Paper: **Object Oriented Programming using Java**

Code: CS504A
Contacts: 3L
Credits: 3
Total Lectures: 36

➢ Objective(s)

- It allows to map with real world Object (Object orientation)rather than action(Procedure) that comes to produce softwares as seperated code modules which rise up decoupling and increases code re-usability.
- It demonostrates that how can you change the implementation of an object without affecting any other code by increasing data security and protecting unwanted data access. (Encapsulation).
- It allows you to have many different functions, all with the same name, all doing the same job, but depending upon different data. (Polymorphism).
- It guides you to write generic code: which will work with a range of data, so you don't have to write basic stuff over, and over again. (Generics).
- It lets you write a set of functions, then expand them in different direction without changing or copying them in any way. (Inheritance)

➢ Outcome(s)

- Design the process of interaction between Objects, classes & methods w.r.t. Object Oriented Programming.
- Acquire a basic knowledge of Object Orientation with different properties as well as different features of Java.
- Analyze various activities of different string handling functions with various I/O operations.
- Discuss basic Code Reusability concept w.r.t. Inheritance, Package and Interface.
- Implement Exception handling, Multithreading and Applet (Web program in java) programming concept in Java.

➢ Prerequisites:

1. Computer Fundamentals
2. Basic understanding of Computer Programming and related Programming Paradigms
3. Problem Solving Techniques with proper logic Implementation.
4. Basic Computer memory architecture with data accession.

➢ Proposed Syllabus:

Paper: Object Oriented Programming using Java
Code: CS504A
Contacts: 3L
Credits: 3
Total Lectures: 36

Module 1: [5L]

Introduction:
Object Oriented Analysis & Design-Concepts of object oriented programming language, Object, Class.**[1L];** Relationships among objects and classes-Generalization, Specialization, Aggregation, Association, Composition, links, Meta-class. **[1L] ;** Object Oriented Programming concepts - Difference between OOP and other conventional programming – advantages and disadvantages. Class, object, Method. **[1L];** Properties of OOP- message passing, inheritance, encapsulation, polymorphism, Data abstraction. **[1L];** Difference between different OOPs Languages. **[1L].**

Module 2: [9L]

Java Basics:
Basic concepts of java programming - Advantages of java, Byte-code & JVM, Data types, Different types of Variables. **[1L] ;**Access specifiers, Operators, Control statements & loops. **[1L];** Array. **[1L] ;**Creation of class, object, method. **[1L];** Constructor- Definition, Usage of Constructor, Different types of Constructor. **[1L];** finalize method and garbage collection, Method & Constructor overloading. **[1L];** this keyword, use of objects as parameter & methods returning objects. **[1L];** Call by value & call by reference. **[1L];** Static variables & methods. Nested & inner classes. **[1L].**

Module 3:[4L]

Basic String handling & I/O :
Basic string handling concepts- Concept of mutable and immutable string, Methods of String class-charAt(), compareTo(), equals(), equalsIgnoreCase(), indexOf(), length() , substring(). **[1L];** toCharArray(), toLowerCase(), toString(), toUpperCase() , trim() , valueOf() methods, Methods of String buffer class- append(), capacity(), charAt(), delete(), deleteCharAt(). **[1L];** ensureCapacity(), getChars(), indexOf(), insert(), length(), setCharAt(), setLength(), substring(), toString(). **[1L] ;**Command line arguments, basics of I/O operations – keyboard input using BufferedReader & Scanner classes. **[1L].**

Module 4: [8L]

Inheritance and Java Packages :
Inheritance - Definition, Advantages, Different types of inheritance and their implementation. **[1L] ;** Super and final keywords, super() method. **[1L];** Method overriding, Dynamic method dispatch. **[1L];** Abstract classes & methods. **[1L];** Interface - Definition, Use of Interface. **[1L];** Multiple inheritance by using Interface. **[1L] ;** Java Packages -Definition, Creation of packages. **[1L];** Importing packages, member access for packages. **[1L]**

Module 5: [10L]

Exception handling, Multithreading and Applet Programming :
Exception handling - Basics, different types of exception classes. Difference between Checked & Unchecked Exception. **[1L];** Try & catch related case studies.**[1L];** Throw, throws & finally. **[1L];** Creation of user defined exception. **[1L];** Multithreading - Basics, main thread, thread life cycle.**[1L];** Creation of multiple threads-yield(), suspend(), sleep(n), resume(), wait(), notify(), join(), isAlive().**[1L] ;**Thread priorities, thread synchronization.**[1L];**Interthread communication, deadlocks for threads**[1L];** Applet Programming - Basics, applet life cycle, difference between application & applet programming**[1L];** Parameter passing in applets. **[1L]**

Recommended Books:

Textbooks:

1. Herbert Schildt – "Java: The Complete Reference " – 9th Ed. – TMH
2. E. Balagurusamy – " Programming With Java: A Primer " – 3rd Ed. – TMH.

References:
1. R.K Das – " Core Java for Beginners " – VIKAS PUBLISHING.
2. Rambaugh, James Michael, Blaha – " Object Oriented Modelling and Design " – Prentice Hall, India.

➢ Lesson Plan for B.Tech Computer Science and Engineering Program(Autonomy)

Paper: Object Oriented Programming using Java
Code: CS504A
Contacts: 3L
Credits: 3
Total Lectures: 36

Module No.	Course Content	Lecture Required	Reference / Text Books
1	**Module 1: [5L]** **Introduction:** Object Oriented Analysis & Design - Concepts of object oriented programming language, Object, Class. **[1L];** Relationships among objects and classes-Generalization, Specialization, Aggregation, Association, Composition, links, Meta-class. **[1L];** Object Oriented Programming concepts - Difference between OOP and other conventional programming – advantages and disadvantages. Class, object, Method. **[1L];** Properties of OOP- message passing, inheritance, encapsulation, polymorphism, Data abstraction. **[1L];** Difference between different OOPs Languages. **[1L]**	5L	**Textbooks:** 1. Herbert Schildt – " Java: The Complete Reference " – 9th Ed. – TMH 2. E. Balagurusamy – " Programming With Java: A Primer " – 3rd Ed. – TMH. **References:** 1. R.K Das – " Core Java for Beginners " – VIKAS PUBLISHING. 2. Rambaugh, James Michael, Blaha – " Object Oriented Modelling and Design " – Prentice Hall, India.
2	**Module 2: [9L]** **Java Basics:** Basic concepts of java programming - Advantages of java, Byte-code & JVM, Data types, Different types of Variables. **[1L];** Access specifiers, Operators, Control statements & loops. **[1L];** Array. **[1L];** Creation of class, object, method. **[1L];** Constructor-Definition, Usage of Constructor, Different types of Constructor. **[1L];** finalize method and garbage collection, Method & Constructor overloading. **[1L];** this keyword, use of objects as	9L	**Textbooks:** 1. Herbert Schildt – " Java: The Complete Reference " – 9th Ed. – TMH 2. E. Balagurusamy – " Programming With Java: A Primer " – 3rd Ed. – TMH. **References:** 1. R.K Das – " Core Java for Beginners " – VIKAS PUBLISHING. 2. Rambaugh, James

	parameter & methods returning objects. **[1L];** Call by value & call by reference. **[1L];** Static variables & methods. Nested & inner classes. **[1L]**		Michael, Blaha – " Object Oriented Modelling and Design " – Prentice Hall, India.
3	**Module 3:[4L]** **Basic String handling & I/O:** Basic string handling concepts- Concept of mutable and immutable string, Methods of String class - charAt(), compareTo(), equals(), equalsIgnoreCase(), indexOf(), length(), substring(). **[1L];** toCharArray(), toLowerCase(), toString(), toUpperCase(), trim(), valueOf() methods, Methods of String buffer class- append(), capacity(), charAt(), delete(), deleteCharAt(). **[1L];** ensureCapacity(), getChars(), indexOf(), insert(), length(), setCharAt(), setLength(), substring(), toString(). **[1L];** Command line arguments, basics of I/O operations – keyboard input using BufferedReader & Scanner classes. **[1L]**	4L	**Textbooks:** 1. Herbert Schildt – " Java: The Complete Reference " – 9th Ed. – TMH 2. E. Balagurusamy – " Programming With Java: A Primer " – 3rd Ed. – TMH. **References:** 1. R.K Das – " Core Java for Beginners " – VIKAS PUBLISHING. 2. Rambaugh, James Michael, Blaha – " Object Oriented Modelling and Design " – Prentice Hall, India.
4	**Module 4: [8L]** **Inheritance & Java Packages:** Inheritance - Definition, Advantages, Different types of inheritance and their implementation. **[1L];** Super and final keywords, super() method. **[1L];** Method overriding, Dynamic method dispatch. **[1L];** Abstract classes & methods. **[1L];** Interface - Definition, Use of Interface. **[1L];** Multiple inheritance by using Interface. **[1L];** Packages -Definition, Creation of packages. **[1L];** Importing packages, member access for packages. **[1L]**	8L	**Textbooks:** 1. Herbert Schildt – " Java: The Complete Reference " – 9th Ed. – TMH 2. E. Balagurusamy – " Programming With Java: A Primer " – 3rd Ed. – TMH. **References:** 1. R.K Das – " Core Java for Beginners " – VIKAS PUBLISHING. 2. Rambaugh, James Michael, Blaha – " Object Oriented Modelling and Design " – Prentice Hall, India.

5	**<u>Module 5:</u> [10L]** **Exception Handling, Multithreading & Applet Programming:** Exception handling - Basics, different types of exception classes. Difference between Checked & Unchecked Exception. **[1L];** Try & catch related case studies. **[1L];** Throw, throws & finally. **[1L];** Creation of user defined exception. **[1L];** Multithreading - Basics, main thread, thread life cycle. **[1L];** Creation of multiple threads - yield(), suspend(), sleep(n), resume(), wait(), notify(), join(), isAlive(). **[1L];** Thread priorities, thread synchronization. **[1L];** Interthread communication, deadlocks for threads. **[1L];** Applet Programming - Basics, applet life cycle, difference between application & applet programming. **[1L];** Parameter passing in applets. **[1L]**	10L	**Textbooks:** 1. Herbert Schildt – " Java: The Complete Reference " – 9th Ed. – TMH 2. E. Balagurusamy – " Programming With Java: A Primer " – 3rd Ed. – TMH. **References:** 1. R.K Das – " Core Java for Beginners " – VIKAS PUBLISHING. 2. Rambaugh, James Michael, Blaha – " Object Oriented Modelling and Design " – Prentice Hall, India.

MODULE 1: INTRODUCTION

Lecture 1: Object Oriented Analysis & Design

1.1. Object-Oriented Analysis:

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, "*Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain*".

The primary tasks in object-oriented analysis (OOA) are –

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact
- The common models used in OOA are use cases and object models.

1.2. Object-Oriented Design:

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology–independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include –

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and

- Implementation of associations.

Grady Booch has defined object-oriented design as "*a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design*".

1.3. Object-Oriented Programming:

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are –

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object–oriented programming as "*a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships*".

1.4. Objects and Classes:

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object–oriented paradigm.

1.4.1. Object:

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has –

- *Identity* that distinguishes it from other objects in the system.

- *State* that determines the characteristic properties of an object as well as the values of the properties that the object holds.

- *Behavior* that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

1.4.2. Class:

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are –

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.

- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

1.4.3. Example:

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two-dimensional space. The attributes of this class can be identified as follows –

- x-coord, to denote x-coordinate of the center
- y-coord, to denote y-coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows –

- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

Lecture 2: Relationships among objects and classes

2.1. Generalization:

Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods.

In the Figure 2.1, the classes Piece of Luggage and Piece of Cargo partially share the same attributes. From a domain perspective, the two classes are also very similar. During generalization, the shared characteristics are combined and used to create a new superclass Freight. Piece of Luggage and Piece of Cargo become subclasses of the class Freight.
The shared attributes are only listed in the superclass, but also apply to the two subclasses, even though they are not listed there.

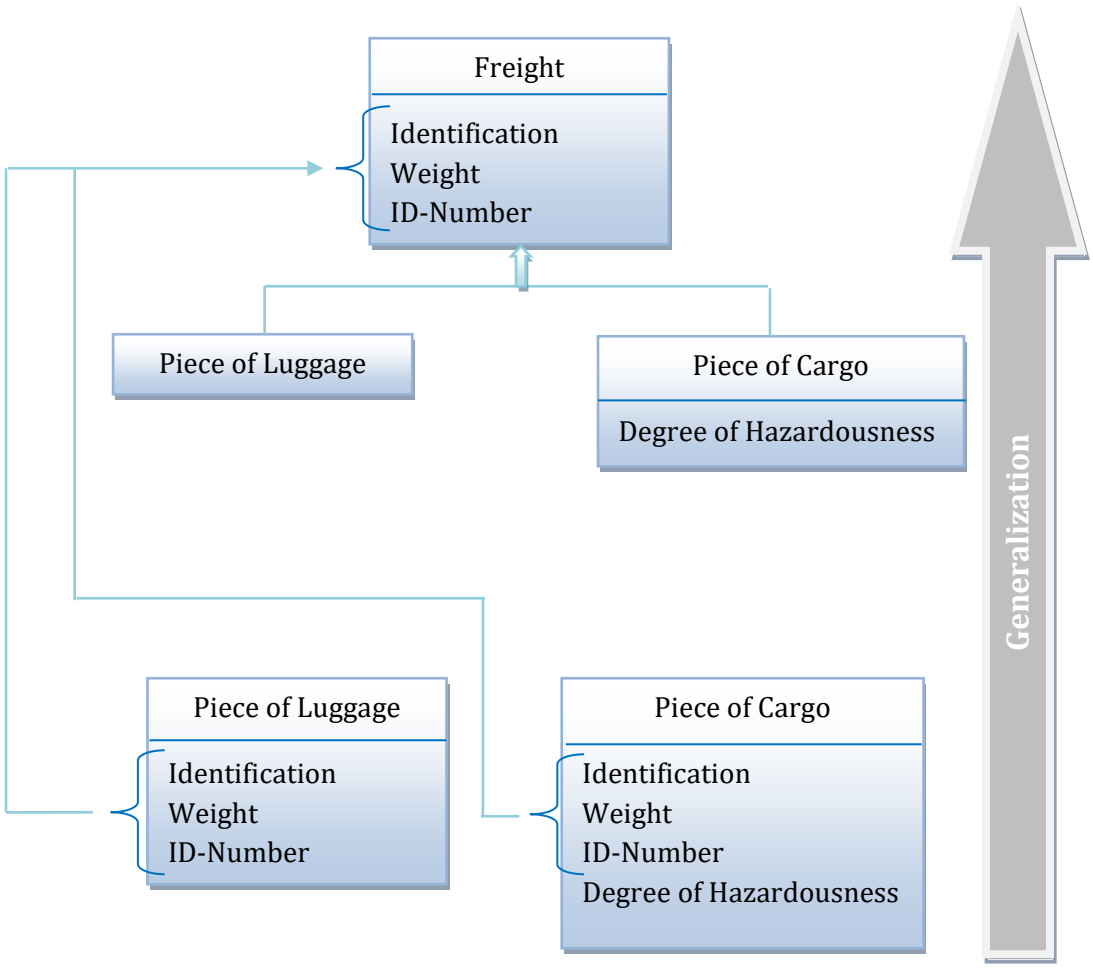


Figure 2.1: Example of generalization

2.2. Specialization:

In contrast to generalization, specialization means creating new subclasses from an existing class. If it turns out that certain attributes, associations, or methods only apply to some of the objects of the class, a subclass can be created. The most inclusive class in a generalization/specialization is called the superclass and is generally located at the top of the diagram. The more specific classes are called subclasses and are generally placed below the superclass.

In the Figure 2.2, the class Freight has the attribute Degree of Hazardousness, which is needed only for cargo, but not for passenger luggage. Additionally (not visible in Figure 2.2), only passenger luggage has a connection to a coupon. Obviously, here two similar but different domain concepts are combined into one class. Through specialization the two special cases of freights are formed: Piece of Cargo and Piece of Luggage. The attribute Degree of Hazardousness is placed where it belongs – inPiece of Cargo. The attributes of the class Freight also apply to the two subclasses Piece of Cargo and Piece of Luggage.

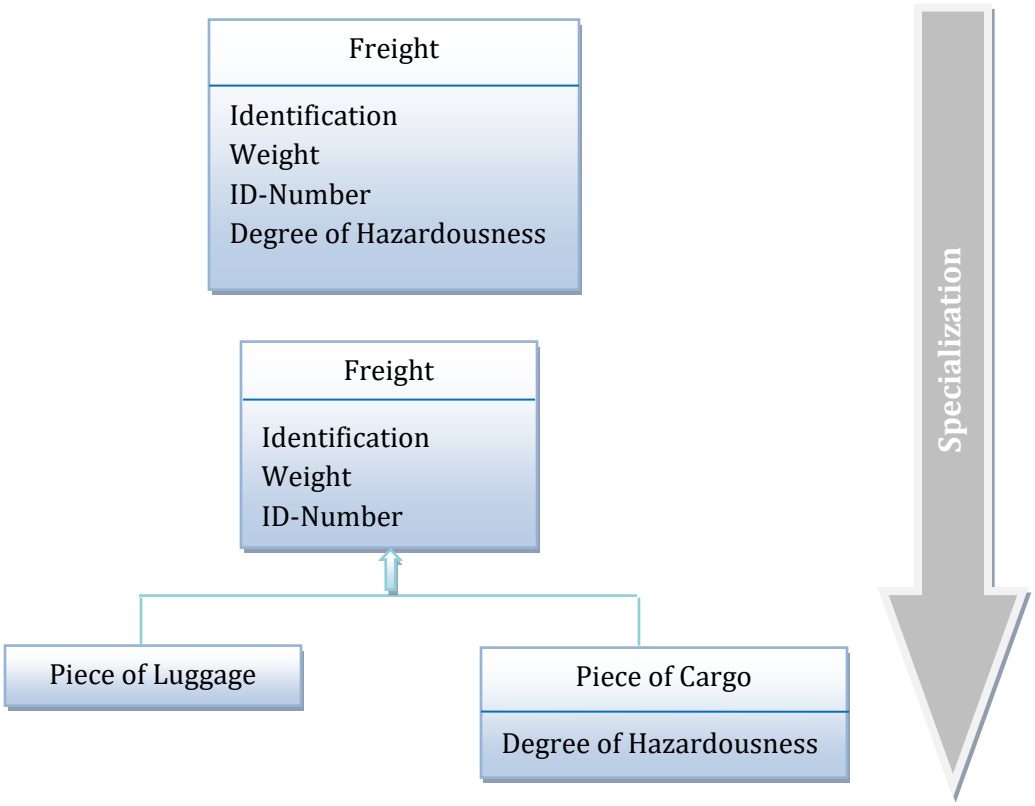
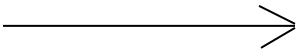


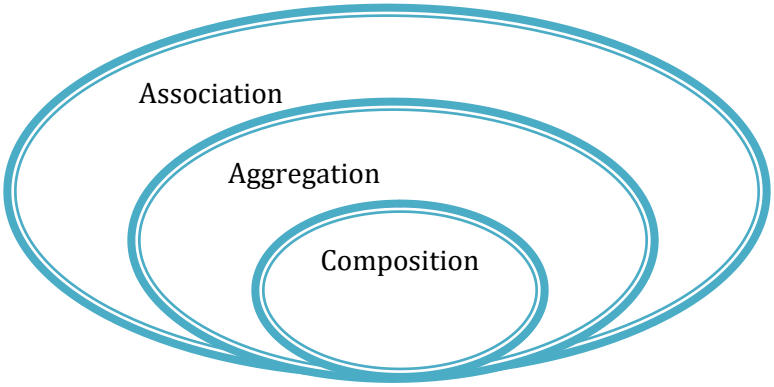
Figure 2.2: Example of specialization

2.3. Association:

Association is a “has-a” type relationship. Association is relation between two separate classes which establishes through their Objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. For example if we have two classes, then these two classes are said to be in a “has-a” relationship if both of these entities share each other’s object for some work and at the same time they can exist without each other’s dependency or both have their own life time. Aggregation is a special form of association. Composition is a special form of aggregation. Association is indicated using



Example: A Student and a Faculty are having an association.



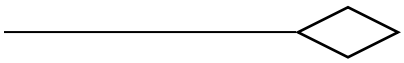
2.4. Aggregation:

Aggregation in Java is a relationship between two classes that is best described as a "has-a" and "whole/part" relationship. It is a more specialized version of the association relationship. The aggregate class contains a reference to another class and is said to have ownership of that class. Each class referenced is considered to be *part-of* the aggregate class.

Ownership occurs because there can be no cyclic references in an aggregation relationship.

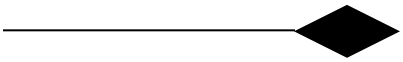
If Class A contains a reference to Class B and Class B contains a reference to Class A then no clear ownership can be determined and the relationship is simply one of association.

For example, if you imagine that a Student class that stores information about individual students at a school. Now assume a Subject class that holds the details about a particular subject (e.g., history, geography). If the Student class is defined to contain a Subject object then it can be said that the Student object has-a Subject object. The Subject object also makes up part-of the Student object – after all, there is no student without a subject to study. The Student object, therefore, owns the Subject object.Aggregation is indicated using



2.5. Composition:

Composition is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then it's a composition. Heart class has no sense if Human class is not present. Composition is indicated using



2.6. Links:

In object modeling links provides a relationship between the objects. These objects or instance may be same or different in data structure and behavior. Therefore a link is a physical or conceptual connection between instances (or objects).

For example: Ram works for HCL company. In this example "works for" is the link between "Ram" and "HCL Company". Links are relationship among the objects (instance).

2.7. Meta-class:

In object-oriented programming, a meta-class is a class whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a meta-class defines the behavior of certain classes and their instances. Not all object-oriented programming languages support meta-classes. Among those that do, the extent to which meta-classes can override any given aspect of class behavior varies. Meta-classes can be implemented by having classes be first-class citizen, in which case a meta-class is simply an object that constructs

classes. Each language has its own meta-object protocol, a set of rules that govern how objects, classes, and meta-classes interact.

Lecture 3: Object Oriented Programming concepts

3.1. Why OOP?

Suppose that you want to assemble your own PC, you go to a hardware store and pick up a motherboard, a processor, some RAMs, a hard disk, a casing, a power supply, and put them together. You turn on the power, and the PC runs. You need not worry whether the CPU is 1-core or 6-core; the motherboard is a 4-layer or 6-layer; the hard disk has 4 plates or 6 plates, 3 inches or 5 inches in diameter; the RAM is made in Japan or Korea, and so on. You simply put the hardware components together and expect the machine to run. Of course, you have to make sure that you have the correct interfaces, i.e., you pick an IDE hard disk rather than a SCSI hard disk, if your motherboard supports only IDE; you have to select RAMs with the correct speed rating, and so on. Nevertheless, it is not difficult to set up a machine from hardware components.

Similarly, a car is assembled from parts and components, such as chassis, doors, engine, wheels, brake and transmission. The components are reusable, e.g., a wheel can be used in many cars (of the same specifications).

Hardware, such as computers and cars, are assembled from parts, which are reusable hardware components.

How about software? Can you "assemble" a software application by picking a routine here, a routine there, and expect the program to run? The answer is obviously NO! Unlike hardware, it is very difficult to "assemble" an application from software components. Since the advent of computer 70 years ago, we have written tons and tons of programs and routines. However, for each new application, we have to re-invent the wheels and write the program from scratch!

Why re-invent the wheels? Why re-writing codes? Can you write better codes than those codes written by the experts?

3.2. Traditional Procedural-Oriented languages:

Traditional procedural-oriented programming languages (such as C, Fortran, Cobol and Pascal) suffer some notable drawbacks in creating reusable software components:

1. The procedural-oriented programs are made up of functions. Functions are less reusable. It is very difficult to copy a function from one program and reuse in another program because the function is likely to reference the global variables and other functions. In other words, functions are not well-encapsulated as a self-contained reusable unit.
2. The procedural languages are not suitable of high-level abstraction for solving real life problems. For example, C programs uses constructs such as if-else, for-loop, array, method, pointer, which are low-level and hard to abstract real problems such as a Customer Relationship Management (CRM) system or a computer soccer game.

The traditional procedural-languages separate the data structures (variables) and algorithms (functions).

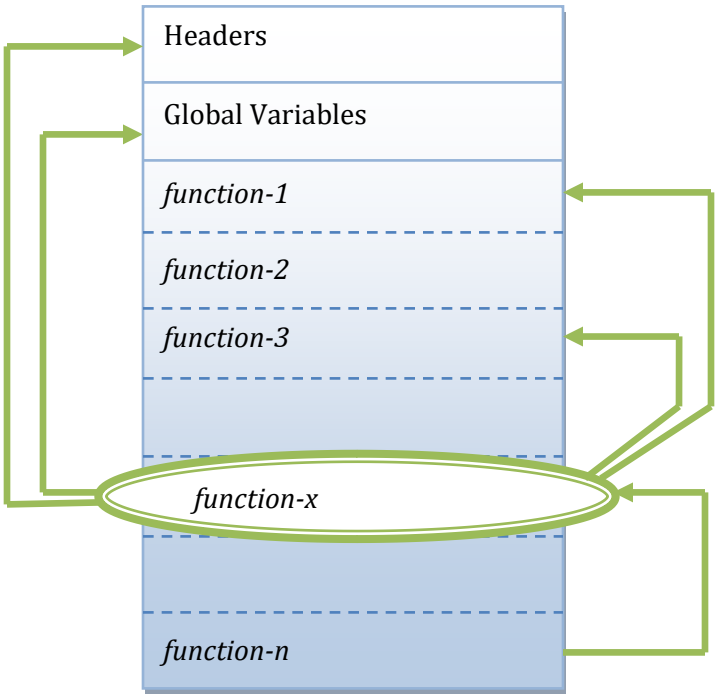


Figure 3.2: A function (in C) is not well-encapsulated

3.3. Object-Oriented Programming Languages:

Object-oriented programming (OOP) languages are designed to overcome these problems.

