
    {  
        // Using Scanner for Getting Input from User  
        Scanner sc = new Scanner(System.in);  
  
        String s = sc.nextLine();  
        System.out.println("You entered string "+s);  
  
        int a = sc.nextInt();  
        System.out.println("You entered integer "+a);  
  
        float b = sc.nextFloat();  
        System.out.println("You entered float "+b);  
    }  
}
```

Input:

Silicon
5
13.4

Output:

You entered integer Silicon
You entered integer 5
You entered float 13.4

Example:

```
import java.util.Scanner;
class ScannerTest
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter your rollno");
        int rollno=sc.nextInt();
        System.out.println("Enter your name");
        String name=sc.next();
        System.out.println("Enter your fee");
        double fee=sc.nextDouble();
        System.out.println("Rollno:"+rollno+"    name:"+name+"
                           fee:"+fee);
    }
}
```

Output:

Enter your rollno
111
Enter your name
Ram
Enter your fee
450000
Rollno:111 name:Ram fee:450000

6. INTRODUCTION TO CLASS AND OBJECTS

6.1 Naming Conventions: By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. **Readability of Java program is very important.** It indicates that less time is spent to figure out what the code does. However, it is not forced to follow. So, it is known as convention not rule.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	Should be in uppercase letter. E.g. RED, YELLOW, MAX_PRIORITY etc.

CamelCase in java naming conventions:

- Java follows camelcase syntax for naming the class, interface, method and variable.
- If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

6.2 Class:

- Class defines a new data type. Once a class is defined, it can be used to create objects of that type.
- A class is a template for an object and object is an instance of a class. It is a logical entity.
- While defining a class we declare its form and nature by specifying the data it contains and code that operates on that data.
- A class is defined by use of the **class** keyword
- The general form of a class definition is:

```
class classname{
    type instance-variable1;
    type instance-variable2;
    . . .
    type instance-variablen;

    type methodname1(parameter-list){
        //body of method
    }
    type methodname2(parameter-list){
        //body of method
    }
    . . .
    type methodnamen(parameter-list){
        //body of method
    }
}
```

- The data or variables defined within a class but outside the method are called **instance variables**. Instance variable doesn't get memory at compile time. It gets memory at run time when object (instance) is created.
- A **method** is like function used to expose behavior of an object. The code is contained within methods. The main advantage of the method is **code reusability**.
- Both variables and methods within a class are called **members of the class**.
- Each instance (object) of the class contains its own copy of these variables. Thus the data of one object is separate and unique from the data for other.
- Example:

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

- A class declaration only creates a template, it does not create and actual object.
- To create an object of Box class use the following code:
Box mybox = new Box();
- Every Box object will contain its own copies of the instance variables width, height and depth of each instance variable defined by the class.
- To access the instance variables and methods within an object we need the **dot(.) operator**.
Example: mybox.height

- We can put each class in separate files.

Program:

```
class Box
{
    double width;
    double height;
    double depth;
}
class BoxDemo
{
    public static void main(String args[])
    {
        Box mybox = new Box();
        double vol;

        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " +vol);
    }
}
```

Output:

volumeis3000.0

- Changes to the instance variables of one object have no effect on the instance variable of another.

Program:

```
class Box{
    double width;
    double height;
    double depth;
}
class BoxDemo{
    public static void main(String args[]){
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        vol = mybox1.width * mybox1.height * mybox1.depth;
```

```

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

        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);