1. The basic unit of OOP is a class, which encapsulates both the static properties and dynamic operations within a "box", and specifies the public interface for using these boxes. Since classes are well-encapsulated, it is easier to reuse these classes. In other words, OOP combines the data structures and algorithms of a software entity inside the same box.
2. OOP languages permit higher level of abstraction for solving real-life problems. The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve. The OOP languages (such as Java, C++ and C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.

As an example, suppose you wish to write a computer soccer games (which I consider as a complex application). It is quite difficult to model the game in procedural-oriented languages. But using OOP languages, you can easily model the program accordingly to the "real things" appear in the soccer games.

- Player: attributes include name, number, location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
- Ball:
- Reference:
- Field:
- Audience:
- Weather:

Most importantly, some of these classes (such as Ball and Audience) can be reused in another application, e.g., computer basketball game, with little or no modification.

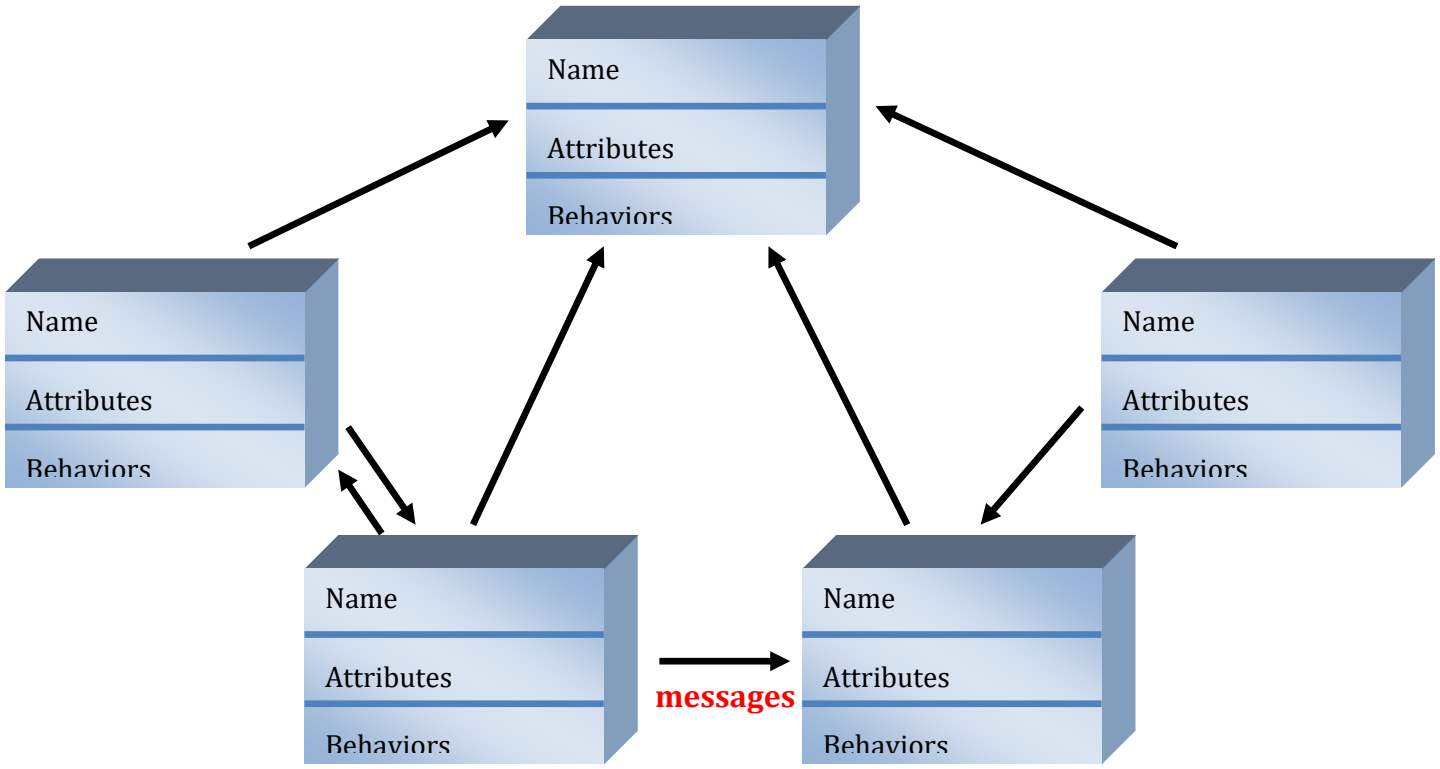


Figure 3.3.1: An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

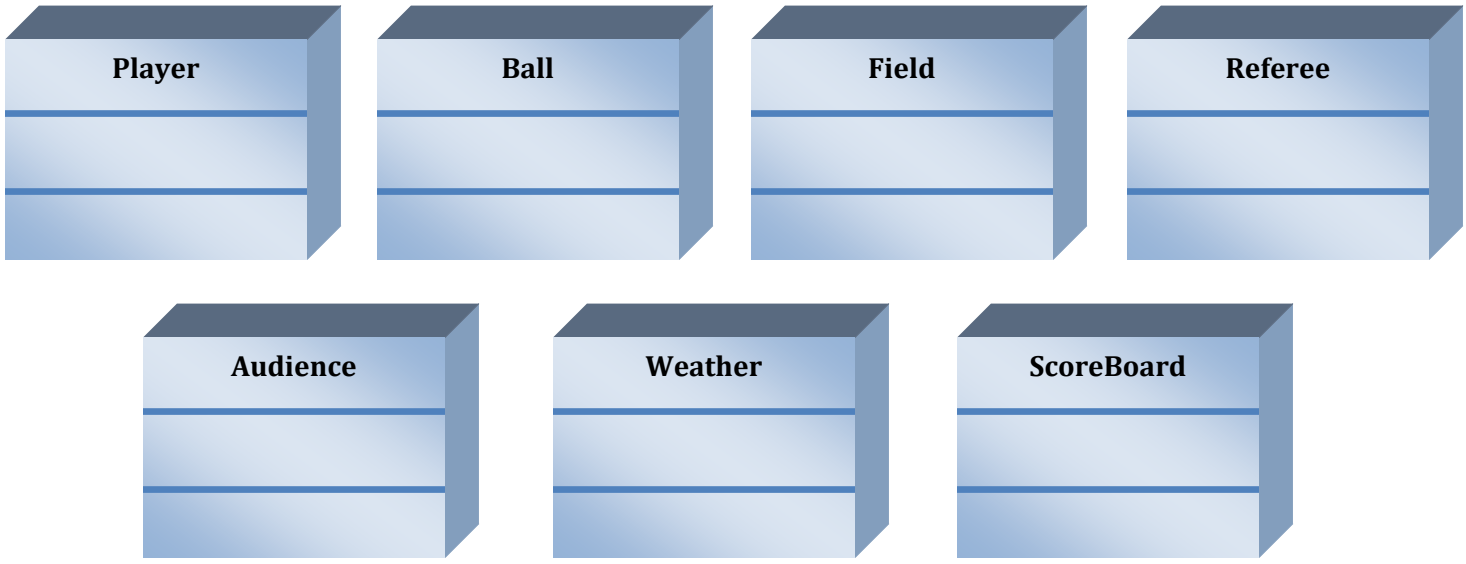


Figure 3.3.2: Classes (Entities) in a Computer Soccer Game

3.3.1. Benefits of OOP:

The procedural-oriented languages focus on procedures, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.

The object-oriented languages focus on components that the user perceives, with objects as the basic unit. You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.

Object-Oriented technology has many benefits:

- *Ease in software design* as you could think in the problem space rather than the machine's bits and bytes. You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
- *Ease in software maintenance:* object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
- *Reusable software:* you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes - fully tested and proven codes.

3.4. Class & Objects:

In Java, a *class* is a definition of objects of the same kind. In other words, a class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind.

An instance is a realization of a particular item of a class. In other words, an instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "Student" and create three instances of the class "Student" for "Peter", "Paul" and "Pauline".

The term "object" usually refers to instance. But it is often used loosely, and may refer to a class or an instance.

A class can be visualized as a three-compartment box, as illustrated:

1. *Name* (or identity): identifies the class.
2. *Variables* (or attribute, state, field): contains the static attributes of the class.

3. *Methods* (or behaviors, function, operation): contains the dynamic behaviors of the class.

In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

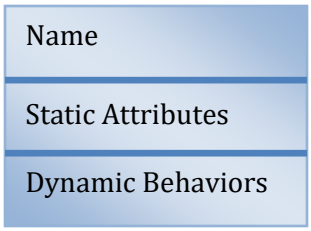


Figure 3.4.1: A class is a 3-compartment box

The followings figure shows a few examples of classes:

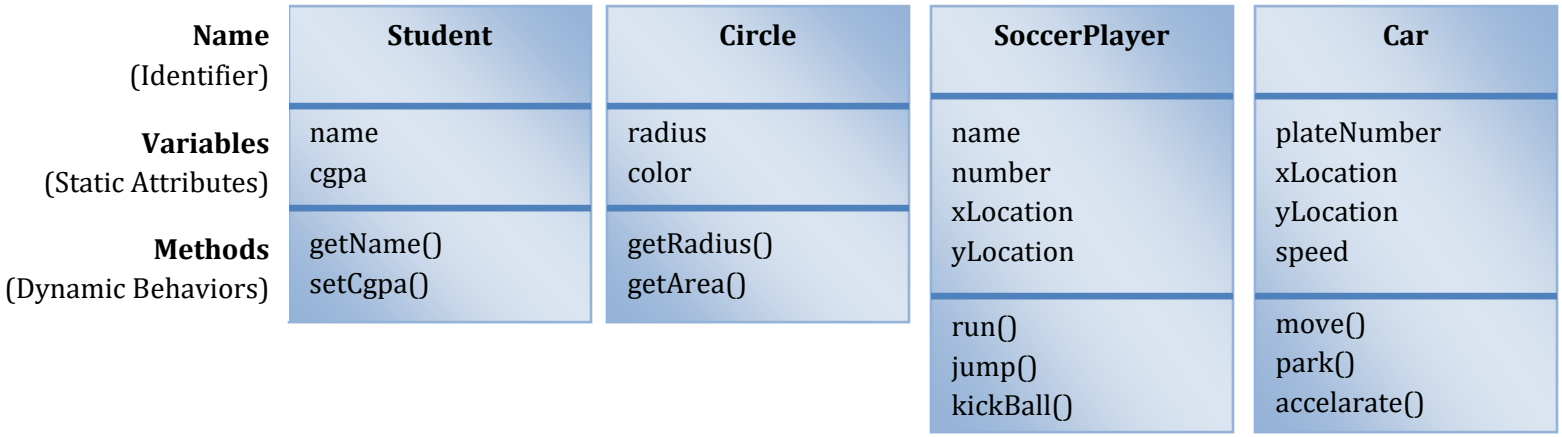


Figure 3.4.2: Example of classes

The following figure shows two instances of the class Student, identified as "paul" and "peter".

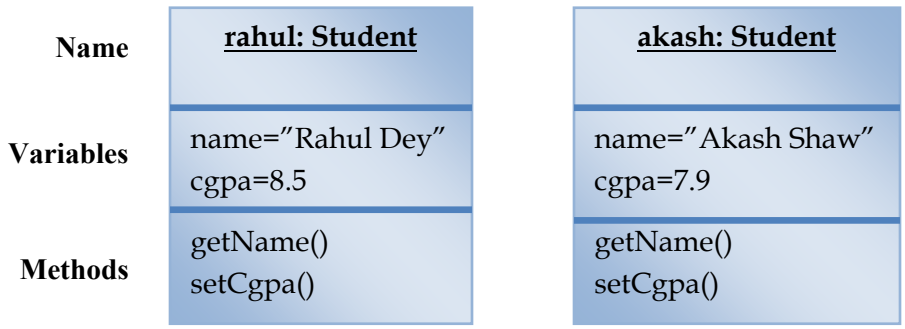


Figure 3.4.3: Two instances – rahul and akash – of the class Student

3.5. Method:

A method is a collection of statements that perform some specific task and return result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++ and Python.

Methods are time savers and help us to reuse the code without retyping the code.

Lecture 4: Properties of OOP

In order for a programming language to be object-oriented, it has to enable working with classes and objects as well as the implementation and use of the fundamental object-oriented principles and concepts: message passing, inheritance, encapsulation, and polymorphism and data abstraction. Let's summarize each of these fundamental principles of OOP:

4.1. Message Passing:

In object oriented languages, you can consider a running program under execution as a pool of objects where objects are created for 'interaction' and later destroyed when their job is over. This interaction is based on 'messages' which are sent from one object to another asking the recipient object to apply one of its own methods on itself and hence, forcing a change in its state. The objects are made to communicate or interact with each other with the help of a mechanism called message passing. The methods of any object may communicate with each other by sending and receiving messages in order to change the state of the object. An object may communicate with other objects by sending and receiving messages to and from their methods in order to change either its own state or the state of other objects taking part in this communication or that of both. An object can both send and receive messages.

4.2. Inheritance:

In OOP, computer programs are designed in such a way where everything is an object that interact with one another. Inheritance is one such concept where the properties of one class can be inherited by the other. It helps to reuse the code and establish a relationship between different classes.

Figure 4.2.1: Real-Life Example of Inheritance

As we can see in the image, a child inherits the properties from his father. Similarly, in Java, there are two classes:

1. Parent class (Super or Base class)
2. Child class (Subclass or Derived class)

A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

Inheritance is further classified into 4 types:

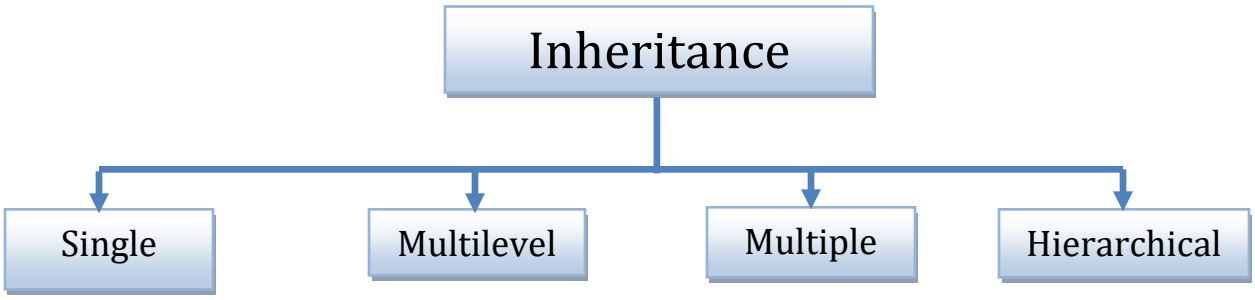


Figure 4.2.2: Types of Inheritance

4.3. Encapsulation:

Encapsulation is a mechanism where you bind your data and code together as a single unit. It also means to hide your data in order to make it safe from any modification. What does this mean? The best way to understand encapsulation is to look at the example of a medical capsule, where the drug is always safe inside the capsule. Similarly, through encapsulation the methods and variables of a class are well hidden and safe.

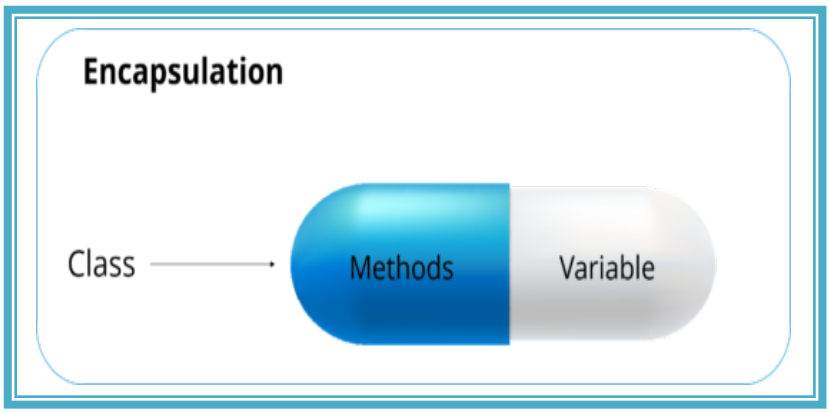


Figure 4.3: Real-Life Example of Encapsulation

4.4. Polymorphism:

Polymorphism means taking many forms, where 'poly' means many and 'morph' means forms. It is the ability of a variable, function or object to take on multiple forms. In other words, polymorphism allows you define one interface or method and have multiple implementations.

Let's understand this by taking a real-life example and how this concept fits into Object oriented programming.

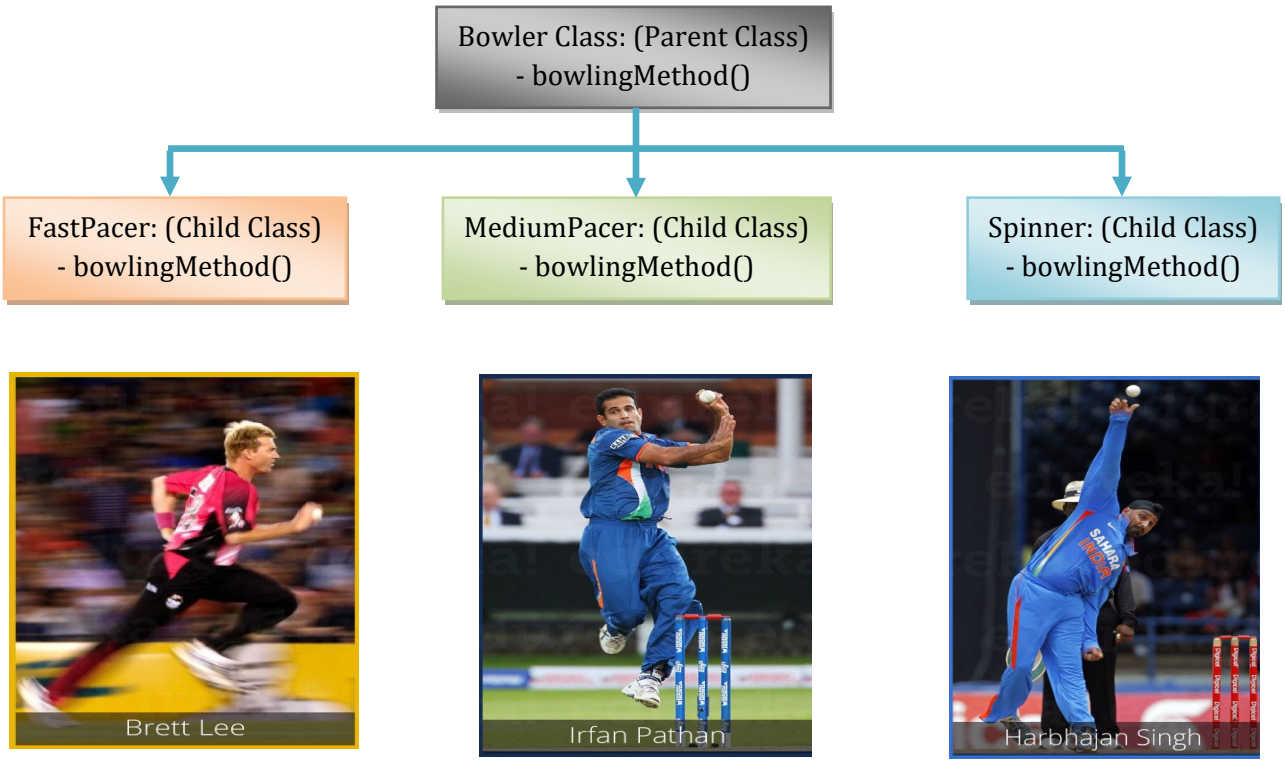


Figure 4.4: Real-Life Example of Polymorphism

Let's consider this real world scenario in cricket, we know that there are different types of bowlers i.e. fast bowlers, Medium pace bowlers and spinners. As you can see in the above figure, there is a parent class- BowlerClass and it has three child classes: FastPacer, MediumPacer and Spinner. Bowler class has bowlingMethod() where all the child classes are inheriting this method. As we all know that a fast bowler will going to bowl differently as compared to medium pacer and spinner in terms of bowling speed, long run up and way of bowling, etc. Similarly a medium pacer's implementation of bowlingMethod() is also going to be different as compared to other bowlers. And same happens with spinner class.

The point of above discussion is simply that a same name tends to multiple forms. All the three classes above inherited the bowlingMethod() but their implementation is totally different from one another.

4.5. Data abstraction:

Data abstraction refers to the quality of dealing with ideas rather than events. It basically deals with hiding the details and showing the essential things to the user. If you look at the image here, whenever we get a call, we get an option to either pick it up or just reject it. But in reality, there is a lot of code that runs in the background. So you don't know the internal processing of how a call is generated, that's the beauty of abstraction. Therefore, abstraction helps to reduce complexity.

Figure 4.5: Real-Life Example of Data abstraction

Lecture 5: Difference between different OOPs Languages

5.1. A Brief History:

An object-oriented programming language is one that allowsobject-oriented programming techniques such as encapsulation, inheritance, modularity, and polymorphism. Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs. The idea of object-oriented programming gained momentum in the 1970s with the introduction of Smalltalk (1972 to 1980), which embraced the concepts of class and message of Simula. Smalltalk is the language with which much of the theory of object-oriented programming was developed. In the early 1980s, Bjorn Stroustrup integrated object-oriented programming into the C language. The resulting language was called C++ and it became the first object-oriented language to be widely used commercially. Then in the 1990s a group at sun led by James Gosling developed a similar version of C++ called Java that was developed to let

devices, peripherals and appliances possess a common programming interface. In 2000, Microsoft announced both the .NET platform and a new programming language called C#. C# is similar in many respects to C++ and Java. Ruby and Python are scripting languages which support the object-oriented paradigm.

Properties of Smalltalk, Java, C++, C#, Eiffel, Ruby and Python which are common Object-Oriented Programming Languages (OOPLs) are outlined in this section.

Languages / Properties	Smalltalk	Java	C++	C#	Eiffel	Ruby	Python
Object-Orientation	Pure	Hybrid	Hybrid	Hybrid	Pure	Pure	Hybrid
Static/Dynamic Typing	Dynamic	Static	Static	Static	Static	Dynamic	Dynamic
Inheritance	Single	Single (class) Multiple (interface)	Multiple	Single (class) Multiple (interface)	Multiple	Single (class) Multiple (mixins)	Multiple
Method Overloading	No	Yes	Yes	Yes	No	No	No
Operator Overloading	Yes	No	Yes	Yes	Yes	Yes	Yes
Generic Classes	N.A.	Yes	Yes	Yes	Yes	N.A.	N.A.
Dynamic Binding	Yes	Yes	Yes (static by default)	Yes (static by default)	Yes	Yes	Yes

Table 5.1: Comparison between different OOPLs

In pure OOPLs everything is treated consistently as an object, from primitives such as integers, all the way up to whole classes, prototypes, modules, etc. They are designed to facilitate, even enforce, the object-oriented paradigm. Of the languages that we considered, Smalltalk, Eiffel and Ruby are pure OOPLs. Languages such as C++, Java, C#, and Python were designed mainly for object-oriented programming, but they also have some procedural elements. This is why they fall into the hybrid OOPLs category.

5.1.1. Smalltalk:

Smalltalk was the first general purpose object-oriented programming language. It is a pure dynamically-typed object-oriented language. Smalltalk supports a uniform object model. Everything a programmer deals with is an object including primitive types (such as

numbers) and user-defined types. Clients can access the functionality of a class only by invoking well defined methods. Hence, all operations are performed by sending messages to objects. Smalltalk supports the ability to instantiate objects (as opposed to using method of prototypical objects). Smalltalk supports full inheritance, where all aspects of the parent class are available to the subclass. Smalltalk does not support multiple inheritance. Multiple inheritance can cause significant maintenance burden, as changes in any parent class will affect multiple paths in the inheritance hierarchy. Initial implementations of Smalltalk support reference counting for automatic memory management. The idea is to reduce the programming burden. Moreover, encapsulation is not enforced in Smalltalk as it allows direct access to the instance slots, and it also allows complete visibility of the slots.

5.1.2. C++:

C++ was developed at Bell Labs by Bjarne Stroustrup (1979). It was designed mainly for systems programming, extending the C programming language. C++ is an object-oriented version of C. It has added support for statically-typed object-oriented programming, exception handling, virtual functions, and generic programming to the C programming language. C++ is not a pure object oriented languages, because both procedural and objected-oriented development. It provides multiple inheritance and exception handling, but it does not provide garbage collection. C++ uses compile-time binding, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power of reuse classes. Unlike Java, it has optional bounds checking, it provides access to low-level system facilities. C++ pointers can be used to manipulate specific memory locations, a task necessary for writing low-level operating system components.