    }
}

```

Output:

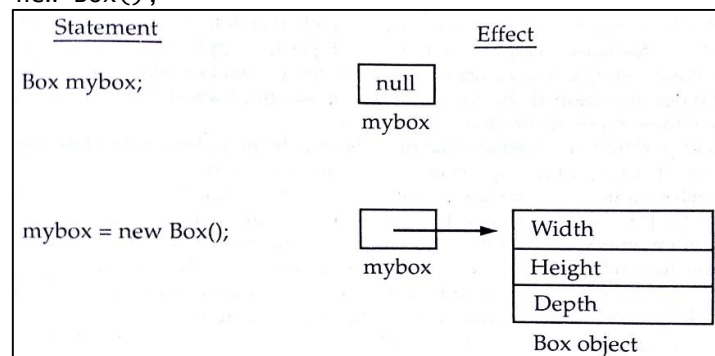
```

volume is 3000.0
volume is 162.0

```

1.1 Object:

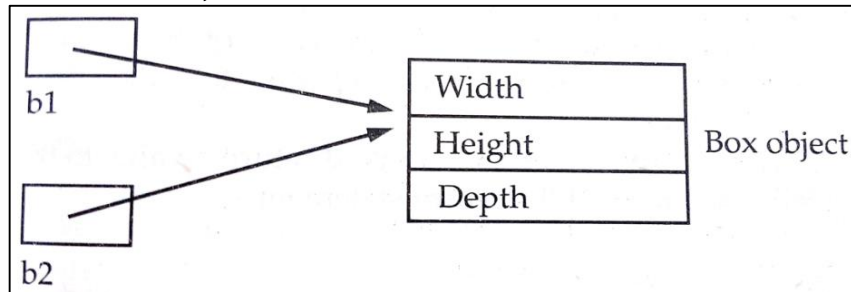
- Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance (result) of a class.
- An object has three characteristics:
 - **state**: represents data (value) of an object.
 - **behavior**: represents the behavior (functionality) of an object such as deposit, withdraw etc.
 - **identity**: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.
- **Example**: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.
- Object Definitions:
 - Object is a real world entity.
 - Object is a run time entity.
 - Object is an entity which has state and behavior.
 - Object is an instance of a class.
- We can create object of a class in the following steps:
 - Declare a variable of the class type.
Box mybox;
 - Acquire an actual, physical copy of the object and assign it to that variable.
mybox = new Box();



- The **new** operator dynamically (at run time) allocates memory for an object and returns a reference (address) to it. All objects get memory in Heap memory area.
- Any attempt to use mybox before assigning any reference of object to it results a compile-time error.
- In the above example Box() is a constructor of classBox. Any class name followed by parentheses specifies the **constructor** of the class.
- A constructor defines what occurs when an object is created.
- In a class if no explicit constructor is defined then Java automatically supplies a default constructor. Here Box() is the default constructor.
- A class creates a new data type that can be used to create objects.

- A class creates a logical framework that defines the relationship between its members.
- When we declare an object of a class we are creating an instance of that class.
- Thus, a **class is a logical construct** and an **object has physical reality**.
- When we assign one object reference variable to another object reference variable, **Java does not create a copy of the object, it only make a copy of the reference**. Thus any changes to the object affect both object reference variables.

Example: `Box b1 = new Box();`
 `Box b2 = b1;`



1.2 Methods:

- A **method is a function that is written in a class**. We **do not have functions in Java**. Whenever a function is written in Java it should be written inside the class only.
- The general form of a method is as follows:

```

type name (parameter-list)
{
    //body of method
}

```
- Type indicates the return type of the method. It can be **primitive or non-primitive type**. In case of a method does not return a value, its return type must be **void**.
- Name indicates the name of the method.
- Parameter-list is a sequence of type and identifier pairs separated by commas.

Program:

```

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

    //display volume of a box
    void volume()
    {
        System.out.println(" volume is "+ (width *height *depth));
    }
}

class BoxDemo
{
    public static void main(String args[])
    {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
    }
}
  
```



```

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        mybox1.volume();
        mybox2.volume();
    }
}

```

Output:

```

Volume is 3000.0
Volume is 162.0

```

- When a **method uses an instance variable that is defined by its class, it does so directly**, without preceding them with an object name or the dot operator. The reason for this is a **method is always invoked relative to some object of its class and once this invocation has occurred, the object is known**.
- A method can return a value. In that case we need to take care of the following:
 - a) The type of data return by a method must be compatible with the return type specified by the method.
 - b) The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Program:

```

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

    //display volume of a box
    double volume(){
        return width *height *depth;
    }
}

class BoxDemo{
    public static void main(String args[]){
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        vol = mybox1.volume();
        System.out.println(" Volume is " + vol);
        vol = mybox2.volume();
    }
}

```

```

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

```

Output:

Volume is 3000.0

Volume is 162.0

- In well-designed Java programs, instance variable should be accessed only through methods defined by their class. In that way we can change the behavior of a method if required easily.

Program:

```

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

    //returns volume of a box
    double volume()
    {
        return width *height *depth;
    }
    void setDim(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
}
class BoxDemo{
    public static void main(String args[]){
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

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

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

```

Output:

Volume is 3000.0

Volume is 162.0

1.3 Constructors:

- Java allows objects to initialize themselves when they are created through the use of a constructor.
- A constructor initializes the instance variables immediately upon the creation of objects.
- A constructor has the following characteristics:
 - a) It has the same name as the class in which it resides and is syntactically similar to a method.
 - b) A constructor does not return any value, not even void. This is because the 'implicit return type of a class' constructor is the class type itself.
 - c) Once we define a constructor, during the object creation, it is automatically called and finishes its execution before the new operator completes its work.
 - d) A constructor is called and executed only once per each object creation.

Program:

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

    //Constructor
    Box()
    {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    //returns volume of a box
    double volume()
    {
        return width *height *depth;
    }
}

class BoxDemo
{
    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);

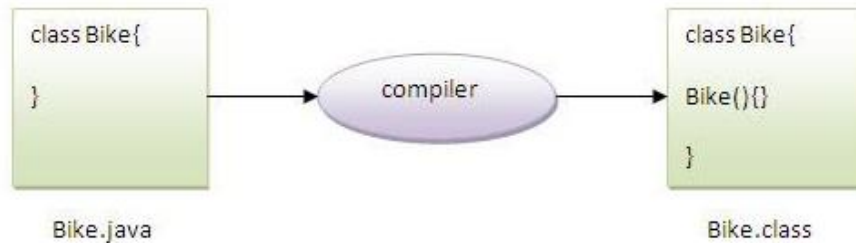
        vol = mybox2.volume();
        System.out.println(" Volume is " + vol);
    }
}
```

```
}
```

Output:

```
Constructing Box  
Constructing Box  
Volume is 1000.0  
Volume is 1000.0
```

- In the line `Box mybox1 = new Box();`, `Box()` constructor is called.
- It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.



➤ Types of Constructors:

- a) **Default constructor:** A constructor is called "Default Constructor" when it doesn't have any parameter. The syntax of default constructor is as follows:

```
class-name(){ }
```

In the above code `Box()` is a default constructor.

- Default constructor is used to provide the default values to the object like 0, null etc. depending on the type.
- It is useful to initialize all objects with the same data.
- Once we define our own constructor, the default constructor is no longer used.
- When data is not passed at the time of creating an object, default constructor is called.

- b) **Parameterized Constructor:** A constructor which has a specific number of parameters is called parameterized constructor.

- It is useful to initialize each object with different data.
- When data is passed at the time of creating an object, parameterized constructor is called.

Program:

```
class Box  
{  
    double width;  
    double height;  
    double depth;  
  
    // Parameterized Constructor  
    Box(double w, double h, double d)  
    {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    //display volume of a box
```

```

        double volume()
        {
            return width *height *depth;
        }
    }
    class BoxDemo
    {
        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);

        }
    }

```

Output:

Volume is 3000.0

Volume is 162.0

Difference between constructor and method:

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

1.4 Data Encapsulation:

- One of OOP's important benefits is the encapsulation of data and the code that manipulates the data.
- In Java encapsulation is achieved by using class.
- By creating a class, we are actually creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it.
- We can use the class through its methods without having to worry about the details of its implementation. So class is like a "data engine", no knowledge of what goes inside the engine is required to use the engine through its controls.