5.1.3. Java:

Java is an object-oriented language introduced by Sun Microsystems. It has similar syntax to C++ making it easier to learn. However, Java is not compatible with C++ and it does not allow low-level programming constructs (such as pointers), which ensures type safety and security. Java does not support C/C++ style pointer arithmetic, which allows the garbage collector to relocate referenced objects, and ensures type safety and security. Like Smalltalk, Java has garbage collection and it runs on a protected java virtual machine. Java is designed as a portable language that can run on any web-enabled computer via that computer's web browser. A major benefit of using Java byte code is portability, since the byte code can be executed regardless of the operating system on a given computer. However, running interpreted programs is almost always slower than running programs

compiled to native executables. Moreover, Java has class hierarchy with class Object at the root and provides single inheritance of classes. In addition to classes, Java provides interfaces with multiple inheritance. Java is considered an impure object-oriented language because its built-in types are not objects (e.g. integers), and it has implemented basic arithmetic operations (such as addition, '+') as built-in operators, rather than messages to objects.

5.1.4. C#:

C# is an OOP language part of the .NET framework. C# is not a pure OOPLs since it encompasses functional, imperative and component-oriented programming in addition to the object-oriented paradigm. It has an object-oriented syntax based on C++ and is heavily influenced by Java. In some communities it is thought of as Microsoft's version of Java. Like Java, it has garbage collection and it is compiled to an intermediate language, which is executed by the runtime environment known as Common Language Runtime (CLR) which is similar to the JVM. The C# conception of class and instances, as well as inheritance and polymorphism, are relatively standard. Methods are somewhat more interesting because of the introduction of so-called properties and delegates.

5.1.5. Eiffel:

Eiffel is a proprietary language, which was developed in 1985 as a pure object-oriented language. The design is based on classes. All messages are directed to a class. A class has ability to export some of its attributes for client visibility and keep others hidden. Eiffel enables use of assertions which express formal properties of member methods in terms of preconditions, post conditions, and class invariants. Multiple inheritance is permitted in Eiffel. The name conflict issue is resolved by providing ability to rename the inherited names. Duplicate names are illegal. Several other feature adaptation clauses are available to make multiple inheritance safe. To avoid wrong definitions of the inherited functions, all the assertions defined in parent classes are inherited too. Thus, class designers can choose to define tight constraints that ensure their subclasses do not deviate much from the expected semantics. The ability to 5 assign a subclass object to a superclass pointer is provided with static checking ensuring that such is the case.

Encapsulation is well supported in Eiffel. The class designer controls the visibility of class features. A method can be explicitly made available to all or some or none of the subclasses. The data can be exported in a read only fashion. There is no syntactic difference between an attribute access and access to a method with no parameters. However there is no

control on the inherited attributes. A subclass inherits all the attributes and can change the visibility of the attributes.

5.1.6. Ruby:

Ruby is an object-oriented scripting language developed in 1993 by Matsumiko Yukihiro. It is similar in purpose to python or Perl. Ruby is designed to be an object-oriented programming language based on Perl, and which borrows features from several OO languages like Eiffel and Smalltalk. Ruby has a pure object-oriented support as it does not allow functions. All methods must belong to some class. Ruby only supports single inheritance, though multiple inheritance functionality is indirectly supported through the modules. A module provides a partial class definition.

5.1.7. Python:

Python is an object-oriented scripting language developed in 1990 by Guido Van Rossum. It has become very popular in recent years due to its application in the internet domain. Python allows both procedural and objected-oriented development. Developers can write classes and define methods in them and can also write functions (routines not in a class). There is a different syntax for invoking methods as opposed to invoking functions and this brings out the heterogeneous nature of python programming. Since a programmer can define his own classes, abstraction is supported by python. The encapsulation however is not fully supported as access control is primitive in Python. There are no public, private methods and the only protection is by name mangling. If a programmer knows how name mangling is performed (which is very simple and known mechanism) he could invoke any class method. Python allows multiple inheritance. The issue of name clashes in multiple inheritance is resolved by letting programmer define the order of super classes by the order in which they are declared.

MODULE 2: JAVA BASICS

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be ***Write Once, Run Anywhere***.

Lecture 1: Basic concepts of java programming

1.1. Advantages 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
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

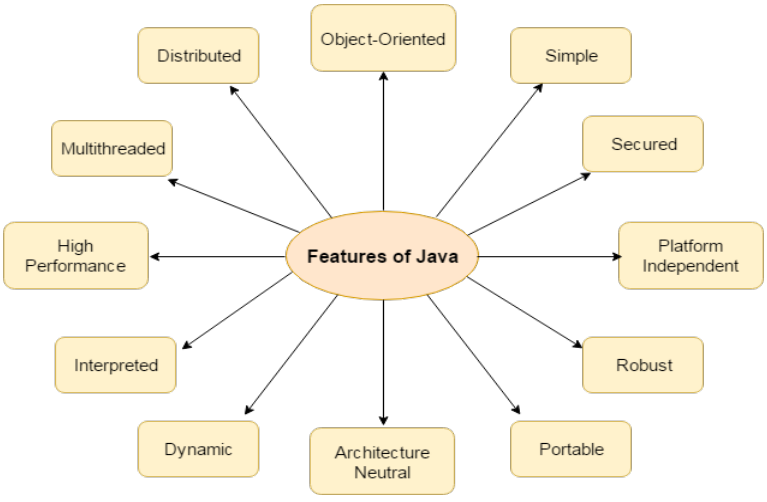


Figure 1.1: Features of Java

1.1.1. 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++ (so 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.

1.1.2. 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 behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

1.1.3. Platform Independent:

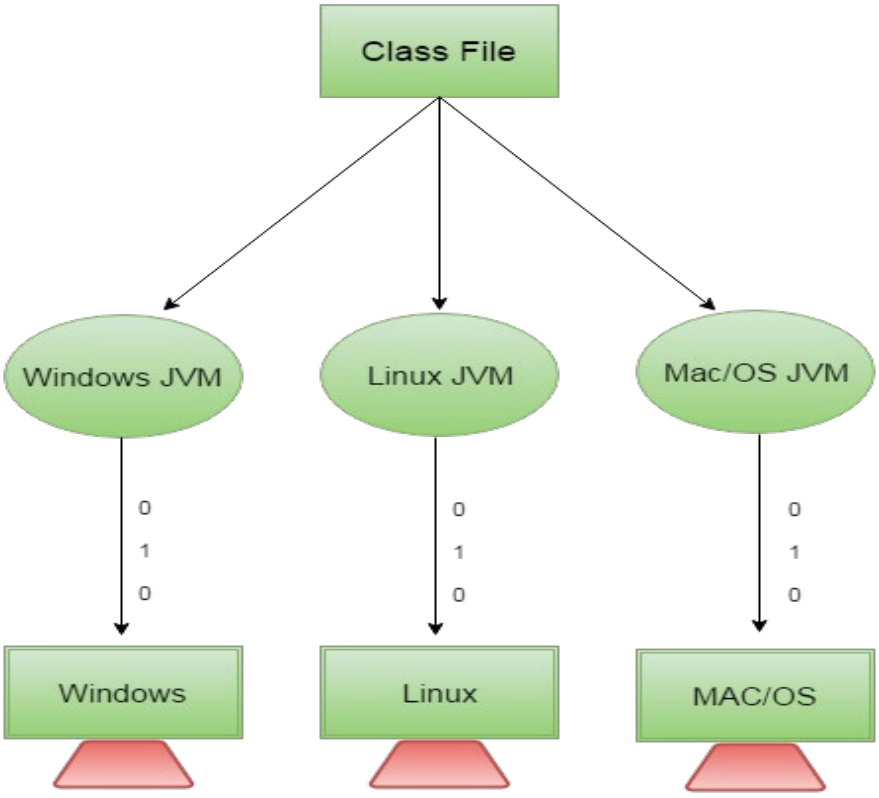
Java is platform independent because it is different from other languages like C, C++ etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides software-based platform.

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:

1. Runtime Environment
2. API(Application Programming Interface)

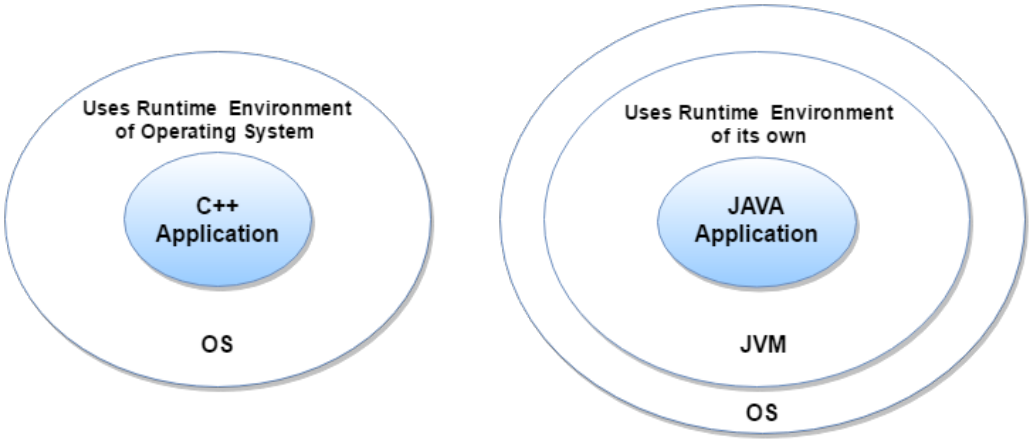
Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere (WORA).



1.1.4. Secured:

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside virtual machine sandbox
- **Class loader:** Class loader 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.



These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, and Cryptography etc.

1.1.5. Robust:

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There are lack of pointers that avoids security problem.
- There is automatic garbage collection in java.
- There is exception handling and type checking mechanism in java. All these points makes java robust.

1.1.6. 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.

1.1.7. Portable:

Java is portable because it facilitates you to carry the java bytecode to any platform.

1.1.8. High-performance:

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++.

1.1.9. Distributed:

Java is distributed because it facilitates us to create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

1.1.10. Multi-threaded:

A thread is like a separate program, executing concurrently. 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.

1.1.11. Simple Program of Java:

To create a simple java program, you need to create a class that contains main method.

1.1.11.1. Creating hello java example:

Let's create the hello java program:

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

Save this file as Simple.java

To compile: javac Simple.java
To execute: java Simple

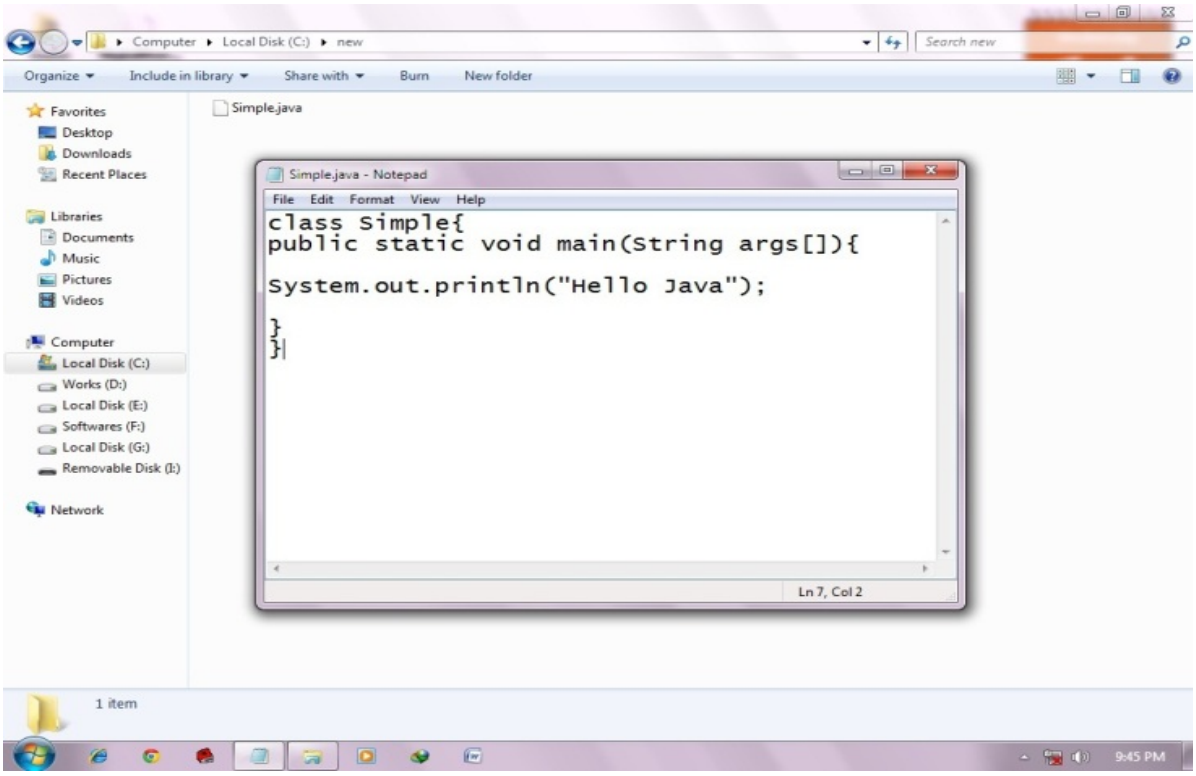
Output: Hello Java

1.1.11.2. Parameters used in first java program:

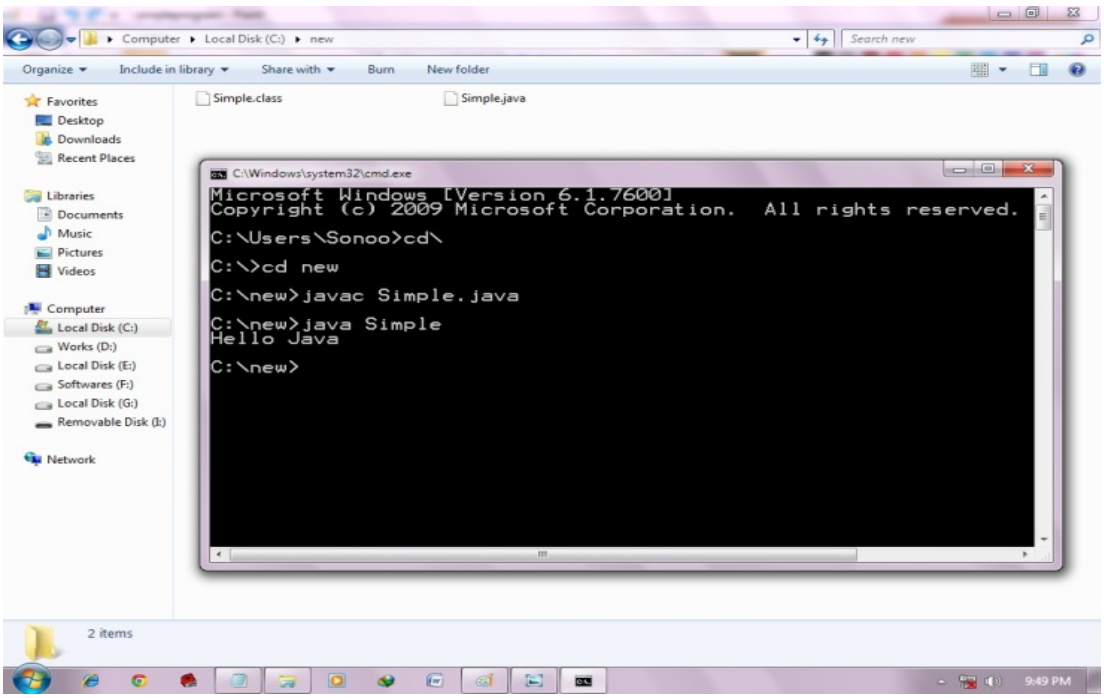
Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.

To write the simple program, open notepad by **start menu -> All Programs -> Accessories -> notepad** and write simple program as displayed below:



As displayed in the above diagram, write the simple program of java in notepad and saved it as Simple.java. To compile and run this program, you need to open command prompt by **start menu -> All Programs -> Accessories -> command prompt**.



To compile and run the above program, go to your current directory first; my current directory is c:\new. Write here:

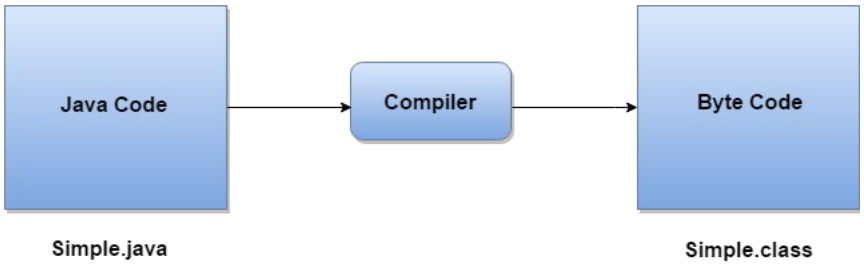
To compile: javac Simple.java
To execute: java Simple

1.2. Byte-Code & JVM:

Here, we are going to learn, what happens while compiling and running the java program.

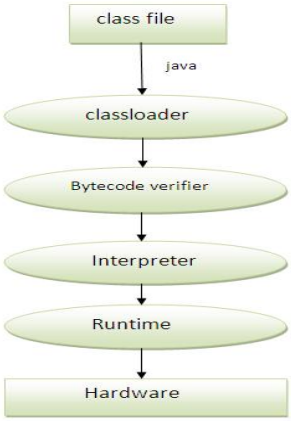
1.2.1. What happens at compile time?

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



1.2.2. What happens at runtime?

At runtime, following steps are performed:



Class loader: 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.

1.2.3. Difference between JDK, JRE and JVM:

We must understand the differences between JDK, JRE and JVM before proceeding further to Java. See the brief overview of JVM here:

1.2.3.1. JVM:

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

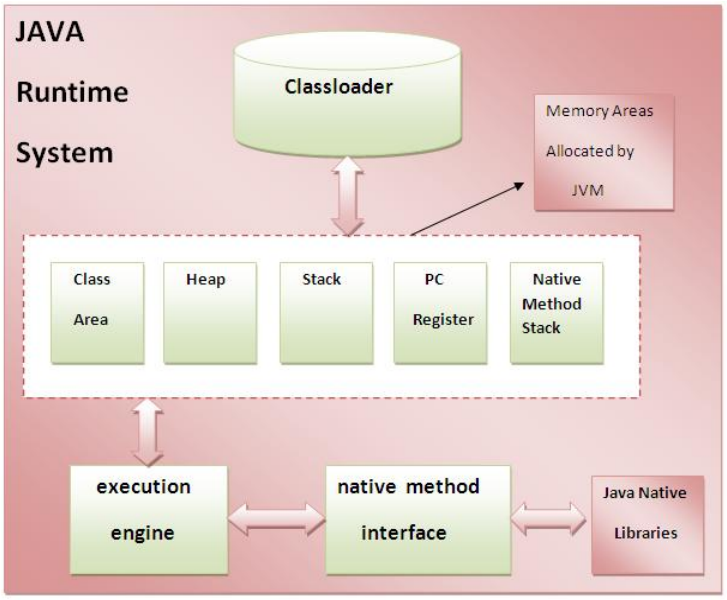
JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

The JVM performs following main tasks:

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

1.2.3.1.1. Internal Architecture of JVM:

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1.2.3.1.1.1. Class loader:

Class loader is a subsystem of JVM that is used to load class files.

1.2.3.1.1.2. Class (Method) Area:

Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

1.2.3.1.1.3. Heap:

It is the runtime data area in which objects are allocated.

1.2.3.1.1.4. Stack:

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

1.2.3.1.1.5. Program Counter Register:

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

1.2.3.1.1.6. Native Method Stack:

It contains all the native methods used in the application.

1.2.3.1.1.7. Execution Engine:

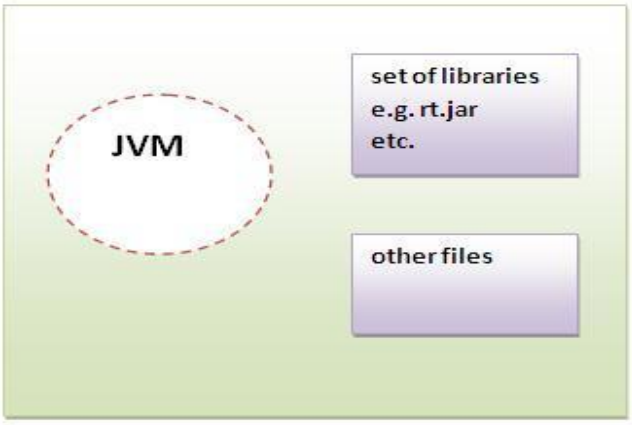
It contains:

1. **A virtual processor**
2. **Interpreter:**Read bytecode stream then execute the instructions.
3. **Just-In-Time (JIT) compiler:**It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

1.2.3.2. 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 + other files that JVM uses at runtime.

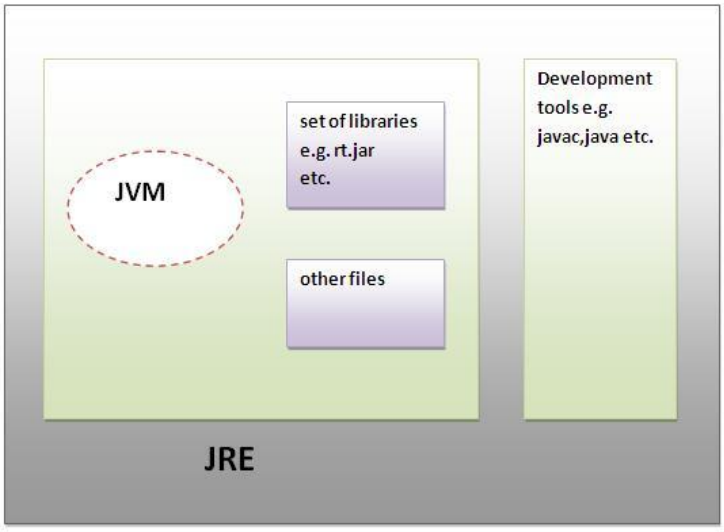
Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

1.2.3.3. JDK:

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



JDK

1.3. Data types:

There are two data types available in Java −

- Primitive Data Types
- Reference/Object Data Types

1.3.1. Primitive Data Types:

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

1.3.1.1. 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.

1.3.1.2. 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.

1.3.1.3. 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.

1.3.1.4. 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.

1.3.1.5. float:

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

1.3.1.6. double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4.

1.3.1.7. Boolean:

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

1.3.1.8. 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'.

1.3.2. Reference Datatypes:

- 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 datatype.
- 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");

1.4. Different types of Variables:

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration –

data type variable [= value][, variable [= value] ...] ;

Here data type is one of Java's datatypes and variable is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java –

Example

```
int a, b, c;         // Declares three ints, a, b, and c.
```

```
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java –

- Local variables
- Instance variables
- Class/Static variables

1.4.1. Local Variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

- ***Example***

  Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to only this method.

```
public class Test {
  public void pupAge() {
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }

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

```
  }
}
```

This will produce the following result –

- ***Output***

Puppy age is: 7

- ***Example***

Following example uses age without initializing it, so it would give an error at the time of compilation.

```
public class Test {
  public void pupAge() {
    int age;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }

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

This will produce the following error while compiling it –

- ***Output***

```
Test.java:4:variable number might not have been initialized
age = age + 7;
^
1 error
```

1.4.2. Instance Variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

- ***Example***

```
import java.io.*;
public class Employee {

  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public Employee (String empName) {
```

```
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal) {
    salary = empSal;
  }

  // This method prints the employee details.
  public void printEmp() {
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }

  public static void main(String args[]) {
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
  }
}
```

This will produce the following result −

- ***Output***

```
name  : Ransika
salary :1000.0
```

1.4.3. Class/Static Variables:

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.

- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

- ***Example***

```
import java.io.*;
public class Employee {

  // salary  variable is a private static variable
  private static double salary;

  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";

  public static void main(String args[]) {
    salary = 1000;
    System.out.println(DEPARTMENT + "average salary:" + salary);
  }
}
```

This will produce the following result −

- ***Output***

```
Development average salary:1000
```

Note – If the variables are accessed from an outside class, the constant should be accessed as Employee.DEPARTMENT.

Lecture 2: Access specifiers, Operators, Control statements & Loops

2.1. Access specifiers:

Modifiers/specifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following –

- Java Access Modifiers
- Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following example.

- ***Example***

```
public class className {
  // ...
}

private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {
  // body of method
}
```

2.1.1. Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are –

- Visible to the package, the default. No modifiers are needed.

- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

2.1.2. Non-Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The static modifier for creating class methods and variables.
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.
- The synchronized and volatile modifiers, which are used for threads.

2.2. 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
- Misc Operators

2.2.1. The 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 −

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

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

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 25;
    int d = 25;

    System.out.println("a + b = " + (a + b) );
    System.out.println("a - b = " + (a - b) );
    System.out.println("a * b = " + (a * b) );
    System.out.println("b / a = " + (b / a) );
    System.out.println("b % a = " + (b % a) );
    System.out.println("c % a = " + (c % a) );
    System.out.println("a++   = " +  (a++) );
    System.out.println("b--   = " +  (a--) );

    // Check the difference in d++ and ++d
    System.out.println("d++   = " +  (d++) );
    System.out.println("++d   = " +  (++d) );
  }
}
```

This will produce the following result –

- ***Output***

a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++ = 10
b-- = 11
d++ = 25
++d = 27

2.2.2. The Relational Operators:

There are following relational operators supported by Java language.