Example: Stack

Program:

```

class Stack
{

```

```

int arr[];
int top;
int size;

Stack(int s)
{
    size=s;
    top = -1;
    arr=new int[size];
}

void push(int item)
{
    if(top==size-1)
        System.out.println("Overflow");
    else
        arr[++top]=item;
}

int pop()
{
    if(top<0)
    {
        System.out.println("Underflow");
        return -1;
    }
    else
        return arr[top--];
}
}
class TestStack
{
    public static void main(String args[])
    {
        Stack s1 = new Stack(5);
        Stack s2 = new Stack(10);
        for(int i=1;i<=5;i++)
            s1.push(i);
        for(int i=11;i<=20;i++)
            s2.push(i);

        System.out.print("Elements in s1:");
        for(int i=0;i<5;i++)
            System.out.print(" "+ s1.pop());

        System.out.println("");
        System.out.print("Elements in s2:");
        for(int i=0;i<10;i++)
            System.out.print(" "+ s2.pop());
    }
}

```

}

}

Output:

Elements in s1: 5 4 3 2 1

Elements in s2: 20 19 18 17 16 15 14 13 12 11

1.5 Access Specifiers:

- Encapsulation provides another important attribute: access control.
- Using encapsulation we can control what parts of a program can access which members of a class. By controlling access, we can prevent misuse.
- A class creates a “black box” which may be used, but the inner working of which are not open to tampering.
- How a member can be accessed is determined by the access specifier that modifies its declaration.
- The access specifier in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access specifiers:
 1. **public:** *public* specifiers achieves the highest level of accessibility. Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.
 2. **private:** *private* specifiers achieves the lowest level of accessibility. Private methods and fields can only be accessed within the same class to which the methods and fields belong. Private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier. Using Private Specifier we can achieve encapsulation and hide data from the outside world. The private access specifier cannot be applied to class because if we do that it will not be available to Java compiler. However, inner classes can be private.
 3. **protected:** *protected* members of the class are accessible within the same class and another class of the same package and also accessible in inherited class of another package. The protected access specifier cannot be applied to class and interfaces.
 4. **default (no specifier):** When you don't set access specifier for the element, it will follow the default accessibility level. There is no default specifier keyword. Classes, variables, and methods can be default accessed. Using default specifier we can access class, method, or field which belongs to same package, but not from outside this package.
- The summary of access specifier is given below:

Access Specifier (AS)	within class	within package	outside package by subclass only	outside package
Public (Universal AS)	Y	Y	Y	Y
Private (Class Level AS)	Y	N	N	N
Protected (Derived Level AS)	Y	Y	Y	N
Default (Package Level AS)	Y	Y	N	N

Example: private AS

```

class Hello
{
    private int a=20;
    private void show()
    {
        System.out.println("Hello java");
    }
}
class Demo
{
    public static void main(String args[])
    {
        Hello obj=new Hello();
        System.out.println(obj.a); //Compile Error, you can't access
                                   private data here
        obj.show();//Compile Time Error, you can't access private
        methods here
    }
}

```

Example: public AS

```

class Hello
{
    public int a=20;
    public void show()
    {
        System.out.println("Hello Java");
    }
}
class Demo
{
    public static void main(String args[])
    {
        Hello obj=new Hello();
        System.out.println(obj.a); // No error
        obj.show();//No error
    }
}

```

Output: 20
Hello Java

Example: protected AS

```

// save A.java
package pack1;
public class A
{
    protected void show()
    {
        System.out.println("Hello Java");
    }
}

//save B.java
package pack2;
import pack1.*;
class B extends A
{
    public static void main(String args[])
    {
        B obj = new B();
        obj.show();
    }
}

```



```
}
```

Output: Hello Java

Example: protected AS

```
class AccessDemo
{
    protected int x = 34;
    public void showDemo()
    {
        System.out.println("The variable value is " + x);
    }
}

class ChildAccess extends AccessDemo
{
    // child class which inherits the properties of AccessDemo class
}

public class AccessExample
{
    public static void main(String[] args)
    {
        ChildAccess ca = new ChildAccess();
        ca.showDemo(); // run properly
        ca.x = 45; // run properly
    }
}
```

Example: default AS

```
// save A.java
package pack1;
class A
{
    void show()
    {
        System.out.println("Hello Java");
    }
}

//save B.java
package pack2;
import pack1.*;
class B
{
    public static void main(String args[])
    {
        A obj = new A();//Compile Time Error, can't access
                        //outside the package
        obj.show();    //Compile Time Error, can't access
                        //outside the package
    }
}
```

- To see how access control can be applied to a more practical example, consider the following improved version of the Stack:

Program:

```
class Stack
{
    private int arr[];
    private int top;
    private int size;

    Stack(int s)
```

```

    {
        size=s;
        top = -1;
        arr=new int[size];
    }

    void push(int item)
    {
        if(top==size-1)
            System.out.println("Overflow");
        else
            arr[++top]=item;
    }

    int pop()
    {
        if(top<0)
        {
            System.out.println("Underflow");
            return -1;
        }
        else
            return arr[top--];
    }
}
class TestStack
{
    public static void main(String args[])
    {
        Stack s1 = new Stack(5);
        Stack s2 = new Stack(10);
        for(int i=1;i<=5;i++)
            s1.push(i);
        for(int i=11;i<=20;i++)
            s2.push(i);

        System.out.print("Elements in s1:");
        for(int i=0;i<5;i++)
            System.out.print(" "+ s1.pop());

        System.out.println("");
        System.out.print("Elements in s2:");
        for(int i=0;i<10;i++)
            System.out.print(" "+ s2.pop());

        //These statements are not legal
        // s1.top = 2;
        // s2.arr[3] = 100;

    }
}

```

Output:

Elements in s1: 9 8 7 6 5 4 3 2 1 0

Elements in s2: 19 18 17 16 15 14 13 12 11 10

1.6 static keyword:

- The static keyword in java is used for memory management mainly.

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare variables, methods, blocks and nested class to be static.

1) static instance variable:

- If you declare any variable as static, it is known static variable.
- Static instance variable are global variables which are used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
- When objects of its class are declared, no copy of the static variable is made. All instances of the class share the same static variable.
- The static variable gets memory only once in class area at the time of class loading.
- **Advantage of static variable:** It makes your program memory efficient (i.e it saves memory).

Example:

```
class Student{
    int rollno;
    String name;
    String college="Silicon";
}
```

Suppose there are 1000 students in my college, now all instance data members will get memory each time when object is created. All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

```
class Student{
    int rollno;
    String name;
    static String college ="Silicon";

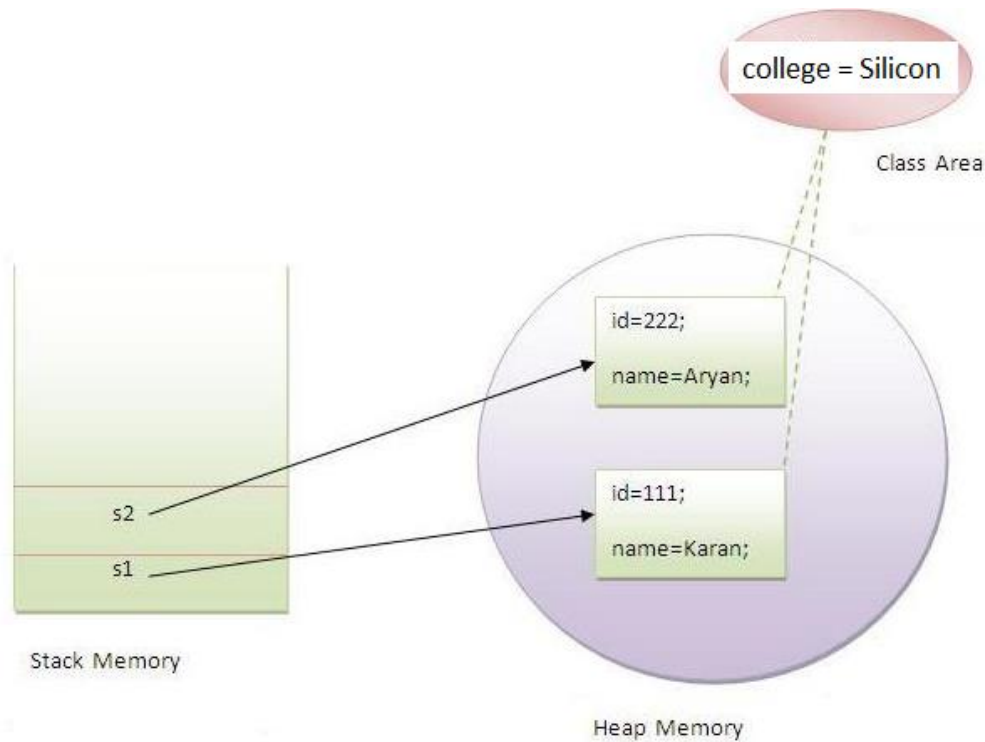
    Student(int r,String n){
        rollno = r;
        name = n;
    }
    void display (){
        System.out.println(rollno+" "+name+" "+college);
    }
}

class StaticTest{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student (222,"Aryan");

        s1.display();
        s2.display();
    }
}
```