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.

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

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;

    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
  }
}
```

This will produce the following result −

- ***Output***

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

2.2.3. The 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 –

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)	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)	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.	A >>>2 will give 15 which is 0000 1111

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

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int a = 60;  /* 60 = 0011 1100 */
    int b = 13;  /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;       /* 12 = 0000 1100 */
    System.out.println("a & b = " + c );

    c = a | b;       /* 61 = 0011 1101 */
    System.out.println("a | b = " + c );

    c = a ^ b;       /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c );

    c = ~a;          /*-61 = 1100 0011 */
    System.out.println("~a = " + c );

    c = a << 2;      /* 240 = 1111 0000 */
    System.out.println("a << 2 = " + c );

    c = a >> 2;      /* 15 = 1111 */
    System.out.println("a >> 2  = " + c );

    c = a >>> 2;     /* 15 = 0000 1111 */
    System.out.println("a >>> 2 = " + c );
  }
}
```

This will produce the following result −

- ***Output***

```
a & b = 12
```

a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2 = 15
a >>> 2 = 15

2.2.4. The Logical Operators:

The following table lists the logical operators –

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

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

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;

    System.out.println("a && b = " + (a&&b));
    System.out.println("a || b = " + (a||b) );
    System.out.println("!(a && b) = " + !(a && b));
  }
}
```

This will produce the following result –

- ***Output***

```
a && b = false
a || b = true
!(a && b) = true
```

2.2.5. The 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 ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 0;

    c = a + b;
    System.out.println("c = a + b = " + c );

    c += a ;
    System.out.println("c += a  = " + c );

    c -= a ;
    System.out.println("c -= a = " + c );

    c *= a ;
    System.out.println("c *= a = " + c );

    a = 10;
    c = 15;
    c /= a ;
    System.out.println("c /= a = " + c );

    a = 10;
```

```
        c = 15;
        c %= a ;
        System.out.println("c %= a  = " + c );

        c <<= 2 ;
        System.out.println("c <<= 2 = " + c );

        c >>= 2 ;
        System.out.println("c >>= 2 = " + c );

        c >>= 2 ;
        System.out.println("c >>= 2 = " + c );

        c &= a ;
        System.out.println("c &= a  = " + c );

        c ^= a ;
        System.out.println("c ^= a   = " + c );

        c |= a ;
        System.out.println("c |= a   = " + c );
    }
}
```

This will produce the following result –

- ***Output***

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
```

```
c ^= a   = 10
c |= a   = 10
```

2.2.6. Miscellaneous Operators:

There are few other operators supported by Java Language.

2.2.6.1. 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

Following is an example −

- ***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 );
  }
}
```

This will produce the following result −

- ***Output***

```
Value of b is : 30
Value of b is : 20
```

2.2.6.2. 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. Following is an example –

- ***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 );
  }
}
```

This will produce the following result –

- ***Output***

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example –

- ***Example***

```
class Vehicle {}

public class Car extends Vehicle {

  public static void main(String args[]) {

    Vehicle a = new Car();
    boolean result =  a instanceof Car;
    System.out.println( result );
  }
}
```

This will produce the following result –

- ***Output***

true

2.2.7. Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3 * 2 and then adds into 7.

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.

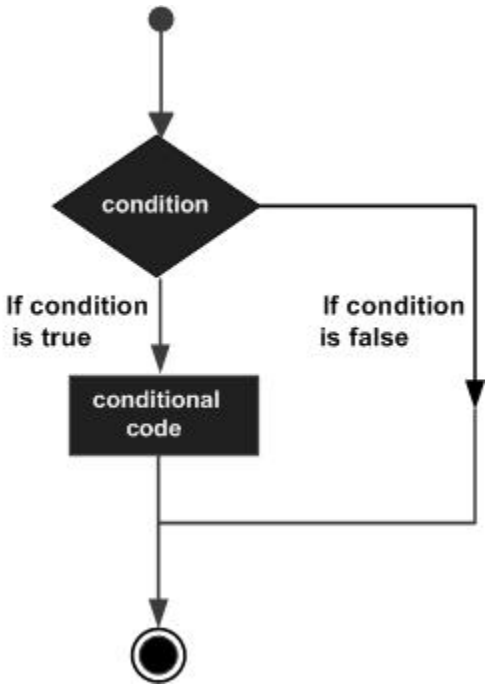
Category	Operator	Associativity
Postfix	>() [] . (dot operator)	Left toright
Unary	>++ - - ! ~	Right to left
Multiplicative	>* /	Left to right
Additive	>+ -	Left to right
Shift	>>>>>>><	Left to right
Relational	>>>= <<=	Left to right

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
Assignment	>= += -= *= /= %= >>= <<= &= ^= |=	Right to left

2.3. Control statements & loops:

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages –



Java programming language provides following types of decision making statements –

Sr.No.	Statement & Description
1	**if statement**

	An**if statement** consists of a boolean expression followed by one or more statements.
2	**if...else statement** An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
3	**nested if statement** You can use one **if** or **else if** statement inside another **if** or **else if**statement(s).
4	**switch statement** A **switch** statement allows a variable to be tested for equality against a list of values.

2.3.1. if statement:

An if statement consists of a Boolean expression followed by one or more statements.

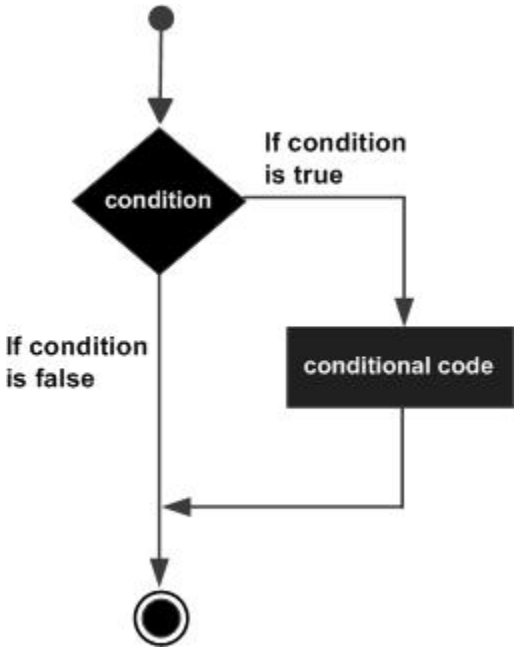
- ***Syntax***

  Following is the syntax of an if statement −

```
if(Boolean_expression) {
  // Statements will execute if the Boolean expression is true
}
```

  If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

- ***Flow Diagram***



- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    if( x < 20 ) {
      System.out.print("This is if statement");
    }
  }
}
```

This will produce the following result –

- ***Output***

This is if statement.

2.3.2. if...else statement:

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

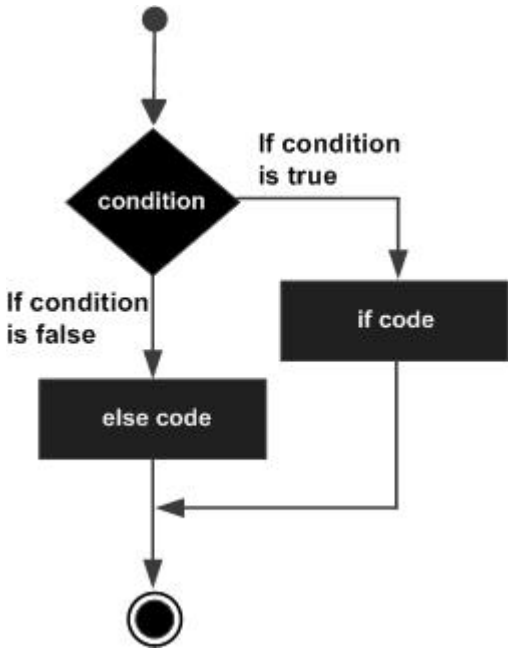
- ***Syntax***

  Following is the syntax of an if...else statement –

```
if(Boolean_expression) {
  // Executes when the Boolean expression is true
}else {
  // Executes when the Boolean expression is false
}
```

  If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

- ***Flow Diagram***



- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int x = 30;

    if( x < 20 ) {
      System.out.print("This is if statement");
```

```
    }else {
      System.out.print("This is else statement");
    }
  }
}
```

This will produce the following result –

- ***Output***

  This is else statement

2.3.2.1. The if...else if...else Statement:

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

- ***Syntax***

  Following is the syntax of an if...else statement –

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2) {
  // Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3) {
  // Executes when the Boolean expression 3 is true
}else {
  // Executes when the none of the above condition is true.
}
```

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int x = 30;

    if( x == 10 ) {
      System.out.print("Value of X is 10");
    }else if( x == 20 ) {
      System.out.print("Value of X is 20");
    }else if( x == 30 ) {
      System.out.print("Value of X is 30");
    }else {
      System.out.print("This is else statement");
    }
  }
}
```

This will produce the following result –

- ***Output***

  Value of X is 30

2.3.3. nested if statement:

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

- ***Syntax***

  The syntax for a nested if...else is as follows –

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
  if(Boolean_expression 2) {
    // Executes when the Boolean expression 2 is true
  }
}
```

You can nest else if...else in the similar way as we have nested if statement.

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int x = 30;
    int y = 10;

    if( x == 30 ) {
      if( y == 10 ) {
        System.out.print("X = 30 and Y = 10");
      }
    }
  }
}
```

This will produce the following result −

- ***Output***

```
X = 30 and Y = 10
```

2.3.4. switch statement:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

- ***Syntax***

The syntax of enhanced for loop is −

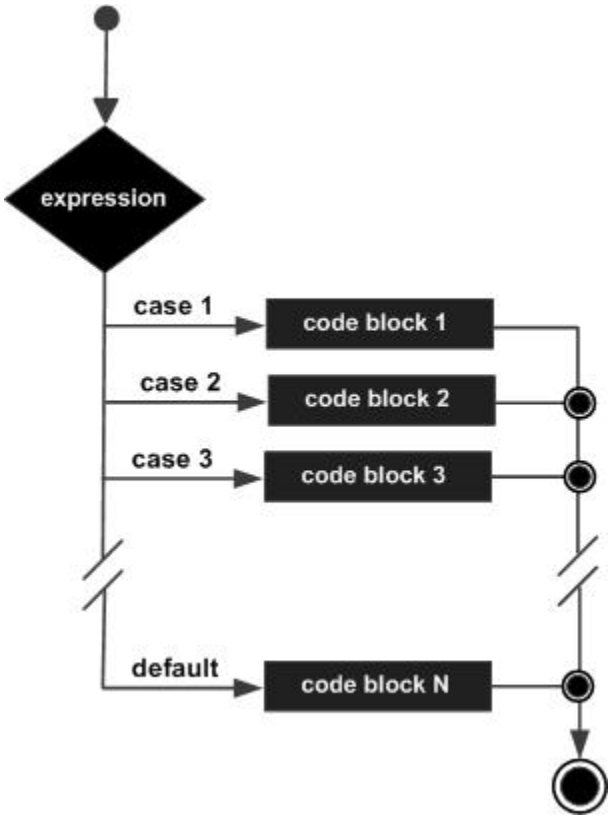
```
switch(expression) {
  case value :
    // Statements
    break; // optional
```

```
   case value :
     // Statements
     break; // optional

   // You can have any number of case statements.
   default : // Optional
     // Statements
}
```

The following rules apply to a switch statement –

- The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

- ***Flow Diagram***



- ***Example***

```
public class Test {

  public static void main(String args[]) {
    // char grade = args[0].charAt(0);
    char grade = 'C';

    switch(grade) {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
```

```
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

Compile and run the above program using various command line arguments. This will produce the following result –

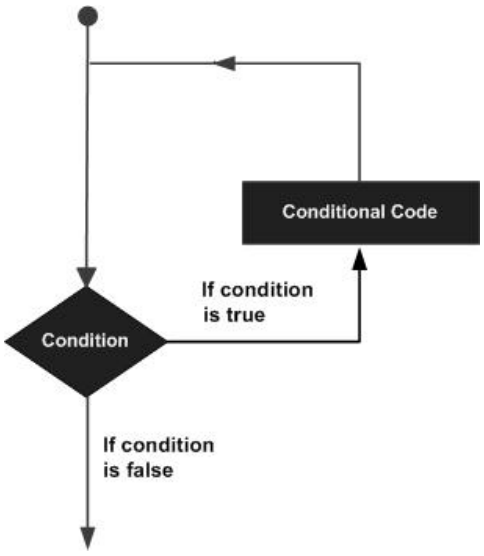
- ***Output***

```
Well done
Your grade is C
```

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



Java programming language provides the following types of loop to handle looping requirements.

Sr.No.	Loop & Description
1	**while loop** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** Like a while statement, except that it tests the condition at the end of the loop body.

2.3.5. while loop:

A while loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

- ***Syntax***

  The syntax of a while loop is −

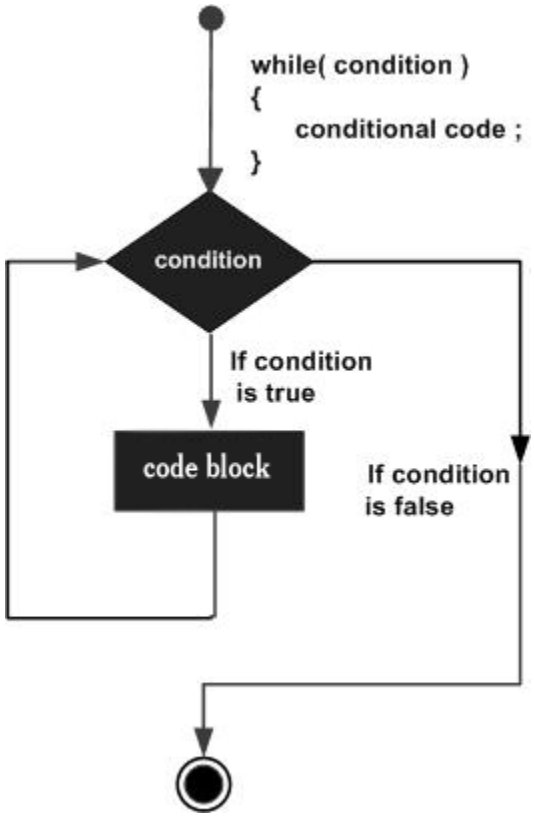
```
while(Boolean_expression) {
  // Statements
}
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non zero value.

When executing, if the boolean_expression result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

- ***Flow Diagram***



Here, key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    while( x < 20 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
}
```

This will produce the following result –

- ***Output***

value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19

2.3.6. for loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

- ***Syntax***

The syntax of a for loop is –

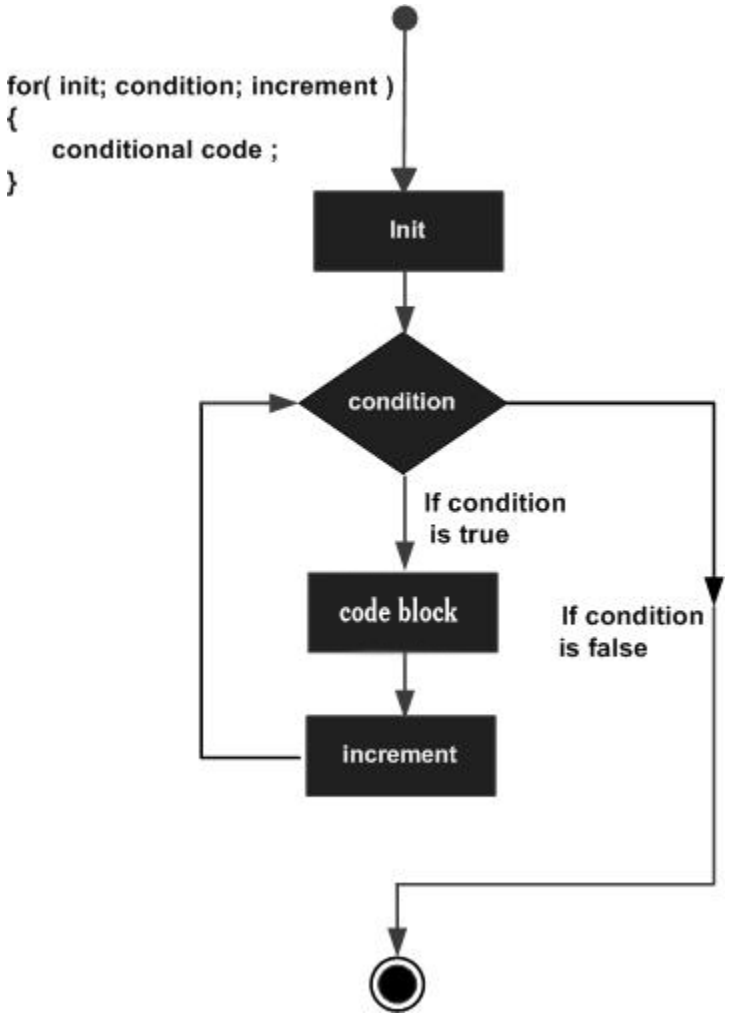
```
for(initialization; Boolean_expression; update) {
  // Statements
}
```

Here is the flow of control in a for loop –

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.

- After the body of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

- ***Flow Diagram***



- ***Example***

Following is an example code of the for loop in Java.

```
public class Test {

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

```
    for(int x = 10; x < 20; x = x + 1) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
    }
  }
}
```

This will produce the following result −

- ***Output***

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

2.3.7. do...while loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

- ***Syntax***

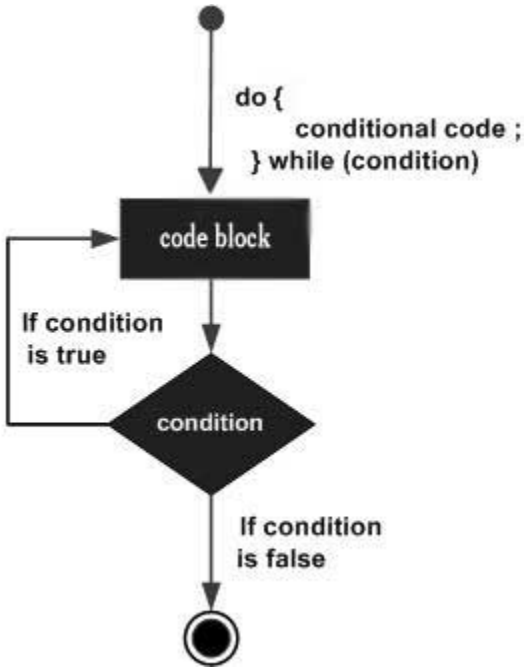
Following is the syntax of a do...while loop −

```
do {
  // Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

- ***Flow Diagram***



- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    do {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
}
```

This will produce the following result –

- ***Output***

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

2.3.8. Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements.

Sr.No.	Control Statement & Description
1	**break statement** Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

2.3.8.1. break statement:

The break statement in Java programming language has the following two usages −

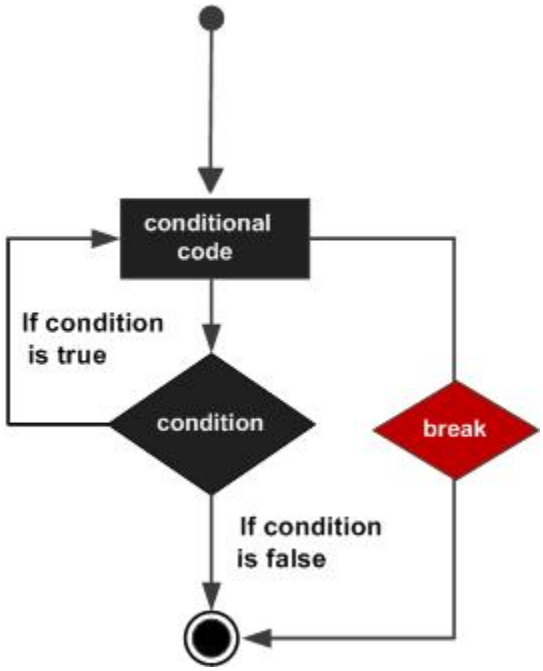
- When the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the switch statement.

- ***Syntax***

  The syntax of a break is a single statement inside any loop –

  ```
  break;
  ```

- ***Flow Diagram***



- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      if( x == 30 ) {
        break;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This will produce the following result –

- ***Output***

10
20

2.3.8.2. continue statement:

The continue keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

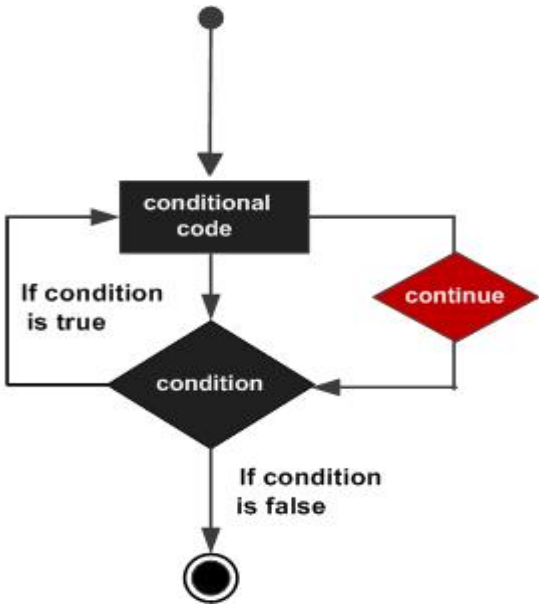
- In a for loop, the continue keyword causes control to immediately jump to the update statement.
- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

- ***Syntax***

The syntax of a continue is a single statement inside any loop –

continue;

- ***Flow Diagram***



- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      if( x == 30 ) {
        continue;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This will produce the following result −

- ***Output***

```
10
20
40
50
```

2.3.9. Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

- ***Syntax***

Following is the syntax of enhanced for loop −

```
for(declaration : expression) {
  // Statements
}
```

- Declaration − The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- Expression − This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

- ***Example***

```
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      System.out.print( x );
      System.out.print(",");
    }
    System.out.print("\n");
    String [] names = {"James", "Larry", "Tom", "Lacy"};

    for( String name : names ) {
      System.out.print( name );
      System.out.print(",");
    }
  }
}
```

This will produce the following result −

- ***Output***

```
10, 20, 30, 40, 50,
James, Larry, Tom, Lacy,
```

Lecture 3: Array

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

3.1. Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

- ***Syntax***

  ```
  dataType[] arrayRefVar;   // preferred way.
  or
  dataType arrayRefVar[];  // works but not preferred way.
  ```

Note – The style dataType[] arrayRefVar is preferred. The style dataType arrayRefVar[] comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

- ***Example***

  The following code snippets are examples of this syntax –

  ```
  double[] myList;   // preferred way.
  or
  double myList[];   // works but not preferred way.
  ```

3.2. Creating Arrays:

You can create an array by using the new operator with the following syntax −

- ***Syntax***

  arrayRefVar = new dataType[arraySize];

The above statement does two things −

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows −

dataType[] arrayRefVar = {value0, value1, ..., valuek};

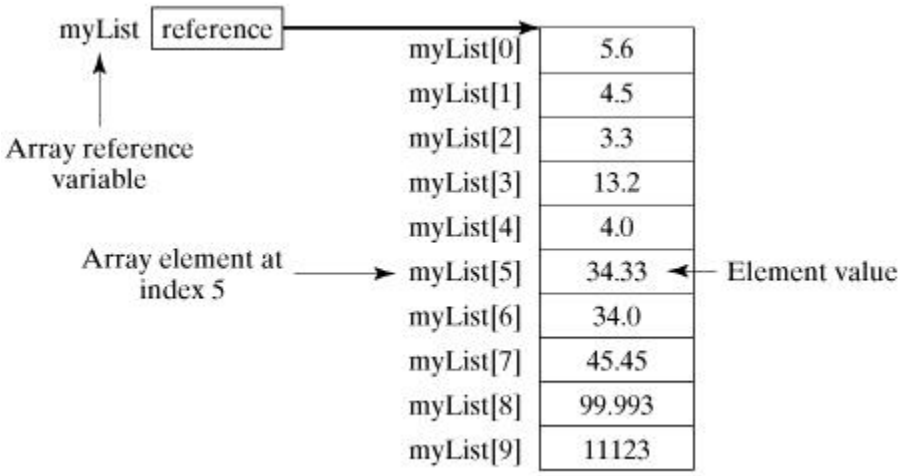
The array elements are accessed through the index. Array indices are 0-based; that is, they start from 0 to arrayRefVar.length-1.

- ***Example***

  Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

  double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



3.3. Processing Arrays:

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

- ***Example***

  Here is a complete example showing how to create, initialize, and process arrays –

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }

    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);

    // Finding the largest element
```

```
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

This will produce the following result –

- ***Output***

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

3.4. The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

- ***Example***

The following code displays all the elements in the array myList –

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (double element: myList) {
      System.out.println(element);
    }
  }
```

```
}
```

This will produce the following result –

- ***Output***

```
1.9
2.9
3.4
3.5
```

3.5. Multidimensional array in java:

In such case, data is stored in row and column based index (also known as matrix form).