Output:

```
111 Karan Silicon
222 Aryan Silicon
```



- static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
class Counter{
    static int count=0;//will get memory only once and retain its
    value
    Counter(){
        count++;
        System.out.println(count);
    }
    public static void main(String args[]){
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
    }
}
```

Output:

```
1
2
3
```

2) 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 object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can only directly access static data member and can change the value of it.
- A static method cannot call non-static method directly it can call static methods only.
- A non-static method can call both static and non-static methods.

Example:

```
class Student{
    int rollno;
```

```

String name;
static String college ="Silicon";

Student(int r,String n){
    rollno = r;
    name = n;
}
static void change(){
    college = "SIT";
}
void display (){
    System.out.println(rollno+" "+name+" "+college);
}
}

class StaticTest{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");

        s1.display();
        s2.display();

        Student.change();

        s1.display();
        s2.display();
    }
}

```

Output:

```

111 Karan Silicon
222 Aryan Silicon
111 Karan SIT
222 Aryan SIT

```

- **Why java main method is static?**

Ans: Because object is not required to call static methods. If we would make main() method as non-static then, JVM would need to create object of that class first to call the main() method which would lead the problem of extra memory allocation.

3) static block:

- A static block is a block of statements declared as static.

```

static{
    statements;
}

```
- **JVM executes the static block before executing main() method.**
- **Static block is used for initializing the static variables. This block gets executed when the class is loaded in the memory.**

Example:

```

class JavaExample
{
    static int num;
}

```

```

static String mystr;
static
{
    num = 97;
    mystr = "Static keyword in Java";
}
public static void main(String args[])
{
    System.out.println("Value of num: "+num);
    System.out.println("Value of mystr: "+mystr);
}
}

```

Output:

Value of num: 97

Value of mystr: Static keyword in Java

- A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Example:

```

class JavaExample2
{
    static int num;
    static String mystr;
    //First Static block
    static
    {
        System.out.println("Static Block 1");
        num = 68;
        mystr = "Block1";
    }
    //Second static block
    static
    {
        System.out.println("Static Block 2");
        num = 98;
        mystr = "Block2";
    }
    public static void main(String args[])
    {
        System.out.println("Value of num: "+num);
        System.out.println("Value of mystr: "+mystr);
    }
}

```

Output:

Static Block 1

Static Block 2

Value of num: 98

Value of mystr: Block2

4) static class:

- A class can be made static only if it is a nested class.
 - Nested static class doesn't need reference of Outer class
 - A static class cannot access non-static members of the Outer class

Example:

```

class JavaExample
{
    private static String str = "BeginnersBook";
}

```

```

//Static class
static class MyNestedClass
{
    //non-static method
    public void disp() {

        /* If you make the str variable of outer class
        non-static then you will get compilation error
        because: a nested static class cannot access non-
        static members of the outer class.
        */
        System.out.println(str);
    }

}

public static void main(String args[])
{
    /* To create instance of nested class we didn't need the outer
    class instance but for a regular nested class you would need
    to create an instance of outer class first*/
    MyNestedClass obj = new MyNestedClass();
    obj.disp();
}
}

```

Output:

BeginnersBook

1.7 this keyword:

- In java, '*this*' is a **reference variable** that **refers to the current object**.
- The '*this*' keyword refers to the current class instance variable.

In this program, we are printing the value of reference variable and this, output of both variables are same.

```

class A5
{
    void m()
    {
        System.out.println(this); //prints same reference ID
    }
    public static void main(String args[])
    {
        A5 obj=new A5();
        System.out.println(obj); //prints the reference ID
        obj.m();
    }
}

```

Output:

A5@15db9742

A5@15db9742

There can be a lot of usage of **java this keyword**. The following are different usage of java this keyword:

1) this: to refer to the current class instance variable:

Let us consider the following program:

```

class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        rollno=rollno;
        name=name;
        fee=fee;
    }
}

```

```

        void display(){
            System.out.println(rollno+" "+name+" "+fee);
        }
    }
    class TestThis1{
        public static void main(String args[]){
            Student s1=new Student(111,"ankit",5000f);
            Student s2=new Student(112,"sumit",6000f);
            s1.display();
            s2.display();
        }
    }
}

```

Output:

```

0 null 0.0
0 null 0.0

```

In the above program, the constructor contains the formal arguments whose names are same as the instance variables. In that case, when we refer to the instance variable the compiler understands it as local variable only. To avoid this we can use this keyword to distinguish local variable and instance variable.

```

        class Student{
            int rollno;
            String name;
            float fee;
            Student(int rollno,String name,float fee){
                this.rollno=rollno;
                this.name=name;
                this.fee=fee;
            }
            void display(){
                System.out.println(rollno+" "+name+" "+fee);
            }
        }
        class TestThis1{
            public static void main(String args[]){
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
            }
        }
    }
}

```

Output:

```

111ankit5000
112sumit6000

```

Example:

```

class Student
{
    String name;
    int markProg,markLogic;
    char gradeLab;
    Student(String s,int mp,int ml,char gl)
    {
        name=new String(s);
        markProg=mp;
        markLogic=ml;
        gradeLab=gl;
    }
}

```



```

        Student isStronger(Student s)
        {
            if(markProg>s.markProg)
                return this;
            else if(markProg==s.markProg && markLogic>s.markLogic)
                return this;
            else if(markProg==s.markProg && markLogic==s.markLogic &&
                    gradeLab<s.gradeLab)
                return this;
            else if(markProg==s.markProg && markLogic==s.markLogic &&
                    gradeLab==s.gradeLab)
                return null;
            else
                return s;
        }
    }
}
class TestStudent
{
    public static void main(String args[])
    {
        Student s1=new Student("Ram",56,40,'A');
        Student s2=new Student("Shyam",56,40,'A');
        if(s1.isStronger(s2)==null)
            System.out.println("Performance of both are equal");
        else if(s1.isStronger(s2)==s1)
            System.out.println("First student is better");
        else
            System.out.println("Second student is better");
    }
}

```

Output:

Performance of both are equal

NOTE: If the name of local variable and instance variable are different, then no need of using this keyword.

2) 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. The following program shows 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);
    }
}

```

Output:

Hello a

10

The following program shows calling parameterized constructor from default constructor:

```
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();
    }
}
```

Output:

5
Hello

The this() constructor call should be **used to reuse the constructor from the constructor**. It maintains the chain between the constructors i.e. it is used for constructor chaining. So the real usage of this() constructor call is explained through the following example:

```
class Student{
    int rollno;
    String name,course;
    float fee;
    Student(int rollno,String name,String course){
        this.rollno=rollno;
        this.name=name;
        this.course=course;
    }
    Student(int rollno,String name,String course,float fee){
        this(rollno,name,course);//reusing constructor
        this.fee=fee;
    }
    void display()
    {
        System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit","java");
        Student s2=new Student(112,"sumit","java",6000f);
        s1.display();
        s2.display();
    }
}
```

Output:

111 ankit java null
112 sumit java 6000

NOTE: Call to `this()` must be the first statement in constructor. Otherwise java shows compile time error.

3) **this: to pass as an argument in the method**

The *this* keyword can also be passed as an argument in the method. In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods. Let's see one example:

```
class S2
{
    void m(S2 obj)
    {
        System.out.println("method is invoked");
    }
    void p()
    {
        m(this);
    }
    public static void main(String args[])
    {
        S2 s1 = new S2();
        s1.p();
    }
}
```

Output:

method is invoked

4) **this: to pass as argument in the constructor call**

We can pass the *this* keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
    A4 obj;
    B(A4 obj){
        this.obj=obj;
    }
    void display(){
        System.out.println(obj.data);//using data member of A4 class
    }
}
class A4{
    int data=10;
    A4(){
        B b=new B(this);
        b.display();
    }
    public static void main(String args[]){
        A4 a=new A4();
    }
}
```

Output:

10

5) **this keyword can be used to return current class instance**

We can return *this* keyword as a statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