- ***Syntax***

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

- ***Example to instantiate Multidimensional Array in java***

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

- ***Example to initialize Multidimensional Array in java***

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

- ***Example of Multidimensional java array***

  Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
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
```

Lecture 4: Creation of Class, Object, Method

In this chapter, we will look into the concepts - Classes and Objects.

- **Object –** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
- **Class –** A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

4.1. Classes in Java:

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

- ***Example***

```
public class Dog {
  String breed;
  int age;
  String color;
}
```

A class can have any number of variables which will store the data.

4.2. Objects in Java:

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

4.2.1. Creating an Object:

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

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

- **Declaration –** A variable declaration with a variable name with an object type.
- **Instantiation –** The 'new' keyword is used to create the object.
- **Initialization –** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

- ***Example***

```
public class Puppy {
  String color;

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

    myPuppy.color="beige";
    System.out.println("The color of the puppy is : " +myPuppy.color);
  }
}
```

If we compile and run the above program, then it will produce the following result −

- ***Output***

```
The color of the puppy is : beige
```

4.3. Methods in Java:

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

4.3.1. Creating Method:

Considering the following example to explain the syntax of a method –

- ***Syntax***

```
public static int methodName(int a, int b) {
  // body
}
```

Here,

- public static – modifier
- int – return type
- methodName – name of the method
- a, b – formal parameters
- int a, int b – list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax –

- ***Syntax***

```
modifier returnType nameOfMethod (Parameter List) {
  // method body
}
```

The syntax shown above includes –

- modifier – It defines the access type of the method and it is optional to use.
- returnType – Method may return a value.
- nameOfMethod – This is the method name. The method signature consists of the method name and the parameter list.
- Parameter List – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- method body – The method body defines what the method does with the statements.

- ***Example***

  Here is the source code of the above defined method called min(). This method takes two parameters num1 and num2 and returns the maximum between the two –

```
/** the snippet returns the minimum between two numbers */

public static int minFunction(int n1, int n2) {
  int min;
  if (n1 > n2)
    min = n2;
  else
    min = n1;

  return min;
}
```

4.3.2. Method Calling:

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example –

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example –

```
int result = sum(6, 9);
```

Following is the example to demonstrate how to define a method and how to call it –

- ***Example***

```
public class ExampleMinNumber {

  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
  }

  /** returns the minimum of two numbers */
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
}
```

This will produce the following result –

- ***Output***

```
Minimum value = 6
```

4.3.3. The void Keyword:

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method methodRankPoints. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. methodRankPoints(255.7);. It is a Java statement which ends with a semicolon as shown in the following example.

- ***Example***

```
public class ExampleVoid {

  public static void main(String[] args) {
    methodRankPoints(255.7);
  }

  public static void methodRankPoints(double points) {
    if (points >= 202.5) {
      System.out.println("Rank:A1");
    }else if (points >= 122.4) {
      System.out.println("Rank:A2");
    }else {
      System.out.println("Rank:A3");
    }
  }
}
```

This will produce the following result –

- ***Output***

```
Rank:A1
```

4.4. Accessing Instance Variables and Methods:

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path –

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
```

ObjectReference.MethodName();

- ***Example***

  This example explains how to access instance variables and methods of a class.

```
public class Puppy {
  int puppyAge;

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

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

  public int getAge( ) {
    System.out.println("Puppy's age is :" + puppyAge );
    return puppyAge;
  }

  public static void main(String []args) {
    /* Object creation */
    Puppy myPuppy = new Puppy( "tommy" );

    /* Call class method to set puppy's age */
    myPuppy.setAge( 2 );

    /* Call another class method to get puppy's age */
    myPuppy.getAge( );

    /* You can access instance variable as follows as well */
    System.out.println("Variable Value :" + myPuppy.puppyAge );
  }
}
```

  If we compile and run the above program, then it will produce the following result –

- ***Output***

  Name chosen is :tommy
  Puppy's age is :2
  Variable Value :2

4.5. Source File Declaration Rules:

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, import statements and package statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.
- The public class name should be the name of the source file as well which should be appended by .java at the end. For example: the class name is public class Employee{} then the source file should be as Employee.java.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Lecture 5: Constructor

In Java, constructor is a block of codes similar to method. It is called when an instance of object is created and memory is allocated for the object.

It is a special type of method which is used to initialize the object.

5.1. When a constructor is called:

Everytime an object is created using new() keyword, atleast one constructor is called. It is called a default constructor.

Note: It is called constructor because it constructs the values at the time of object creation. 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.

5.2. Rules for creating java constructor:

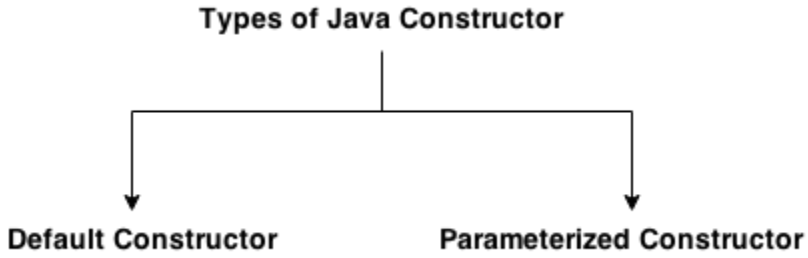
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

5.3. Types of java constructors:

There are two types of constructors in java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



5.4. Java Default Constructor:

A constructor is called "Default Constructor" when it doesn't have any parameter.

- ***Syntax of default constructor***

  <class_name>(){}

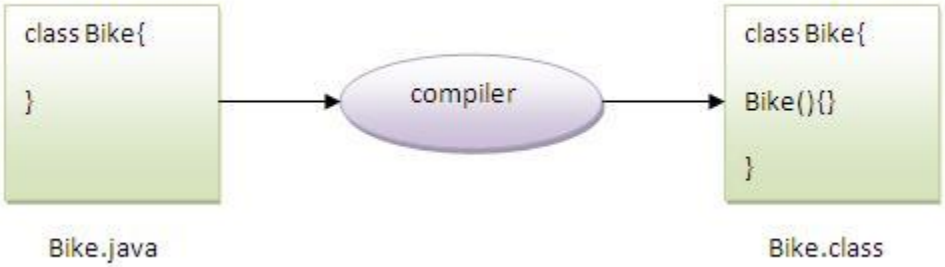
- ***Example of default constructor***

  In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

- ***Output***

  Bike is created



Note –Default constructor is used to provide the default values to the object like 0, null etc. depending on the type.

- ***Example of default constructor that displays the default values***

```
class Student3{
int id;
String name;

void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

- ***Output***

  0 null
  0 null

Explanation – In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor.

5.5. Java parameterized constructor:

A constructor which has a specific number of parameters is called parameterized constructor.

Parameterized constructor is used to provide different values to the distinct objects.

- ***Example of parameterized constructor***

  In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student4{
   int id;
   String name;

   Student4(int i,String n){
   id = i;
   name = n;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

- ***Output***

111 Karan
222 Aryan

5.6. Difference between constructor and method in java:

There are many differences between constructors and methods. They are given below.

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.

5.7. Java Copy Constructor:

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
class Student6{
   int id;
   String name;
   Student6(int i,String n){
   id = i;
```

```
  name = n;
  }

  Student6(Student6 s){
  id = s.id;
  name =s.name;
  }
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student6 s1 = new Student6(111,"Karan");
  Student6 s2 = new Student6(s1);
  s1.display();
  s2.display();
 }
}
```

- ***Output***

  111 Karan
  111 Karan

Note – Constructors return a value, i.e. current class instance (You cannot use return type yet it returns a value).

Lecture 6: finalize Method and Garbage Collection, Method & Constructor overloading

6.1. Java Garbage Collection:

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

6.1.1. Advantage of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

6.1.2. How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By annonymous object etc.

6.1.2.1. By nulling a reference:

```
Employee e=new Employee();
e=null;
```

6.1.2.2. By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

6.1.2.3. By annonymous object:

```
new Employee();
```

6.2. finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

Note –The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

6.3. gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Note – Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

6.4. Simple Example of garbage collection in java:

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

- ***Output***

  ```
  object is garbage collected
  object is garbage collected
  ```

Note – Neither finalization nor garbage collection is guaranteed.

6.5. Method Overloading:

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

6.5.1. Advantage of method overloading:

Method overloading increases the readability of the program.

6.5.2. Different ways to overload the method:

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Note –In java, Method Overloading is not possible by changing the return type of the method only.

6.5.2.1. Method Overloading: changing no. of arguments:

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
```

```
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

- ***Output***

  22
  33

6.5.2.2. Method Overloading: changing data type of arguments:

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

- ***Output***

  22
  24.9

6.5.3. Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
```

```
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

- ***Output***

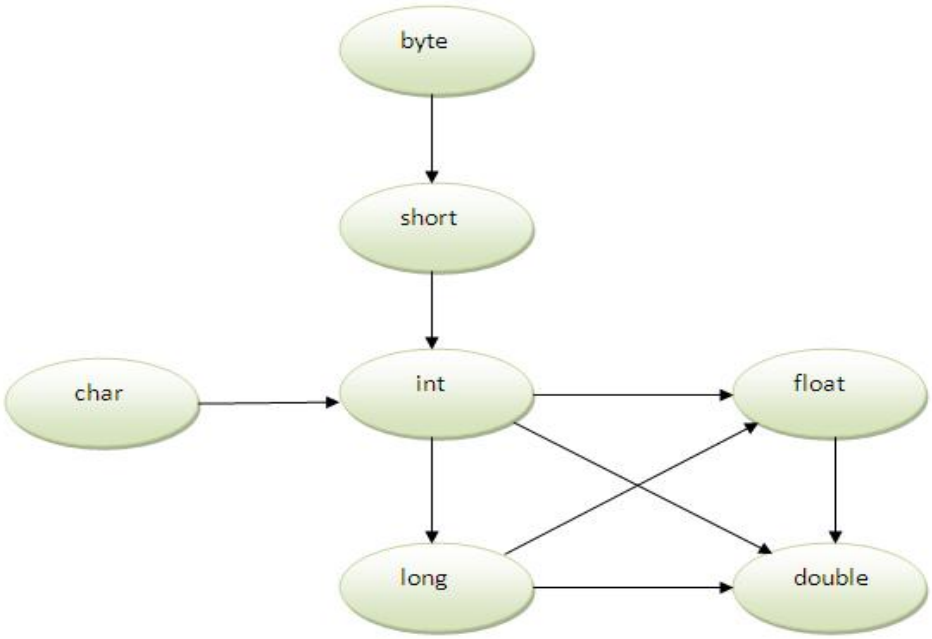
  Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note – Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

6.5.4. Method Overloading and Type Promotion:

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

- ***Example of Method Overloading with TypePromotion***

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);
  }
}
```

- ***Output***

```
40
60
```

- ***Example of Method Overloading with Type Promotion if matching found***

If there are matching type arguments in the method, type promotion is not performed.

```
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

- ***Output***

  int arg method invoked

- ***Example of Method Overloading with Type Promotion in case of ambiguity***

  If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}
```

- ***Output***

  Compile Time Error

Note –One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

6.6. Constructor Overloading:

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

- ***Example of Constructor Overloading***

```
class Student5{
   int id;
   String name;
   int age;
   Student5(int i,String n){
   id = i;
   name = n;
   }
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}

   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

- ***Output***

```
111 Karan 0
222 Aryan 25
```

Lecture 7: this keyword, use of objects as parameter & methods returning objects

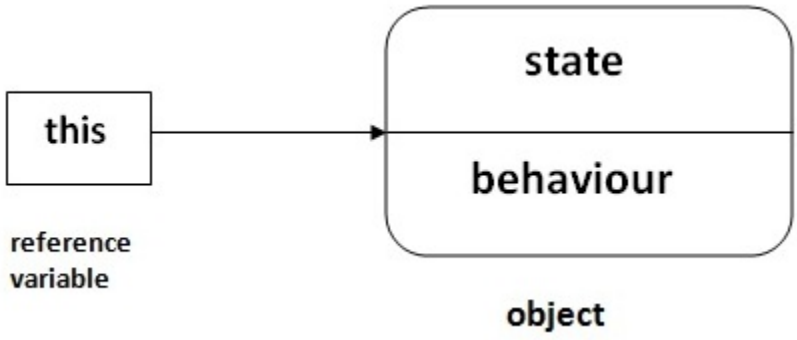
7.1. this keyword in java:

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

7.1.1. Usage of java this keyword:

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.



7.1.1.1. this – to refer current class instance variable:

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

- ***Understanding the problem without this keyword***

  Let's understand the problem if we don't use this keyword by the example given below:

```
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 example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

- ***Solution of the above problem by this keyword***

```
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 TestThis2{
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***

```
111 ankit 5000
112 sumit 6000
```

If local variables (formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

- ***Program where this keyword is not required***

```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
```

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

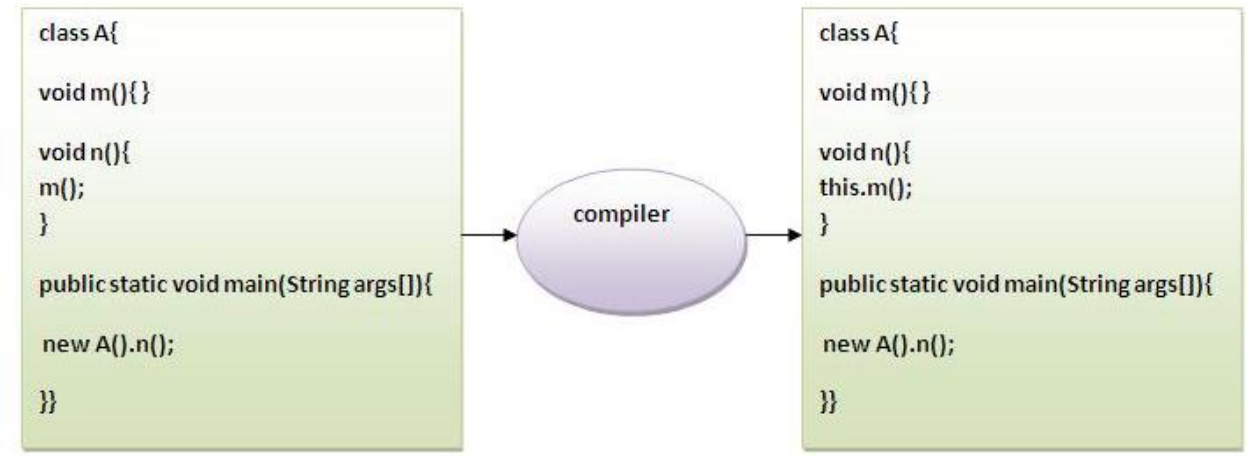
- ***Output***

```
111 ankit 5000
112 sumit 6000
```

Note – It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

7.1.1.2. this – to invoke current class method:

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




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

```
}}
```

- ***Output***

```
hello n
hello m
```

7.1.1.3. this() – to invoke current class constructor:

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

- ***Calling default constructor from parameterized constructor***

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

- ***Output***

```
hello a
10
```

- ***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 a
```

- ***Real usage of this() constructor call***

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. Let's see the example given below that displays the actual use of this keyword.

```
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
```

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

```
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.fee=fee;
this(rollno,name,course);//C.T.Error
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis8{
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***

```
Compile Time Error: Call to this must be first statement in constructor
```

7.1.1.4. this – to pass as an argument in the method:

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the 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

7.1.1.4.1. Application of this that can be passed as an argument:

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.

7.1.1.5. 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

7.1.1.6. 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:

- ***Syntax of this that can be returned as a statement***

  ```
  return_type method_name(){
  return this;
  }
  ```

- ***Example of this keyword that you return as a statement from the method***

  ```
  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

- ***Proving this keyword***
  Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the 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@22b3ea59
A5@22b3ea59
```

7.2. use of objects as parameter:

Although Java is strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed.

When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what

is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference. Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.

- Changes to the object inside the method do reflect in the object used as an argument.

In Java we can pass objects to methods. For example, consider the following program:

```
// Java program to demonstrate objects
// passing to methods.
class ObjectPassDemo
{
   int a, b;

   ObjectPassDemo(int i, int j)
   {
      a = i;
      b = j;
   }

   // return true if o is equal to the invoking
   // object notice an object is passed as an
   // argument to method
   boolean equalTo(ObjectPassDemo o)
   {
      return (o.a == a && o.b == b);
   }
}

// Driver class
```

```
public class Test
{
  public static void main(String args[])
  {
    ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
    ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
    ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
    System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
  }
}
```

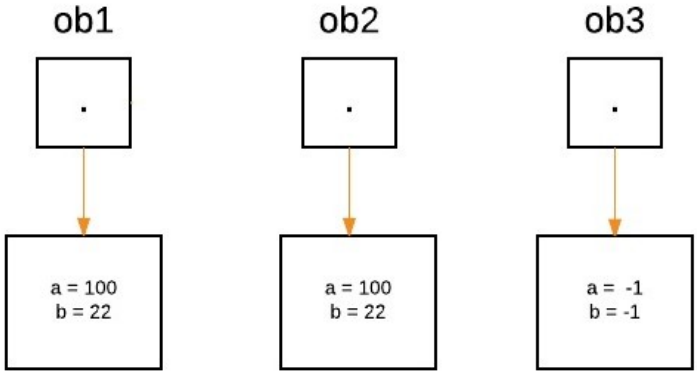
- ***Output***

  ob1 == ob2: true
  ob1 == ob3: false

7.2.1. Illustration:

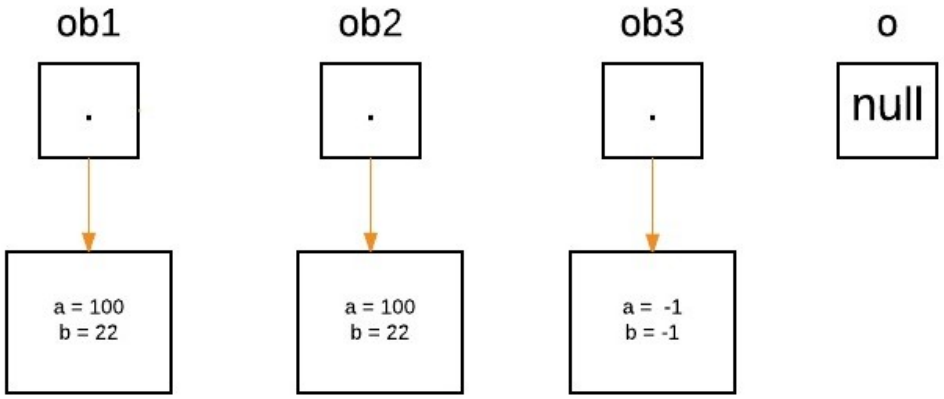
- Three objects 'ob1' , 'ob2' and 'ob3' are created:

  ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
  ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
  ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);



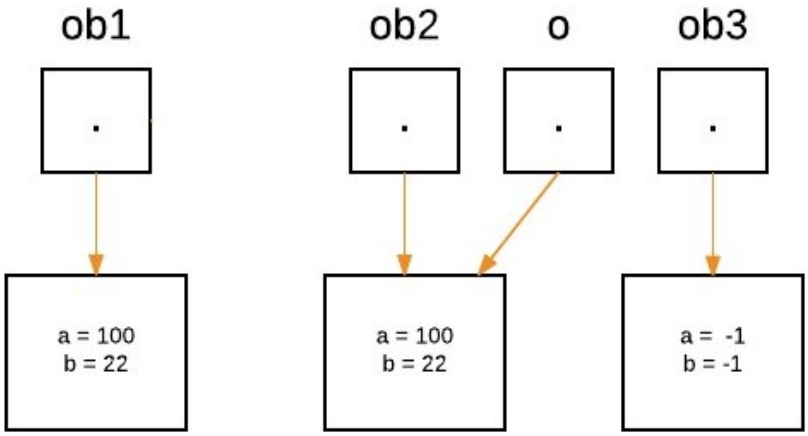
- From the method side, a reference of type Foo with a name a is declared and it's initially assigned to null.

  boolean equalTo(ObjectPassDemo o);



- As we call the method equalTo, the reference 'o' will be assigned to the object which is passed as an argument, i.e. 'o' will refer to 'ob2' as following statement execute.

  System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));

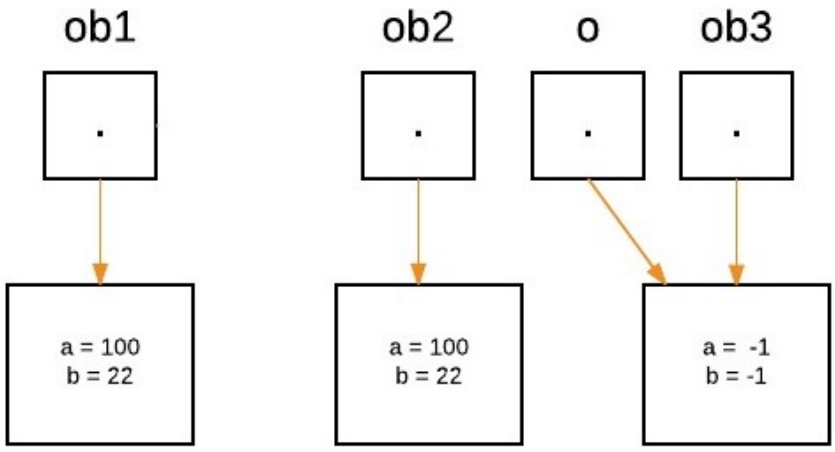


- Now as we can see, equalTo method is called on 'ob1' , and 'o' is referring to 'ob2'. Since values of 'a' and 'b' are same for both the references, so if(condition) is true, so boolean true will be return.

  if(o.a == a && o.b == b)

- Again 'o' will reassign to 'ob3' as the following statement execute.

  System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));



- Now as we can see, equalTo method is called on 'ob1' , and 'o' is referring to 'ob3'. Since values of 'a' and 'b' are not same for both the references, so if(condition) is false, so else block will execute and false will be return.

7.2.2. Defining a constructor that takes an object of its class as a parameter:

One of the most common uses of object parameters involves constructors. Frequently, in practice, there is need to construct a new object so that it is initially the same as some existing object. To do this, either we can use Object.clone() method or define a constructor that takes an object of its class as a parameter. The second option is illustrated in below example:

```
// Java program to demonstrate one object to
// initialize another
class Box
{
   double width, height, depth;

   // Notice this constructor. It takes an
   // object of type Box. This constructor use
   // one object to initialize another
   Box(Box ob)
   {
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
```

```
  // constructor used when all dimensions
  // specified
  Box(double w, double h, double d)
  {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume()
  {
    return width * height * depth;
  }
}

// driver class
public class Test
{
  public static void main(String args[])
  {
    // creating a box with all dimensions specified
    Box mybox = new Box(10, 20, 15);

    //  creating a copy of mybox
    Box myclone = new Box(mybox);

    double vol;

    // get volume of mybox
    vol = mybox.volume();
    System.out.println("Volume of mybox is " + vol);

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

- ***Output***

  Volume of mybox is 3000.0
  Volume of myclone is 3000.0

7.3. Methods returning objects:

In java, a method can return any type of data, including objects. For example, in the following program, the incrByTen( ) method returns an object in which the value of a (an integer variable) is ten greater than it is in the invoking object.

```
// Java program to demonstrate returning
// of objects
class ObjectReturnDemo
{
   int a;

   ObjectReturnDemo(int i)
   {
      a = i;
   }

   // This method returns an object
   ObjectReturnDemo incrByTen()
   {
      ObjectReturnDemo temp =
           new ObjectReturnDemo(a+10);
      return temp;
   }
}

// Driver class
public class Test
{
   public static void main(String args[])
   {
      ObjectReturnDemo ob1 = new ObjectReturnDemo(2);
```

```
    ObjectReturnDemo ob2;

    ob2 = ob1.incrByTen();

    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);
  }
}
```

- ***Output***

  ob1.a: 2
  ob2.a: 12

Note – When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does. That's why we said that java is strictly pass-by-value.

Lecture 8: Call by Value & Call by Reference

8.1. Call by Value in Java:

If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

- ***Example of call by value in java***

  In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{
 int data=50;

 void change(int data){
 data=data+100;//changes will be in the local variable only
 }

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

```
    Operation op=new Operation();

    System.out.println("before change "+op.data);
    op.change(500);
    System.out.println("after change "+op.data);

   }
  }
```

- ***Output***

  before change 50
  after change 50

8.2. Call by Reference in Java:

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

```
class Operation2{
 int data=50;

 void change(Operation2 op){
 op.data=op.data+100;//changes will be in the instance variable
 }


 public static void main(String args[]){
   Operation2 op=new Operation2();

   System.out.println("before change "+op.data);
   op.change(op);//passing object
   System.out.println("after change "+op.data);

 }
}
```

- ***Output***

  before change 50
  after change 150

Lecture 9:Static Variables & Methods, Nested & Inner Classes

9.1. static keyword in Java:

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

9.1.1. Java static variable:

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

- The static variable can be 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.
- The static variable gets memory only once in class area at the time of class loading.

9.1.1.1. Advantage of static variable:

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

9.1.1.2. Understanding problem without static variable:

```
class Student{
   int rollno;
   String name;
```

```
   String college="ITS";
}
```

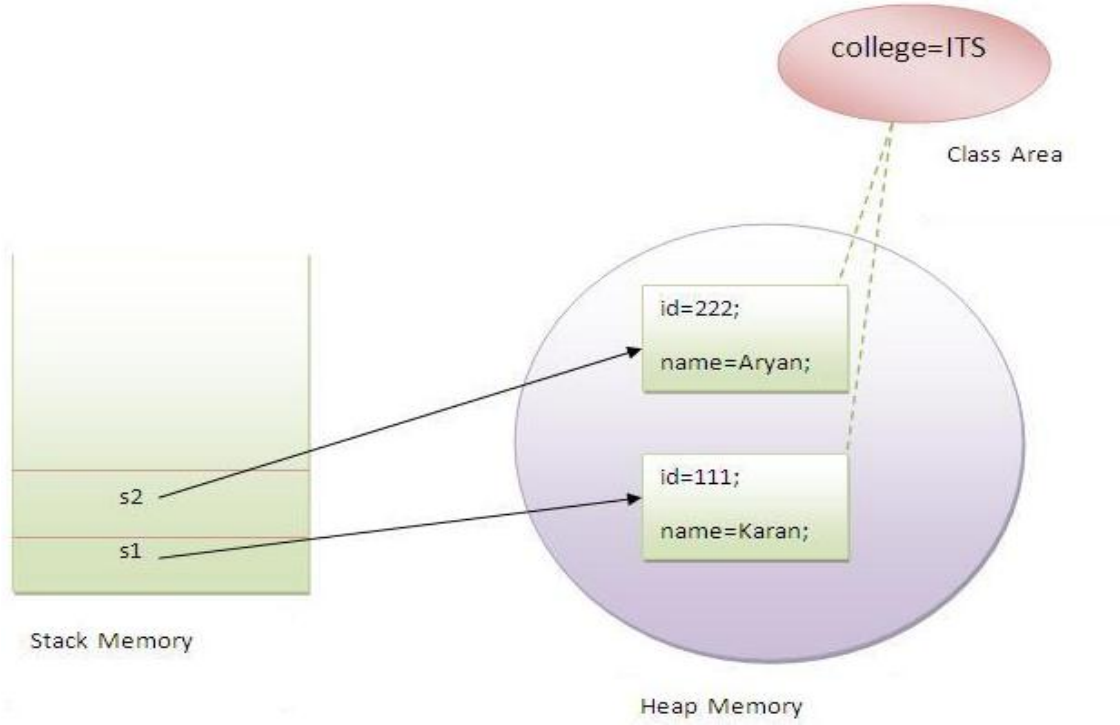
Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static,this field will get memory only once.

- ***Example of static variable***

```
//Program of static variable

class Student8{
  int rollno;
  String name;
  static String college ="ITS";

  Student8(int r,String n){
  rollno = r;
  name = n;
  }
 void display (){System.out.println(rollno+" "+name+" "+college);}

 public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");

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

- ***Output***

```
111 Karan ITS
222 Aryan ITS
```



9.1.2. Java static method:

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

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

- ***Example of static method***

```
//Program of changing the common property of all objects(static field).

class Student9{
    int rollno;
    String name;
    static String college = "ITS";

    static void change(){
    college = "BBDIT";
    }

    Student9(int r, String n){
```

```
    rollno = r;
    name = n;
    }

    void display (){System.out.println(rollno+" "+name+" "+college);}

   public static void main(String args[]){
   Student9.change();

   Student9 s1 = new Student9 (111,"Karan");
   Student9 s2 = new Student9 (222,"Aryan");
   Student9 s3 = new Student9 (333,"Sonoo");

   s1.display();
   s2.display();
   s3.display();
   }
}
```

- ***Output***

  111 Karan BBDIT
  222 Aryan BBDIT
  333 Sonoo BBDIT

9.1.2.1. Restrictions for static method:

There are two main restrictions for the static method. They are:

1. The static method cannot use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

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

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

```
 }
}
```

- ***Output***

  Compile Time Error

9.1.3. Java static block:

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

- ***Example of static block***

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

- ***Output***

  static block is invoked
  Hello main

9.2. Nested & Inner Classes:

9.2.1. Nested Classes:

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the nested class, and the class that holds the inner class is called the outer class.

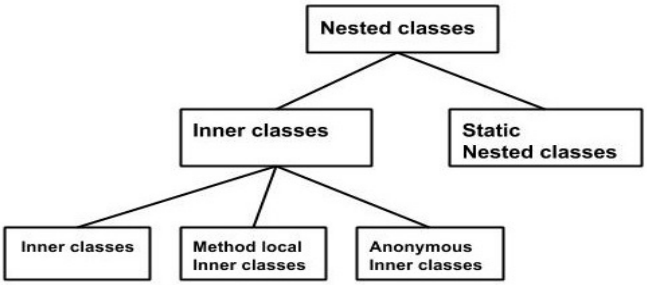
- ***Syntax***

  Following is the syntax to write a nested class. Here, the class Outer_Demo is the outer class and the class Inner_Demo is the nested class.

```
class Outer_Demo {
  class Nested_Demo {
  }
}
```

Nested classes are divided into two types –

- Non-static nested classes – These are the non-static members of a class.
- Static nested classes – These are the static members of a class.



9.2.2. Inner Classes (Non-static Nested Classes):

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are –

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

9.2.2.1. Inner Class:

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

- ***Example***

```
class Outer_Demo {
  int num;

  // inner class
  private class Inner_Demo {
    public void print() {
      System.out.println("This is an inner class");
    }
  }

  // Accessing he inner class from the method within
  void display_Inner() {
    Inner_Demo inner = new Inner_Demo();
    inner.print();
  }
}

public class My_class {

  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();

    // Accessing the display_Inner() method.
    outer.display_Inner();
  }
}
```

Here you can observe that Outer_Demo is the outer class, Inner_Demo is the inner class, display_Inner() is the method inside which we are instantiating the inner class, and this method is invoked from the main method.

If you compile and execute the above program, you will get the following result –

- ***Output***

This is an inner class.

9.2.2.1.1. Accessing the Private Members:

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, getValue(), and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

The following program shows how to access the private members of a class using inner class.

- ***Example***

```
class Outer_Demo {
  // private variable of the outer class
  private int num = 175;

  // inner class
  public class Inner_Demo {
    public int getNum() {
      System.out.println("This is the getnum method of the inner class");
      return num;
    }
  }
}

public class My_class2 {
```

```
  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();

    // Instantiating the inner class
    Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
    System.out.println(inner.getNum());
  }
}
```

If you compile and execute the above program, you will get the following result –

- ***Output***

This is the getnum method of the inner class: 175

9.2.2.2. Method-local Inner Class:

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

- ***Example***

```
public class Outerclass {
  // instance method of the outer class
  void my_Method() {
    int num = 23;

    // method-local inner class
    class MethodInner_Demo {
      public void print() {
        System.out.println("This is method inner class "+num);
      }
    } // end of inner class

    // Accessing the inner class
```

```
    MethodInner_Demo inner = new MethodInner_Demo();
    inner.print();
  }

  public static void main(String args[]) {
    Outerclass outer = new Outerclass();
    outer.my_Method();
  }
}
```

If you compile and execute the above program, you will get the following result –

- ***Output***

```
This is method inner class 23
```

9.2.2.3. Anonymous Inner Class:

An inner class declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows –

- ***Syntax***

```
AnonymousInner an_inner = new AnonymousInner() {
  public void my_method() {
    ........
    ........
  }
};
```

The following program shows how to override the method of a class using anonymous inner class.

- ***Example***

```
abstract class AnonymousInner {
  public abstract void mymethod();
```

```
}

public class Outer_class {

  public static void main(String args[]) {
    AnonymousInner inner = new AnonymousInner() {
      public void mymethod() {
        System.out.println("This is an example of anonymous inner class");
      }
    };
    inner.mymethod();
  }
}
```

If you compile and execute the above program, you will get the following result –

- ***Output***

  This is an example of anonymous inner class

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

9.2.3. Static Nested Class:

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows –

- Syntax

```
class MyOuter {
  static class Nested_Demo {
  }
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

- ***Example***

```
public class Outer {
  static class Nested_Demo {
    public void my_method() {
      System.out.println("This is my nested class");
    }
  }

  public static void main(String args[]) {
    Outer.Nested_Demo nested = new Outer.Nested_Demo();
    nested.my_method();
  }
}
```

If you compile and execute the above program, you will get the following result –

- ***Output***

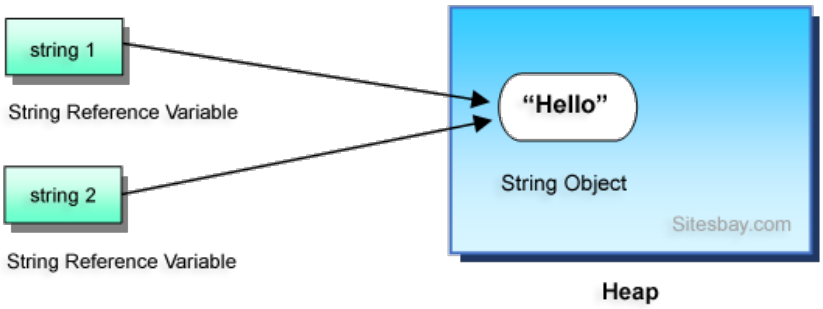
```
This is my nested class
```

MODULE 3: BASIC STRING HANDLING & I/O

Lecture 1: Basic String handling Concepts and Methods of String Class

1.1. String Handling in Java:

The basic aim of String Handling concept is storing the string data in the main memory (RAM), manipulating the data of the String, and retrieving the part of the String etc. String Handling provides a lot of concepts that can be performed on a string such as concatenation of string, comparison of string, find sub string etc.



Java String contains an immutable sequence of Unicode characters. Java String is different from string in C or C++, where (in C or C++) string is simply an array of char. String class is encapsulated under java.lang package.

1.1.1. Immutable class in Java:

Immutable class means that once an object is created, we cannot change its content. In Java, String, Integer, Byte, Short, Float, Double and all other wrapper classes are immutable.

- ***Example of Immutable class in Java***

```
// An immutable class
public final class Student
{
   final String name;
   final int roll_no;

   public Student(String name, int roll_no)
   {
      this.name = name;
      this.regNo = roll_no;
   }
   public String getName()
   {
      return name;
   }
   public int getRollNo()
   {
      return roll_no;
   }
```

```
}

// Driver class
class Result
{
   public static void main(String args[])
   {
      Student s = new Student("Hitesh", 18);
      System.out.println(s.name);
      System.out.println(s.roll_no);
   }
}
```

1.1.2. Character:

It is an identifier enclosed within single quotes (' ').
Example: 'A', '$', 'p'

1.1.3. String:

String is a sequence of characters enclosed within double quotes (" ").
Example: "Java Programming".

In java programming to store the character data we have a fundamental datatype called char. Similarly to store the string data and to perform various operation on String data, we have three predefined classes they are:

- String
- StringBuffer
- StringBuilder

1.1.3.1. String Class in Java:

It is a predefined class in java.lang package that can be used to handle the String. String class is immutable that means its content cannot be changed at the time of execution of program.

String class object is immutable that means when we create an object of String class it never gets changed in the existing object.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s=new String("java");
s.concat("software");
System.out.println(s);
}
}
```

- ***Output***

```
Java
```

- ***Explanation***
  Here we cannot change the object of String class so output is only java not java software.

1.1.3.1.1. Methods of String class:

1.1.3.1.1.1. length():

This method is used to get the number of character of any string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
int l;
String s=new String("Java");
l=s.length();
System.out.println("Length: "+l);
```

```
}
}
```

- ***Output***

  Length: 4

1.1.3.1.1.2. charAt():

This method is used to get the character at a given index value.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
char c;
String s=new String("Java");
c=s.charAt(2);
System.out.println("Character: "+c);
}
}
```

- ***Output***

  Character: v

1.1.3.1.1.3. compareTo():

This method is used to compare two strings by taking unicode values. It returns 0 if the strings are same otherwise returns +ve or -ve integer values.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
```

```
String s1="Hitesh";
String s2="Raddy";
int i;
i=s1.compareTo(s2);
if(i==0)
{
System.out.println("Strings are same");
}
else
{
System.out.println("Strings are not same");
}
}
}
```

- ***Output***

  Strings are not same

1.1.3.1.1.4. compareToIgnoreCase():

This method is a case insensitive method, which is used to compare two strings similar to compareTo().

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1="Hitesh";
String s2="HITESH";
int i;
i=s1.compareToIgnoreCase(s2);
if(i==0)
{
System.out.println("Strings are same");
}
```

```
else
{
System.out.println("Strings are not same");
}
}
}
```

- ***Output***

```
Strings are same
```

1.1.3.1.1.5. equals():

This method is used to compare two strings. It returns true if strings are same otherwise returns false. It is a case sensitive method.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1="Hitesh";
String s2="Raddy";
String s3="Hitesh";
System.out.println("Compare String: "+s1.equals(s2));
System.out.println("Compare String: "+s1.equals(s3));
}
}
```

- ***Output***

```
Compare String: false
Compare String: true
```

1.1.3.1.1.6. equalsIgnoreCase():

This method is a case insensitive method. It returns true if the contents of both the strings are same otherwise returns false.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1="Hitesh";
String s2="HITESH";
String s3="Raddy";
System.out.println("Compare String: "+s1.equalsIgnoreCase(s2));
System.out.println("Compare String: "+s1.equalsIgnoreCase(s3));
}
}
```

- ***Output***

```
Compare String: true
Compare String: false
```

1.1.3.1.1.7. indexOf():

This method is used to find the index value of a given string. It always gives the starting index value of first occurrence of string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s="Java is programming language";
System.out.println(s.indexOf("programming"));
}
}
```

- ***Output***

  8

1.1.3.1.1.8. subString():

This method is used to get the part of a given string.

- ***Example1***

```
class StringHandling
{
public static void main(String arg[])
{
String s="Java is programming language";
System.out.println(s.substring(8)); // 8 is starting index
}
}
```

- ***Output***

  programming language

- ***Example2***

```
class StringHandling
{
public static void main(String arg[])
{
String s="Java is programming language";
System.out.println(s.substring(8, 12));
}
}
```

- ***Output***

  prog

1.1.3.1.1.9. lastIndexOf():

This method used to return the starting index value of last occurence of the given string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1="Java is programming language";
String s2="Java is good programming language";
System.out.println(s1.lastIndexOf("programming"));
System.out.println(s2.lastIndexOf("programming"));
}
}
```

- ***Output***

```
8
13
```

Lecture 2: Methods of String Class (Contd.) and Methods of StringBuffer Class

2.1. toUpperCase():

This method is used to convert lower case string into upper case.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s="Java";
System.out.println("String: "+s.toUpperCase());
}
```

```
}
```

- ***Output***

  String: JAVA

2.2. toLowerCase():

This method is used to convert lower case string into upper case.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s="JAVA";
System.out.println("String: "+s.toLowerCase());
}
}
```

- ***Output***

  String: java

2.3. trim():

This method removes space which are available before starting of the string and after ending of the string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s="    Java is programming language    ";
System.out.println(s.trim());
}
```

```
}
```

- ***Output***

  Java is programming language

2.4. startsWith():

This method returns true if the string is started with the given string within the () of startsWith(), otherwise it returns false.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s="Java is programming language";
System.out.println(s.startsWith("Java"));
}
}
```

- ***Output***

  true

2.5. endsWith():

This method returns true if the string is ended with the given string within the () of endsWith(),otherwise it returns false.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s="Java is programming language";
```

```
System.out.println(s.endsWith("language"));
}
}
```

- ***Output***

true

2.6. concat():

This method is used to combine two string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1="Hitesh";
String s2="Raddy";
System.out.println("Combined String: "+s1.concat(s2));
}
}
```

- ***Output***

Combined String: HiteshRaddy

2.7. toCharArray():

This method converts the given string into a sequence of characters. The returned array length is equal to the length of the string.It returns a newly allocated character array.

- ***Example***

```
class Gfg {
   public static void main(String args[])
   {
      String s = "HelloGNIT";
```

```
        char[] gfg = s.toCharArray();
        for (int i = 0; i < gfg.length; i++) {
           System.out.println(gfg[i]);
        }
     }
}
```

- ***Output***

```
H
e
l
l
o
G
N
I
T
```

2.8. toString():

If you want to represent any object as a string, toString() method comes into existence.
The toString() method returns the string representation of the object.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

Note –By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

- ***Understanding problem without toString() method***

  Let's see the simple code that prints reference.

```
class Student{
 int rollno;
 String name;
```

```
String city;

Student(int rollno, String name, String city){
this.rollno=rollno;
this.name=name;
this.city=city;
}

public static void main(String args[]){
  Student s1=new Student(101,"Raj","lucknow");
  Student s2=new Student(102,"Vijay","ghaziabad");

  System.out.println(s1);//compiler writes here s1.toString()
  System.out.println(s2);//compiler writes here s2.toString()
}
}
```

- ***Output***

```
Student@1fee6fc
Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }
```

```
 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```

- ***Output***

  101 Raj lucknow
  102 Vijay Ghaziabad

2.9. valueOf():

This method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

- ***Signature***

  The signature or syntax of string valueOf() method is given below:

  public static String valueOf(boolean b)
  public static String valueOf(char c)
  public static String valueOf(char[] c)
  public static String valueOf(int i)
  public static String valueOf(long l)
  public static String valueOf(float f)
  public static String valueOf(double d)
  public static String valueOf(Object o)

- ***Returns***

  string representation of given value.

- ***Example***

```
public class StringValueOfExample{
public static void main(String args[]){
int value=30;
String s1=String.valueOf(value);
System.out.println(s1+10);//concatenating string with 10
}}
```

- ***Output***

```
3010
```

2.10. StringBuffer Class in Java:

It is a predefined class in java.lang package that can be used to handle the String, whose object is mutable that means content can be modified.

StringBuffer class works with thread safe mechanism that means multiple threads are not allowed simultaneously to perform operation of StringBuffer.

StringBuffer class object is mutable that means when we create an object of StringBuffer class it can be changed.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
StringBuffer sb=new StringBuffer("java");
sb.append("software");
System.out.println(sb);
}
}
```

- ***Output***

Javasoftware

- ***Explanation***

  Here we can change in the existing object of StringBuffer class so output is javasoftware.

2.10.1. Difference between String and StringBuffer:

String	StringBuffer
The data which is enclosed within double quotes (" ") is by default treated as String class.	The data which is enclosed within double quotes (" ") is not by default treated as StringBuffer class.
String class object is immutable.	StringBuffer class object is mutable.
When we create an object of String class by default no additional character memory space is created.	When we create an object of StringBuffer class by default we get 16 additional character memory space.

2.10.2. Difference between String and StringBuffer:

- Both of them are public and final, so they never participate in inheritance that is IS-A relationship is not possible.
- We cannot override the methods of String and StringBuffer.

2.10.3. Methods of StringBuffer class:

2.10.3.1. append():

This method is used to add a new string at the end of original string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
```

```
StringBuffer sb=new StringBuffer("java is easy");
System.out.println(sb.append(" to learn"));
}
}
```

- ***Output***

  java is easy to learn

2.10.3.2. delete():

This method is used to delete a string from the given string based on index value.

- Example

```
class StringHandling
{
public static void main(String arg[])
{
StringBuffer sb=new StringBuffer("java is easy to learn");
StringBuffer s;
s=sb.delete(8, 13);
System.out.println(sb);
}
}
```

- ***Output***

  java is to learn

2.10.3.3. deleteCharAt():

This method is used to delete a character at given index value.

- ***Example***

```
class StringHandling
{
```

```
public static void main(String arg[])
{
StringBuffer sb=new StringBuffer("java");
System.out.println(sb.deleteCharAt(3));
}
}
```

- ***Output***

```
jav
```

2.10.3.4. capacity():

This method returns the current capacity. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

- ***Example***

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

    StringBuffer buff = new StringBuffer("ObjectOriented");

    // returns the current capacity of the String buffer i.e. 16 + 14
    System.out.println("capacity = " + buff.capacity());

    buff = new StringBuffer(" ");

    // returns the current capacity of the String buffer i.e. 16 + 1
    System.out.println("capacity = " + buff.capacity());
  }
}
```

- ***Output***

```
capacity = 30
capacity = 17
```

2.10.3.5. charAt():

This method returns the char value in this sequence at the specified index. The first char value is at index 0, the next at index 1, and so on, as in array indexing.

The index argument must be greater than or equal to 0, and less than the length of this sequence.

Note – If index is negative or greater than or equal to length(), then this method throws **IndexOutOfBoundsException**.

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff = new StringBuffer("Object Oriented");
    System.out.println("buffer = " + buff);

    // returns the char at index 4
    System.out.println("character = " + buff.charAt(4));

    buff = new StringBuffer("amrood admin ");
    System.out.println("buffer = " + buff);

    // returns the char at index 6, whitespace gets printed here
    System.out.println("character = " + buff.charAt(6));
  }
}
```

- ***Output***

```
buffer = Object Oriented
character = c
buffer = amrood admin
character =
```

Lecture 3: Methods of StringBuffer Class (Contd.)

3.1. ensureCapacity():

This method ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of –

- The minimumCapacity argument.
- Twice the old capacity, plus 2.

If the minimumCapacity argument is nonpositive, this method takes no action and simply returns.

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff1 = new StringBuffer("tuts point");
    System.out.println("buffer1 = " + buff1);

    // returns the current capacity of the string buffer 1
    System.out.println("Old Capacity = " + buff1.capacity());

    /* increases the capacity, as needed, to the specified amount in the
      given string buffer object */

    // returns twice the capacity plus 2
    buff1.ensureCapacity(28);
    System.out.println("New Capacity = " + buff1.capacity());

    StringBuffer buff2 = new StringBuffer("compile online");
    System.out.println("buffer2 = " + buff2);

    // returns the current capacity of string buffer 2
    System.out.println("Old Capacity = " + buff2.capacity());
```

```
        /* returns the old capacity as the capacity ensured is less than
          the old capacity */
        buff2.ensureCapacity(29);
        System.out.println("New Capacity = " + buff2.capacity());
      }
    }
```

- ***Output***

```
buffer1 = tuts point
Old Capacity = 26
New Capacity = 54
buffer2 = compile online
Old Capacity = 30
New Capacity = 30
```

3.2. getChars():

This method copy the characters from this sequence into the destination character array **dst**.

The first character to be copied is at index **srcBegin**. The last character to be copied is at index **srcEnd - 1**. The total number of characters to be copied is **srcEnd - srcBegin**. The characters are copied into the subarray of **dst** starting at index **dstBegin** and ending at index: **dstbegin + (srcEnd-srcBegin) – 1**

- ***Exception***
    - **NullPointerException –** if dst is null
    - **IndexOutOfBoundsException –** this is thrown if any of the following is true –
        srcBegin is negative
        dstBegin is negative
        the srcBegin argument is greater than the srcEnd argument.
        srcEnd is greater than this.length().
        dstBegin + srcEnd - srcBegin is greater than dst.length

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff = new StringBuffer("java programming");
    System.out.println("buffer = " + buff);

    // char array
    char[] chArr = new char[]{'t','u','t','o','r','i','a','l','s'};

    // copy the chars from index 5 to index 10 into subarray of chArr
    // the offset into destination subarray is set to 3

    buff.getChars(5, 10, chArr, 3);

    // print character array
    System.out.println(chArr);
  }
}
```

- ***Output***

  buffer = java programming
  tutprogrs

3.3. indexOf():

The java.lang.StringBuffer.indexOf(String str, int fromIndex) method returns the index within this string of the first occurrence of the specified substring, starting at the specified index.The **fromIndex** argument is the index from which to start the search.

Note – This method throws NullPointerException if str is null.

- ***Example***

```
public class StringBufferDemo {

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

```
        StringBuffer buff = new StringBuffer("programming language");
        System.out.println("buffer = " + buff);

        // returns the index of the specified substring
        System.out.println("Index of substring = " + buff.indexOf("age"));

        /* returns the index of the specified substring, starting at
           the specified index */
        System.out.println("Index of substring = " + buff.indexOf("am",2));

        /* returns -1 as the substring is not found starting at the
           specified index */
        System.out.println("Index of substring = " + buff.indexOf("am",10));
      }
    }
```

- ***Output***

```
buffer = programming language
Index of substring = 17
Index of substring = 5
Index of substring = -1
```

3.4. insert():

This method is used to insert either string or character or integer or real constant or boolean value at a specific index value of the given string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
StringBuffer sb=new StringBuffer("this is my java code");
System.out.println(sb.insert(11, "first "));
}
```

```
}
```

- ***Output***

  this is my first java code

3.5. length():

This method returns the length (character count) of the sequence of characters currently represented by this object.

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff = new StringBuffer("Tutorials");

    // printing the length of stringbuffer
    System.out.println("length = " + buff.length());

    buff = new StringBuffer("");

    // printing the length of empty stringbuffer
    System.out.println("length = " + buff.length());
  }
}
```

- ***Output***

```
length = 9
length = 0
```

3.6. setCharAt():

This method sets the character at the specified **index** to **ch**.This sequence is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character **ch** at position **index**.

Note – This method throws **IndexOutOfBoundsException**if index is negative or greater than or equal to length().