```
class A{
    A getA(){
        return this;
    }
}
```

```

    }
    void msg(){System.out.println("Hello Java");}
}
class Test1{
    public static void main(String args[]){
        new A().getA().msg();
    }
}

```

Output:

Hello Java

1.8 Array of Objects:

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object. As we know, an array is a collection of similar type, therefore an array can be a collection of class type.

Syntax:

```

class ClassName
{
    datatype var1;
    datatype var2;
    - - - - -
    datatype varN;

    method1();
    method2();
    - - - - -
    methodN();
}
Class-Name [] object = new Class-Name [size];

```

Example:

```

import java.util.Scanner;
class Employee
{
    int Id;
    String Name;
    int Age;
    long Salary;

    void GetData()           // Defining GetData()
    {
        Scanner sc = new Scanner(System.in);

        System.out.print("\n\tEnter Employee Id : ");
        Id = Integer.parseInt(sc.nextLine());

        System.out.print("\n\tEnter Employee Name : ");
        Name = sc.nextLine();

        System.out.print("\n\tEnter Employee Age : ");
        Age = Integer.parseInt(sc.nextLine());

        System.out.print("\n\tEnter Employee Salary : ");
        Salary = Integer.parseInt(sc.nextLine());

    }
}

```

```

void PutData()           // Defining PutData()
{
    System.out.print("\n\t" + Id + "\t" +Name +
        "\t" +Age + "\t" +Salary);
}

public static void main(String args[])
{
    Employee[] Emp = new Employee[3];
    int i;
    for(i=0;i<3;i++)
        Emp[i] = new Employee();
        // Allocating memory to each object
    for(i=0;i<3;i++)
    {
        System.out.print("\nEnter details of "+ (i+1)
            +" Employee\n");
        Emp[i].GetData();
    }
    System.out.print("\nDetails of Employees\n");
    for(i=0;i<3;i++)
        Emp[i].PutData();
}
}

```

Output:

Enter details of 1 Employee
 Enter Employee Id : 101
 Enter Employee Name : Suresh
 Enter Employee Age : 29
 Enter Employee Salary : 45000

Enter details of 2 Employee
 Enter Employee Id : 102
 Enter Employee Name : Mukesh
 Enter Employee Age : 31
 Enter Employee Salary : 51000

Enter details of 3 Employee
 Enter Employee Id : 103
 Enter Employee Name : Ramesh
 Enter Employee Age : 28
 Enter Employee Salary : 47000

Details of Employees

101	Suresh	29	45000
102	Mukesh	31	51000
103	Ramesh	28	47000

NOTE: In the above example an array of objects is created just like an array of primitive type data items in the following way.

```
Employee[] Emp = new Employee[3];
```

The above statement creates the array which can hold references to three Employee objects. It doesn't create the Employee objects themselves. They have to be created separately using the constructor of the Student class. The following code does that:

```
for(i=0;i<3;i++)
    Emp[i] = new Employee();
```

Example:

```
class Student
{
    public int roll_no;
    public String name;
    Student(int roll_no, String name)
    {
        this.roll_no = roll_no;
        this.name = name;
    }
}

// Elements of array are objects of a class Student.
public class GFG
{
    public static void main (String[] args)
    {
        // declares an Array of integers.
        Student[] arr;

        // allocating memory for 5 objects of type Student.
        arr = new Student[5];

        // initialize the first elements of the array
        arr[0] = new Student(1,"aman");

        // initialize the second elements of the array
        arr[1] = new Student(2,"vaibhav");

        // so on...
        arr[2] = new Student(3,"shikar");
        arr[3] = new Student(4,"dharmesh");
        arr[4] = new Student(5,"mohit");

        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at " + i + " : " +
                               arr[i].roll_no + " " + arr[i].name);
    }
}
```

Output:

```
Element at 0 : 1 aman
Element at 1 : 2 vaibhav
Element at 2 : 3 shikar
```

Element at 3 : 4 dharmesh

Element at 4 : 5 mohit