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff = new StringBuffer("AMIT");
    System.out.println("buffer = " + buff);

    // character at index 3
    System.out.println("character at index 3 = " + buff.charAt(3));

    // set character at index 3
    buff.setCharAt(3, 'L');

    System.out.println("After Set, buffer = " + buff);

    // character at index 3
    System.out.println("character at index 3 = " + buff.charAt(3));
  }
}
```

- ***Output***

```
buffer = AMIT
character at index 3 = T
After Set, buffer = AMIL
character at index 3 = L
```

3.7. setLength():

This method sets the length of the character sequence. The sequence is changed to a new character sequence whose length is specified by the argument.

If the **newLength** argument is greater than or equal to the current length, sufficient null characters ('\u0000') are appended so that length becomes the **newLength** argument.

Note – This method throws **IndexOutOfBoundsException** if the newLength argument is negative.

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff = new StringBuffer("tutorials");
    System.out.println("buffer1 = " + buff);

    // length of stringbuffer
    System.out.println("length = " + buff.length());

    // set the length of stringbuffer to 5
    buff.setLength(5);

    // print new stringbuffer value after changing length
    System.out.println("buffer2 = " + buff);

    // length of stringbuffer after changing length
    System.out.println("length = " + buff.length());
  }
}
```

- ***Output***

```
buffer1 = tutorials
length = 9
buffer2 = tutor
length = 5
```

3.8. substring():

The java.lang.StringBuffer.substring(int start) method returns a new String that contains a subsequence of characters currently contained in this character sequence. The substring begins at the specified index, **start** and extends to the end of this sequence.

Note – This method throws **StringIndexOutOfBoundsException** if start is less than zero, or greater than the length of this object.

- ***Example***

```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer buff = new StringBuffer("tutorials");
    System.out.println("buffer = " + buff);

    // prints substring from index 2
    System.out.println("substring = "+ buff.substring(2));
  }
}
```

- ***Output***

```
buffer = tutorials
substring = torials
```

3.9. toString():

This method is used to convert mutable string values into immutable string.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
```

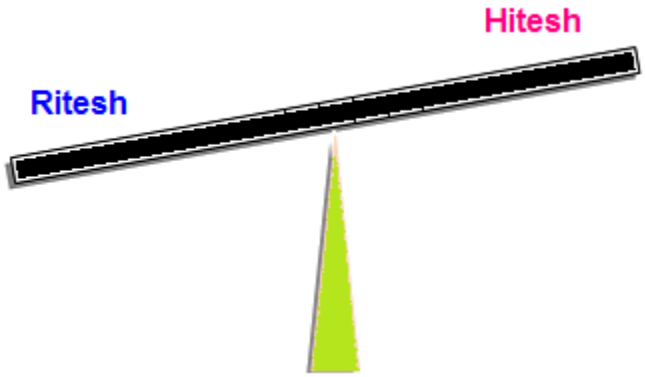
```
{
StringBuffer sb=new StringBuffer("java");
String s=sb.toString();
System.out.println(s);
s.concat("code");
}
}
```

- ***Output***

  java

3.10. String Compare in Java:

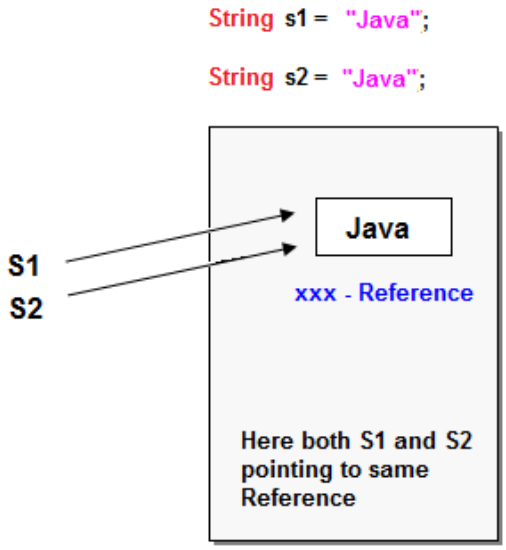
There are three ways to compare string object in java:



- By equals() method
- By == operator
- By compreTo() method

3.10.1. equals() Method in Java:

equals() method always used to compare contents of both source and destination string. It returns true if both strings are same in meaning and case otherwise it returns false. It is a case sensitive method.



- ***Example***

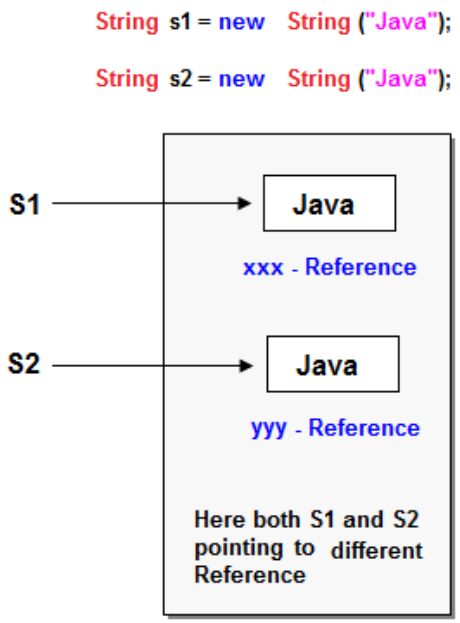
```
class StringHandling
{
public static void main(String arg[])
{
String s1="Hitesh";
String s2="Raddy";
String s3="Hitesh";
System.out.println("Compare String: "+s1.equals(s2));
System.out.println("Compare String: "+s1.equals(s3));
}
}
```

- ***Output***

  Compare String: false
  Compare String: true

3.10.2. == or Double Equals to Operator in Java:

== Operator is always used for comparing references of both source and destination objects but not their contents.



- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1=new String("java");
String s2=new String("java");
if(s1==s2)
{
System.out.println("Strings are same");
}
else
{
System.out.println("Strings are not same");
}
}
}
```

- ***Output***

Strings are not same

3.10.3. compareTo() Method in Java:

comapreTo() method can be used to compare two strings by taking unicode values. It returns 0 if the strings are same otherwise returns either +ve or -ve integer.

- ***Example***

```
class StringHandling
{
public static void main(String arg[])
{
String s1="Hitesh";
String s2="Raddy";
int i;
i=s1.compareTo(s2);
if(i==0)
{
System.out.println("Strings are same");
}
else
{
System.out.println("Strings are not same");
}
}
}
```

- ***Output***

```
Strings are not same
```

Lecture 4: Command line arguments, Basics of I/O operations

4.1. Using Command-Line Arguments:

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

- ***Example***
  The following program displays all of the command-line arguments that it is called with –

```
public class CommandLine {

  public static void main(String args[]) {
    for(int i = 0; i<args.length; i++) {
      System.out.println("args[" + i + "]: " +  args[i]);
    }
  }
}
```

  Try executing this program as shown here –

```
java CommandLine this is a command line 200 -100
```

- ***Output***

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

4.2. IO Stream:

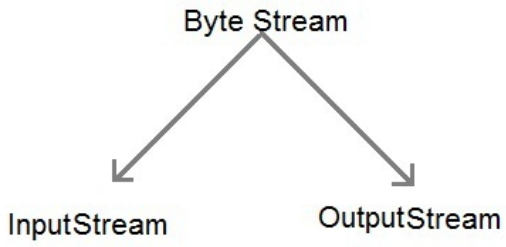
Java performs I/O through Streams. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Java encapsulates Stream under java.io package. Java defines two types of streams. They are,

1. Byte Stream: It provides a convenient means for handling input and output of byte.
2. Character Stream: It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

4.2.1. Byte Stream Classes:

Byte stream is defined by using two abstract classes at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

4.2.1.1. Some important Byte stream classes:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

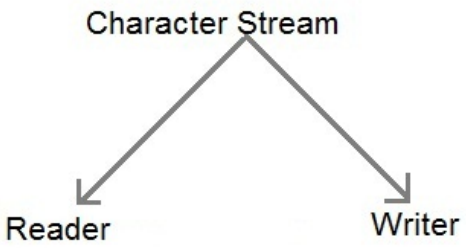
These classes define several key methods. Two most important are

read() : reads byte of data.

write() : Writes byte of data.

4.2.2. Character Stream Classes:

Character stream is also defined by using two abstract classes at the top of hierarchy, they are Reader and Writer.



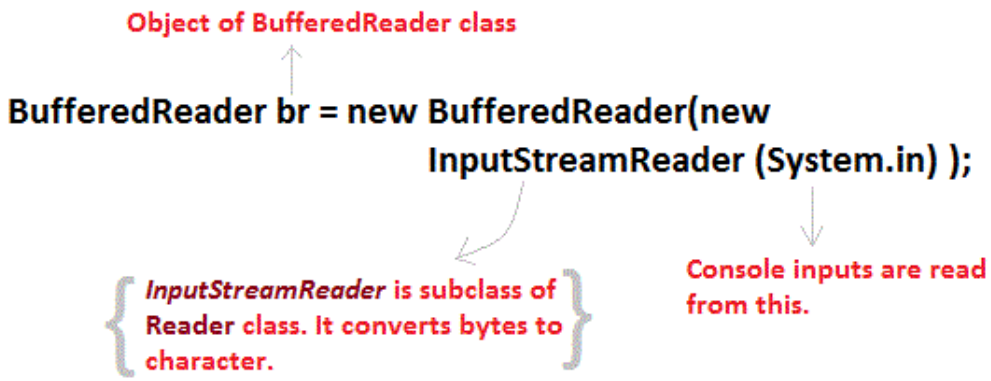
These two abstract classes have several concrete classes that handle unicode character.

4.2.2.1. Some important Character stream classes:

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

4.2.3. Reading Console Input:

We use the object of BufferedReader class to take inputs from the keyboard.



4.2.4. Reading Characters:

read() method is used with BufferedReader object to read characters. As this function returns integer type value, we need to use typecasting to convert it into char type.

- ***Syntax***

  int read() throws IOException

Below is a simple example explaining character input –

```
class CharRead
{
 public static void main( String args[])
 {
  BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
  char c = (char)br.read();      //Reading character
 }
}
```

4.2.5. Reading Strings:

To read string we have to use readLine() function with BufferedReader class's object.

- ***Syntax***

  String readLine() throws IOException

4.2.6. Program to take String input from Keyboard in Java:

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();         //Reading String
  System.out.println(text);
 }
}
```

4.2.7. Program to read from a file using BufferedReader class:

```
import java. Io *;
class ReadTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   BufferedReader br = new BufferedReader(new FileReader(fl)) ;
   String str;
   while ((str=br.readLine())!=null)
   {
    System.out.println(str);
   }
   br.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
}
```

4.2.8. Program to write to a File using FileWriter class:

```
import java. Io *;
class WriteTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   String str="Write this string to my file";
   FileWriter fw = new FileWriter(fl) ;
   fw.write(str);
   fw.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
}
```

4.3. Java Scanner class:

There are various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter that is whitespace bydefault. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

4.3.1. Commonly used methods of Scanner class:

There 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.

4.3.2. Java Scanner Example to get input from console:

Let's see the simple example of the Java Scanner class which reads the int, string and double value as an input –

```
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);
   sc.close();
 }
}
```

- ***Output***

  Enter your rollno
  111
  Enter your name
  Ratan
  Enter

450000
Rollno:111 name:Ratan fee:450000

4.3.3. Java Scanner Example with delimiter:

Let's see the example of Scanner class with delimiter. The \s represents whitespace.

```
import java.util.*;
public class ScannerTest2{
public static void main(String args[]){
   String input = "10 tea 20 coffee 30 tea buiscuits";
   Scanner s = new Scanner(input).useDelimiter("\\s");
   System.out.println(s.nextInt());
   System.out.println(s.next());
   System.out.println(s.nextInt());
   System.out.println(s.next());
   s.close();
}}
```

- ***Output***

10
tea
20
coffee

4.4. Difference between Scanner and BufferReader Class in Java:

java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters

Following are differences between above two.

Issue with Scanner when nextLine() is used after nextXXX()

```
// Code using Scanner Class
import java.util.Scanner;
class Differ
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        System.out.println("Enter an integer");
        int a = scn.nextInt();
        System.out.println("Enter a String");
        String b = scn.nextLine();
        System.out.printf("You have entered:- "
              + a + " " + "and name as " + b);
    }
}
```

- ***Input***

  50
  Geek

- ***Output***

  Enter an integer
  Enter a String
  You have entered:- 50 and name as

Let us try the same using Buffer class and same Input

```
// Code using Buffer Class
import java.io.*;
class Differ
{
    public static void main(String args[])
            throws IOException
    {
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("Enter an integer");
```

```
        int a = Integer.parseInt(br.readLine());
        System.out.println("Enter a String");
        String b = br.readLine();
        System.out.printf("You have entered:- " + a +
                          " and name as " + b);
    }
}
```

- ***Input***

  50
  Geek

- ***Output***

  Enter an integer
  Enter a String
  you have entered:- 50 and name as Geek

In Scanner class if we call nextLine() method after any one of the seven nextXXX() method then the nextLine() doesn't not read values from console and cursor will not come into console it will skip that step. The nextXXX() methods are nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().

In BufferReader class there is no such type of problem. This problem occurs only for Scanner class, due to nextXXX() methods ignore newline character and nextLine() only reads till first newline character. If we use one more call of nextLine() method between nextXXX() and nextLine(), then this problem will not occur because nextLine() will consume the newline character. See this for the corrected program. This problem is same as scanf() followed by gets() in C/C++.

- ***Other differences***

  - BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
  - BufferedReader has significantly larger buffer memory than Scanner.
  - The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.

- BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

MODULE 4: INHERITANCE AND JAVA PACKAGES

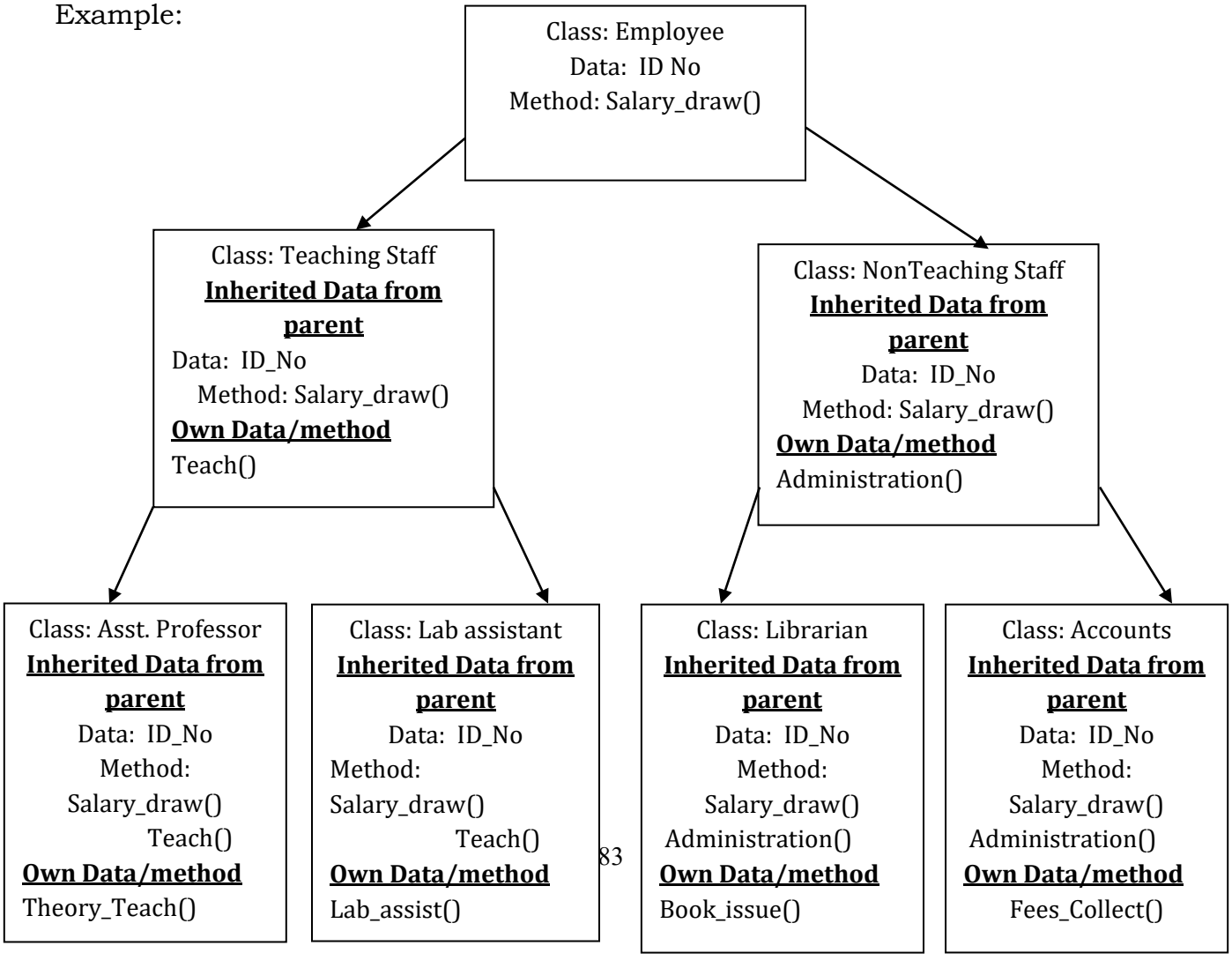
LECTURE 1: Inheritance - Definition, Advantages, Different Types of Inheritance and their Implementation.

4.1. Definition of Inheritance: Inheritance is a process through which some physical appearance (Data) and some logical behavior (method) related properties of parent class can be acquired by it's child class without any modification.

4.2. Advantages of inheritance:

In case of inheritance a piece of code (Data as well as method) can be introduced only once in parent class that can be automatically inherited into its child class and Code is reused.

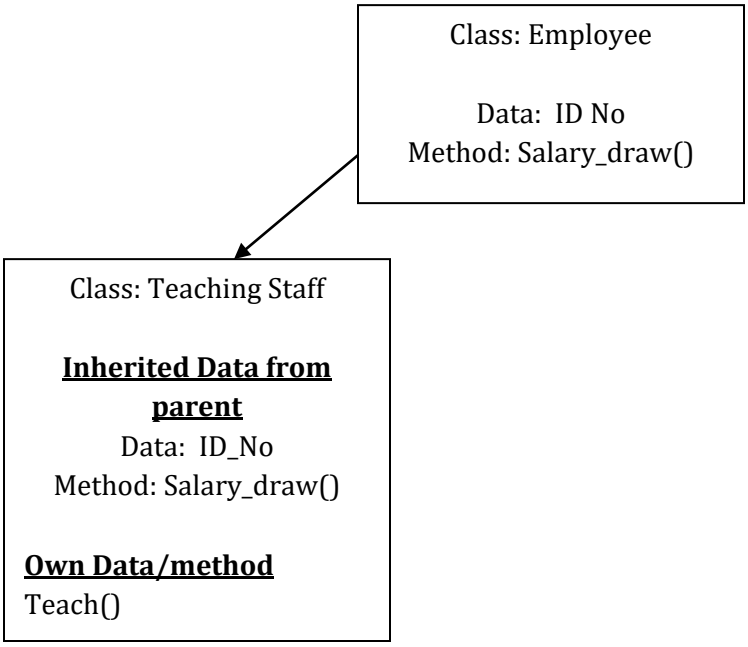
Example:



4.3. *Different types of inheritance and their implementation*

4.3.1. Single Inheritance :

In this inheritance One parent and one child class exists in two different but consecutive levels, and properties acquired from parent class to child class without modification.




```
class Employee
{
   int  ID_No;
   void Salary_draw()
    {
        System.out.println("Every employee must draw their salary");

           }
  }
class Teaching_Staff  extends Employee
{
    void Teach()
    {
        System.out.println("Every teacher can teach their students");
           }
  }
```

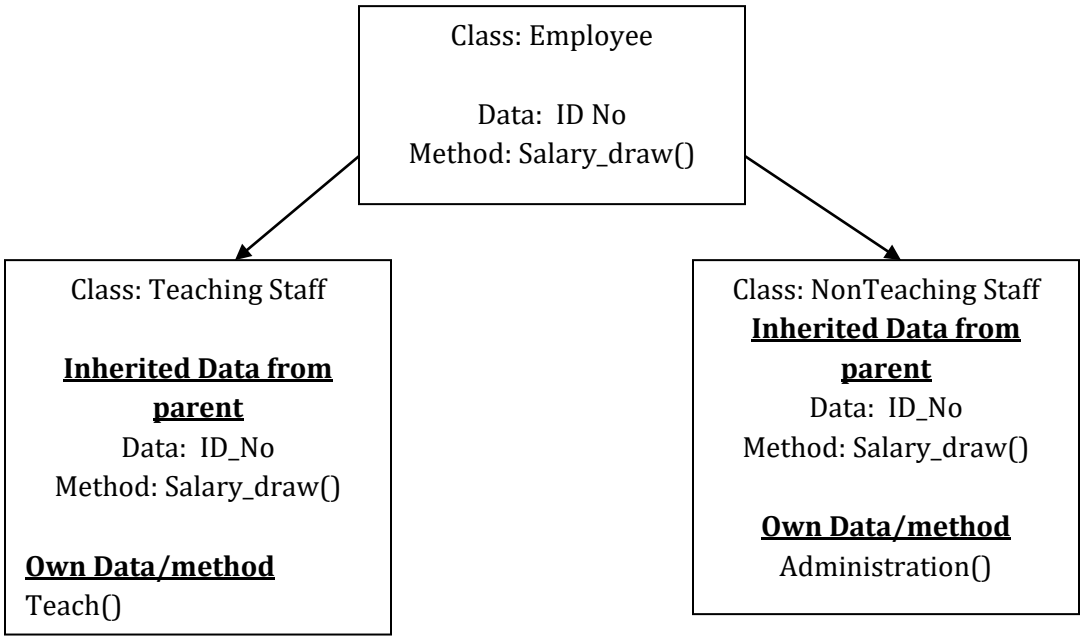
```
class Inherit
 {
   public static void main(String[]  args)
    {
       Teaching_Staff  SM = new Teaching_Staff ();
        SM.ID_No=142;
        SM.Salary_draw();
        SM. Teach();
        System.out.println("ID:="+ SM.ID_No);
         }
}
```

OUTPUT:

Every employee must draw their salary
Every teacher can teach their students
ID:=142

4.3.2 Hierarchical Inheritance:

In this inheritance One parent and more than one child class exists in two different but consecutive levels, and properties acquired from parent class to all child class that exists in same level without any modification.



```
class Employee
{
    int  ID_No;
    void Salary_draw()
     {
         System.out.println("Every employee must draw their salary");
            }
   }
class Teaching_Staff  extends Employee
{
     void Teach()
     {
         System.out.println("Every teacher can teach their students");
            }
   }
class Non_Teaching_Staff  extends Employee
{
   void Administration()
   {
      System.out.println("Every Non_Teaching_Staff  is not a part of admin");
          }
   }
class Inherit
{
   public static void main(String[]  args)
    {
       Teaching_Staff  SM = new Teaching_Staff ();
       Non_Teaching_Staff  SB = new Non_Teaching_Staff ();

        SM.ID_No=142;
        SM.Salary_draw();
        SM. Teach();

        SB.ID_No=212;
        SB.Salary_draw();
        SB. Administration()
```

```
        System.out.println("ID:="+ SB.ID_No);
        }

}
```

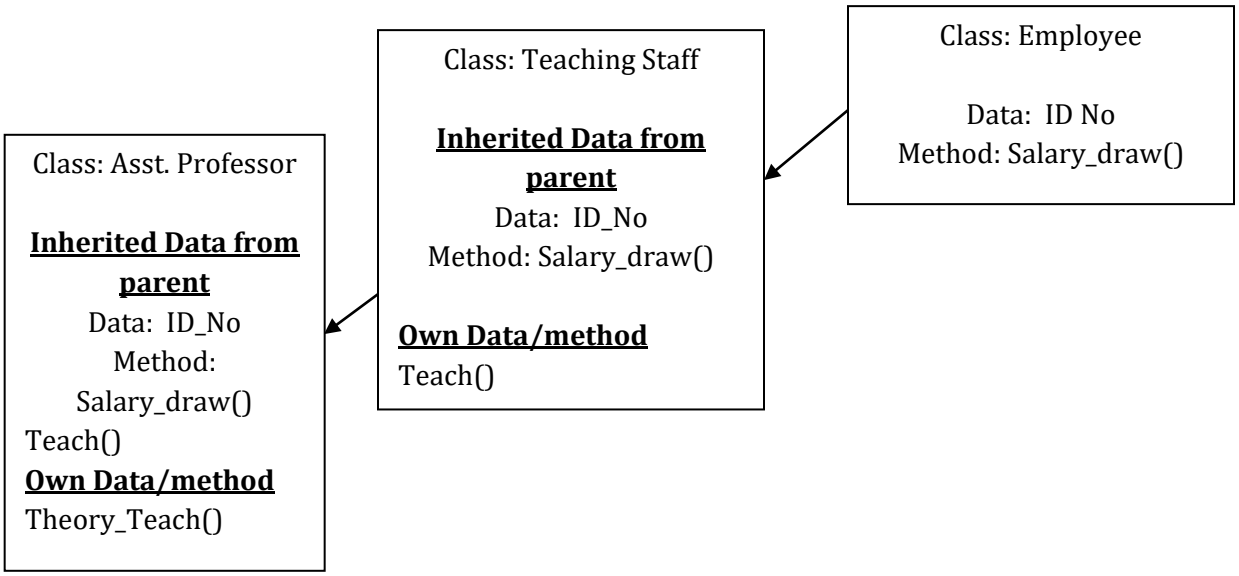
OUTPUT:

Every employee must draw their salary
Every teacher can teach their students
ID:=142

Every employee must draw their salary
Every Non_Teaching_Staff is not a part of admin
ID:=212

4.3.3. Multilevel Inheritance:

In this inheritance one parent single child relationship exists in more than one consecutive levels in chain form, and properties acquired from grand parent class to parent and from parent class into it's child class without any modification.



```
class Employee
{
    int  ID_No;
    void Salary_draw()
     {
         System.out.println("Every employee must draw their salary");
            }
   }
class Teaching_Staff  extends Employee
{
     void Teach()
     {
         System.out.println("Every teacher can teach their students");
            }
   }
class Asst_Professor  extends Teaching_Staff
{
   void Theory_ Teach()
   {
      System.out.println("Asst. Prof  teaches theory subject");

          }
   }
class Inherit
{
   public static void main(String[]  args)
    {
       Asst_Professor  SM = new Asst_Professor();
        SM.ID_No=142;
        SM.Salary_draw();
        SM. Teach();
          SM.Theory_ Teach();
           System.out.println("ID:="+ SM.ID_No);
        }
}
```

OUTPUT:

Every employee must draw their salary
Every teacher can teach their students
Asst. Prof teaches theory subject
ID:=142

4.3.4. Multiple Inheritance:

In this inheritance multiple no of parents exists in single level. Properties of those parent classes can be acquired into the single child class which resided in the recent next level without any modification. Java does not support multiple inheritance. In interface section I can elaborately discuss Multiple Inheritance.

LECTURE 2:super and final keywords, super() method.

4.2.1.Super Keyword:

In case of method overriding we can access parent method rather than same method of child with the help of child class object.

```
class A
{voidabc()
        {System.out.println("Hello GNIT");
                }
}
classB extends A
{  voidabc()
        {System.out.println("Hi GNIT");
                }
        voidpqr()
                {System.out.println("Thanks….");
                        }
        void work()
        {
                super.abc();
                pqr();
                } }

class Test{
public static void main(String args[])
{
        B obj=new B();
```

```
        obj.work();
}
        }
```

OUTPUT:
Hello GNIT
Hi GNIT
Thanks….

4.2.2. Super() method:

Super() method can execute super or parent class constructor where this method exists.

```
class A{
        A()
        {System.out.println("A is created");
                }
        }
class D extends A{
        D(){
        super();
        System.out.println("D is created");
                }
        }
class Test{
        public static void main(String args[])
        {
                D d=new D();
                }
        }
```

OUTPUT:
A is created
D is created

Super() with parameter:

```
class Person
{
int id;
String name;
```

```
Person(intid,String name)
        {
        this.id=id;
        this.name=name;
        }
}
Class Emp extends Person
{
float salary;
Emp(int id,String name,float salary)
        {
                super(id,name);//reusing parent constructor
                this.salary=salary;
        }
void display()
        {System.out.println(id+" "+name+" "+salary);
        }
}

class Test
{
public static void main(String[] args)
        {
                Emp e1=new Emp(11,"Sunil",64000f);
                e1.display();
        }
}
```

OUTPUT:
11 Sunil64000

4.2.3. final Keyword

The final keyword in java is used to restrict the user.

a) It can stop value change when it is used before variable.
b) It can stop method overriding when it is used before method.
c) It can stop inheritance when it is used before class.

Case-1: final keyword used before Variable

```
ClassCycle{

    finalint speed=30;//final variable
```

```
    void run()
        {       speed=30;
                }
    public static void main(String args[])
    {
        Cycleobj=new  Cycle();
        obj.run();
        }
    }//end of class
```

We are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

Case-2: final keyword used before method

```
classCycle{
final void move()
    {
        System.out.println("running");
        }
    }
class Hero extends Cycle{
voidmove()
    {
        System.out.println("Moving with two wheel”);
        }
public static void main(String args[])
{
```

```
Heroobj= new Hero();

obj.run();

  }

}
```

When we are going to assign final keyword before a method of parent class it can't be overridden into the child class.

<u>Case-3: final keyword used before class</u>

```
final class Cycle{

                        void run(){

                        System.out.println("run");

                        }

                }

class hero extends Cycle{

                        void run()

                                {System.out.println("Running");

                                        }

 public static void main(String args[])

{  hero obj= new hero();

        obj.run();

        }

}
```

When we are going to assign final keyword before a class name it can't be inherited means no child class will be created of that particular parent class.

LECTURE 3:Method overriding, Dynamic method dispatch.

4.3.1. Method Overriding:

Method overriding is a process through which a particular method can be introduced in parent class as well as it's child class with same name, same return type, same signature but different implementational logic.

```
class Vehicle
{
   void move()
    {
        System.out.println("Any vehicle can move");
        }
    }
class two_wheeler extends Vehicle
   {
      void move()
       {
           System.out.println("Two wheeler can move with 2 wheel");
        }
    }

class test
{
public static void main(String[]  args)
    {
         two_wheeler  bicycle=new two_wheeler();
         bicycle.move();
     }
}
```

Output:
Two wheeler can move with 2 wheel

Here move() is overrided into parent class Vehicle as well as child class two_wheeler with same name, default empty signature, same return type void but with different internal code. Two move() methods are available into child class (One is it's own and other is acquired from parent). When we create the object of child class the move() of child class can execute rather than parent class. If we want to take the control for execute move() method of parent class or child class as per our wish we can use **Dynamic method dispatch technique.**

4.3.2. Dynamic method dispatch technique:

```
class Vehicle
{
   void move()
    {
        System.out.println("Any vehicle can move");
        }
    }
class two_wheeler extends Vehicle
   {
      void move()
       {
           System.out.println("Two wheeler can move with 2 wheel");
        }
    }

class test
{
public static void main(String[]  args)
    {
         Vehicle obj;
         Vehicle obj1=new Vehicle();
         two_wheeler  bicycle=new two_wheeler();
         obj= bicycle;
         obj.move();
         obj= obj1;
         obj.move();
```

```
    }
}
```

OUTPUT:

Two wheeler can move with 2 wheel.
Any vehicle can move.

Rule-1: Create reference variable of Parent class.
Rule-2: Create object of Parent class.
Rule-3: Create object of Child class.
Rule-4: Assign Parent class object into reference variable of Parent class.
Rule-5: Call overrided method of parent class.
Rule-6: Assign child class object into reference variable of Parent class.
Rule-7: Call overrided method of child class.

LECTURE 4:Abstract classes & methods

4.4.1. Data Abstraction:

Data abstraction is a process through which essential features of an object can be shown instead of unessential implementation details executing on background.

Example:

When you are watching a dance show in Television set, we need to operate the TV remote control only rather than the internal circuitry and it's activity.

4.4.2. Abstract Class& Abstract method:

```
abstract class Shape{

abstract void draw();

}

class circle extends Shape{

        void draw()

        {
```

```
        System.out.println("Draw a shape of circle");
        }
}
Class A {
public static void main(String args[]){
circleobj = new circle();
obj.draw();
}
    }
```

a) Any abstract class must have at least one abstract method without any definition/body. That abstract method must be clearly implemented in it's child. If we can't clearly implement those abstract method of parent class into it's child then the child class also becomes abstract in nature. Any object of abstract class cannot be created.

LECTURE 5: Interface - Definition, Use of Interface.

4.5.1. Interface:

Interface provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface.

Syntax:

```
interface<interface_name>{
  // declare constant fields
 // declare methods that abstract
 }
```

Example program:

```
interfaceDrawable
{
void draw();
```

```
}
//Implementation: by second user
class Rectangle implements Drawable
{
public void draw()
        {System.out.println("drawing rectangle");
        }
}
class Circle implements Drawable
{
public void draw()
        {System.out.println("drawing circle");
        }
}
//Using interface: by third user
class Test{
public static void main(String args[])
{
        Drawable d;
        Circle c=new Circle();
        d=c;
        d.draw();
        Rectangle r=new Rectangle();
        d=r;
        d.draw();
         }
```

```
}
```

4.5.2. Interface inheritance

A class implements interface but one interface extends another interface .

```
interfacePrintdocument{
void print();
}
interface Showdocument extends Printdocument{
void show();
}
class Test implements Showdocument
{
        public void print(){System.out.println("Hello GNIT");}
        public void show(){System.out.println("Welcome");}
        public static void main(String args[]){
        Test obj = new Test();
        obj.print();
        obj.show();
         }
}
```

OUTPUT:

```
Hello GNIT
Welcome
```

LECTURE 6: Multiple inheritance by using Interface.

4.6.1. Multiple Inheritance:

Multiple inheritanceis an inheritance where multiple no of parents exists and same kind of properties of all parent classes can be inherited into a single child. This thing is not directly possible in java.

With the help of interface we can solve this problem.

```
interface Printdocument
        {
        void print();
        }
interface Showdocument{
void print();
}
class Test implements Printdocument, Showdocument
{
public void print()
        {
                System.out.println("Hello");
                }
public static void main(String args[])
{
        Test obj = new Test();
        obj.print();
         }
}
```

Output: Hello

LECTURE 7: Java Packages - Definition, Creation of packages.

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

4.7.2. Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection using different type of access specifier like default, protected, private, public etc.

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

4.7.3. Package Creation:

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

LECTURE 8: Importing packages, member access for packages.

4.8.1. Package import:

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package p;
public class A
{
```

```
 public void print()
        {System.out.println("Hello GNIT");
        }
}

//save by B.java
package p;
public class B
{
 public void show()
        {System.out.println("Hello ")
        }
}

//save by C.java
import  p.*;    [All the class inside package p has been imported in this program.]
 class C
{

public static void main(String args[])
{
  A obj = new A();
  obj.print();
  B obj=new B();
  obj.show();

 }

}
```

Output:
Hello GNIT
Hello

4.8.2. Access Specifier: Access specifier is a keyword through which the accessibility of data or method can be specified.

Private :

```
class A{
private int d=4;
```

```
private void show()
        {System.out.println("Hello GNIT");
                }
}

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

Any private data / method can be accessed from that class only where it can exists.

Public:

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

```
//save as A.java

package p;
public class A{
public void show()
        {System.out.println("Hello GNIT");}
}

//save as B.java

package p1;
import p.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.show();
  }
}
```

Output: Hello GNIT

Default:

The default modifier is accessible only within package.

```
//save by A.java
package p;
public class A{
        int i=5;
         void m()
        {System.out.println("Hello GNIT");
        }
}

//save by B.java
package p;
public class B{
 public static void main(String args[])
        {
                A obj = new A();
                obj.m();
 }
}
```

In the above example, the scope of class A and its method m() is default so it cannot be accessed from outside the package.

protected :

The protected modifier is accessible within same package and child class from other package.

```
//save by A.java
package p;
public class A{
        int i=5;
         void m()
        {System.out.println("Hello GNIT");
        }
}
```

```
//save by A1.java
package p;
public class A1 extends A{
        int j=15;
         void m1()
        {System.out.println("Hello GNIT");
        }
}
//save by A2.java
package p;
public class A2{
        int k=25;
         void m2()
        {System.out.println("Hello GNIT");
        }
}

//save by A3.java
package p1;
public class A3 extends A{
        int s=35;
         void m3()
        {System.out.println("Hello GNIT");
        }
}

//save by B.java
package p1;

public class B{
                int t=45;
                void m4()
                        {System.out.println("Hello GNIT");
                                }
                }

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

```
        A obj = new A();   // Compilation error (m() of class A cannot accessible from
outer package P1 non subclass of A (i.e B).
        obj.m();
 }
}
```

Access Modifier	within class	within package but not within class.	outside package by subclass only	outside package by non-subclass only
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

MODULE 5: Exception handling, Multithreading and Applet Programming

LECTURE 1:Exception handling - Basics, different types of exception classes. Difference between Checked & Unchecked Exception

5.1.1. Exception:

Exception is an abnormal condition arising inside a program which disrupts the normal flow of that program at the time of execution.

5.1.2. Reason of Exception:

- User has entered invalid and improper data.
- A file that needs to be opened has already been deleted.
- Network connection has been lost during communications between nodes.
- JVM has run out ofmemory.

5.1.3. ExceptionHandling:

ExceptionHandlingisaprocesstomaintainnormalflowoftheprogram by avoiding any kind of abnormal condition arises during program execution.

Note: In ExceptionHandlingwe should try to catch the exception object thrown by the error condition and give a proper guidance to user to take necessary actions.

Let's take a scenario:

Program A

{

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;
statement 6;
```

```
statement 7; //Abnormal condition/exception arises during execution
statement 8;
statement 9;
statement 10;
statement 11;
statement 12;
}
```

Suppose there are twelve statements in my program and there occurs an exception at statement seven, rest of the code will not be executed i.e. statement seven to twelve will not run. By using exception handling, rest of the statement will be executed.

5.1.4. Types of Exceptions:

1. **Checked Exception:** If a file is to be opened, but the file cannot be found, an exception occurs. These exceptions are checked at compile-time and cannot simply be ignored at the time of compilation. Classes which extend the Throwable class (excluding RuntimeException and Error) are known as checked exceptions. A checked exception is typically a user error or a problem that cannot be observed by the programmer.

   Example of Checked Exception: IOException, SQLException etc.

2. **Unchecked Exception:** This exception ischecked during execution of the program and also ignored at the time of compilation. Unchecked Classes which extend *the* RuntimeException class are known as Unchecked Exceptions.

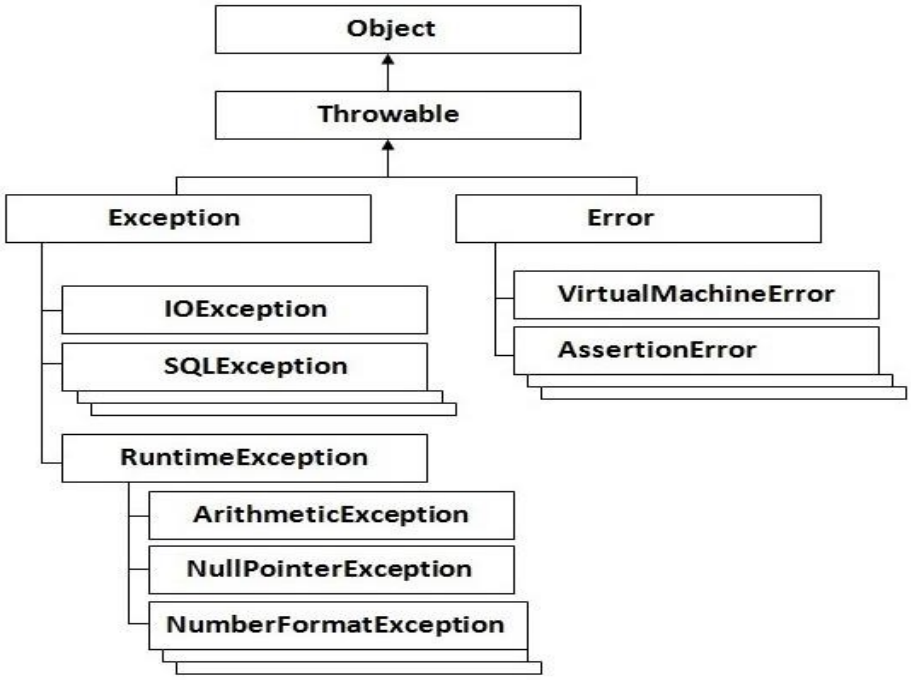
   Example of Checked Exception:
   a)ArithmeticException,
   b)NullPointerException etc.

Error:

Irrecoverable Circumstances arises at the time of program execution is called error. Error is actually not an exception. In case of error problem arises beyond the control of the user or the programmer. For example, if there is a semicolon

missing at the end of a statement or braces are not properly closed in a program an error will arise.

5.1.5. Different types of Exception classes



JAVA – Built in Exceptions:

- list of Java Unchecked Exception

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.

SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

- List of Java Checked Exceptions Defined in java.lang

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

5.1.6. Common Example of Java Exception

1) ArithmeticException :

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=10/0;//ArithmeticException
```

2)NullPointerException :

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3)ArrayIndexOutOfBoundsException :

If you are inserting any value in the wrong index, it would result

ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];

a[15]=750; //ArrayIndexOutOfBoundsException
```

LECTURE 2:Try & catch related case studies.

5.2.1. Handling Exceptions in Java

Following five keywords are used to handle an exception in Java:

1. try
2. catch
3. finally
4. throw
5. throws

When an exception arises inside the block called “try” the exception object can be thrown to it’s corresponding “catch” block for properly handling that abnormal situation (exception). If that exception properly handled within that “catch” block then normal flow of execution continuing otherwise the exception bypassed to java default exception handler. If default exception handler handled this situation properly then a proper predefined message shown to user, otherwise program execution disrupted.

5.2.2. Try & catch related case studies

CASE-1: Problem without exception handling
Let's try to understand the problem if we don't use try-catch block

```
class Test
{
 public static void main(String args[])
{
  int A=50/0;//arithmetic exception arises
  System.out.println("Continuation of rest prog....");
    }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

Justification:

There is no exception Handling mechanism(try-catch). So exception arises due to A=50/0;(devide by zero) this statement and transfer that exception object to java default exception handler and default exception handler handled this situation properly with predefined message shown to user, "Exception in thread main java.lang.ArithmeticException:/ by zero" as an output.As displayed in the above example, rest of the code is not executed (in such case, "Continuation of rest prog...." is not printed).

CASE-2: Exception creates but properly handled

Let's see the solution of above problem by java try-catch block.

```
class Test
        {
        public static void main(String args[])
                {
                        try
                        {
                                int A=50/0;

                                }
                        catch(ArithmeticException e)
                                {
                                        System.out.println("Hi Exception--"+e);
                                        }
                        System.out.println("Continuation of rest prog....");
                        }
                }
```

Output:

Hi Exception-- Exception in thread main java.lang.ArithmeticException:/ by zero
Continuation of rest prog....

Justification:

Arithmatic Exception occurs on int A=50/0 and the exception object of

ArithmeticException class can be thrown by try block to it's corresponding catch and the exception object received into e. The exception is properly handled so rest part is successfully executed i.e "Continuation of rest prog...." is printed after showing the value of e.

CASE-3: Exception creates but not handled

```
class Test
        {
        public static void main(String args[])
                {
                        try
                        {
                                int A=50/0;

                                }
                        catch(ArrayIndexOutofBoundException e)
                                {
                                        System.out.println("Hi Exception--"+e);
                                        }
                        System.out.println("Continuation of rest prog....");
                        }
                }
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

Justification:

Arithmatic Exception occurs on int A=50/0 and the exception object of ArithmeticException class can be thrown by try block to it's corresponding catch but type of exception is not matched. So the exception is not properly handled and then control passed into default exception handler without executing "Continuation of rest prog....".

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

```
class MultipleCatch
    {
     public static void main(String args[])
      {
            try
            {
                    int a[]=new int[5];
                   a[5]=30/0;
                    }
        catch(ArithmeticException e)
       {
            System.out.println("1.....");
            }
       catch(ArrayIndexOutOfBoundsException e)
       {
            System.out.println("2.....");
       }
       catch(Exception e)
       {
       System.out.println("common task completed");
       }
      System.out.println("rest code...");
```

```
            }
        }
```

Output: 1.....

rest code...

Java Nested try statement

```
class A{
public static void main(String args[])
{
        try
        {
           try
                {
                        System.out.println("Hello GNIT");
                        int b =30/0;
                }
        catch(ArithmeticException e)
                {
                        System.out.println(e);
                        }
          try
                {
                         int a[]=new int[5];
                         a[6]=15;
                 }
        catch(ArrayIndexOutOfBoundException e)
                {
                        System.out.println(e);
                        }

         System.out.println("Rest...");
        }
  catch(Exception e)
 {
```

```
   System.out.println("Hi GNIT");
        }

  System.out.println("normal flow..");
   }
}
```

OUTPUT:

Hello GNIT

Exception in thread main java.lang.ArithmeticException:/ by zero
Exception in thread main java.lang.ArrayIndexOutOfBoundException

normal flow....

LECTURE 3:Throw, throws & finally

5.3.1. Java finally block

Let's see the different cases where java finally block can be used.

Case 1:

finally example where exception doesn't occur.

```
class Finally_block{
public static void main(String args[]){
try{
 int d=35/5;
 System.out.println(d);
 }
catch(NullPointerException e)
{System.out.println(e);
    }
```

```
        finally
        {System.out.println("This block execution is confirmed");
            }
        System.out.println("Thank you GNIT");
        }
}
```

Output:7

This block execution is confirmed

Thank you GNIT

Case 2

finally example where exception occurs and not handled.

```
        class ABC{
        public static void main(String args[]){
          try{
                int d=45/0;
               System.out.println(d);
                        }
           catch(NullPointerException e){System.out.println(e);}
          finally{System.out.println("This block execution is confirmed ");}
          System.out.println("Thank you GNIT ");
                  }
                 }
```

Output: This block execution is confirmed

Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

finally example where exception occurs and handled.

```
class ABC{
 public static void main(String args[])
{
   try{
                int d=45/0;
                 System.out.println(d);
 }
 catch(ArithmeticException e){System.out.println("Hello GNIT-"+e);}
finally{System.out.println("This block execution is confirmed ");}
System.out.println("Thank you GNIT");
}
}
```

Output:

Hello GNIT-Exception in thread main java.lang.ArithmeticException:/ by zero.

This block execution is confirmed

Thank you GNIT

5.3.2. Throw Keyword

When we want to throw an exception explicitly and it is created by the user then this type of exception is called user defined or custom exception and we can use throw keyword for this.

Syntax:

throw exception;

Example:

throw new IOException("Input/Output related error);

```
class XYZ
{
        static void test (int temperature)
       {
        if(temperature <98)
                throw new ArithmeticException("Low Temperature");
          else
            System.out.println("Fever...");
     }
       public static void main(String args[])
   {
           test(55);
           System.out.println("Thanks to Temperature measurement");
       }
}
```

OUTPUT

Exception in thread main java.lang.ArithmeticException:not valid

In this example we create the test method which takes integer value as a parameter. If the temperature is less than 98, an object of ArithmeticException : Low Temperature is being thrown by test to main().There is no exception handling mechanism introduced in main(try-catch). So that exception goes to default handler and the message “Thanks to Temperature measurement” is not printed.

5.3.3. Java throws keyword

The **Java throws keyword** is used to declare an exception. When a called method transfer any exception to it's calling method then we use throws keyword.

Syntax:

methodname() **throws** exceptionname{

//method code where exception occurs.

}

Note: Only checked exception should be declared, because:

- **unchecked Exception:** under our control so modification done by us inside the code.
- **error:** beyond the control of programmer

Implementation of throws:

```
import java.io.IOException;
class A{
 void pqr()throws IOException
  {
        throw new IOException("Input/Output excp");//checked exception
   }
 void stw()throws IOException
 {
  pqr();
  }
 void abc()
 {
    try{
          stw();
      }
    catch(Exception e)
     {
    System.out.println(e+"Hello GNIT");
      }
 }
```

```
 public static void main(String args[])
{
  A obj=new A();
  obj.abc();
  System.out.println("END...");
 }
```

OUTPUT:
Input/Output excp Hello GNIT
END...

Justification:

Method pqr() throws the exception "Input/Output excp" to it's calling fn stw(). There is no handling mechanism in stw() so thats why stw() can forwarded that exception into it's calling abc(). Inside abc() , catch statement can handled that exception successfully.So program flow of execution cant interrupted.

5.3.4. throw vs throws

1)Checked exception can be propagated from one function to other function with throws rather than Unchecked exception .
2)Throw keyword is used for an instance where as throws is followed by method inside a class.
3)Throw is used within the method.Throws is used with the method signature.

LECTURE 4:Creation of user defined exception.

5.4.1. When we want to throw an exception explicitly and it is created by the user then this type of exception is called user defined or custom exception.We can use throw keyword for this purpose.The **Java throws keyword** is used to declare an exception. When a called method transfer any exception to its calling method then we use throws keyword. This part is completely explain on 5.3.2 and 5.3.3.

LECTURE 5:Multithreading - Basics, main thread, thread life cycle.

5.5.1. Multithreading

When a program (At execution time program is called process) is sub divided into two or more subprograms, which can be parallely executed at the same time, then each subprogram is called thread andeachthreaddefinesaseparatepathofexecution.When multiple thread parallely executed at the same time it is called multithreading.

5.5.2. Benefits of Multithreading

1. Parallel execution done at sametime but with separatepathofexecution. So one thread can't interfere other in between the parent process.
2. Division of a long program into different threads can increases the speed of the programexecution
3. Improved performance andconcurrency
4. Simultaneous access to multipleapplications

5.5.3. Life Cycle of Thread

Different states of a thread can be discussed following.

1. **Newborn State:** In this state a new thread object is created.

2. **Runnable State:** In this state a thread is ready for execution and waiting for the availability of the processor. From newborn state all new born threads are come into the queue of Runnable state to get CPU.If all threads in queue are of same priority then they are given time slots for execution in round robin fashion.

3. **Running State:** In this state CPU has given its time to the thread one by one coming from Runnable state for execution. A thread keeps running until the following conditions occurs

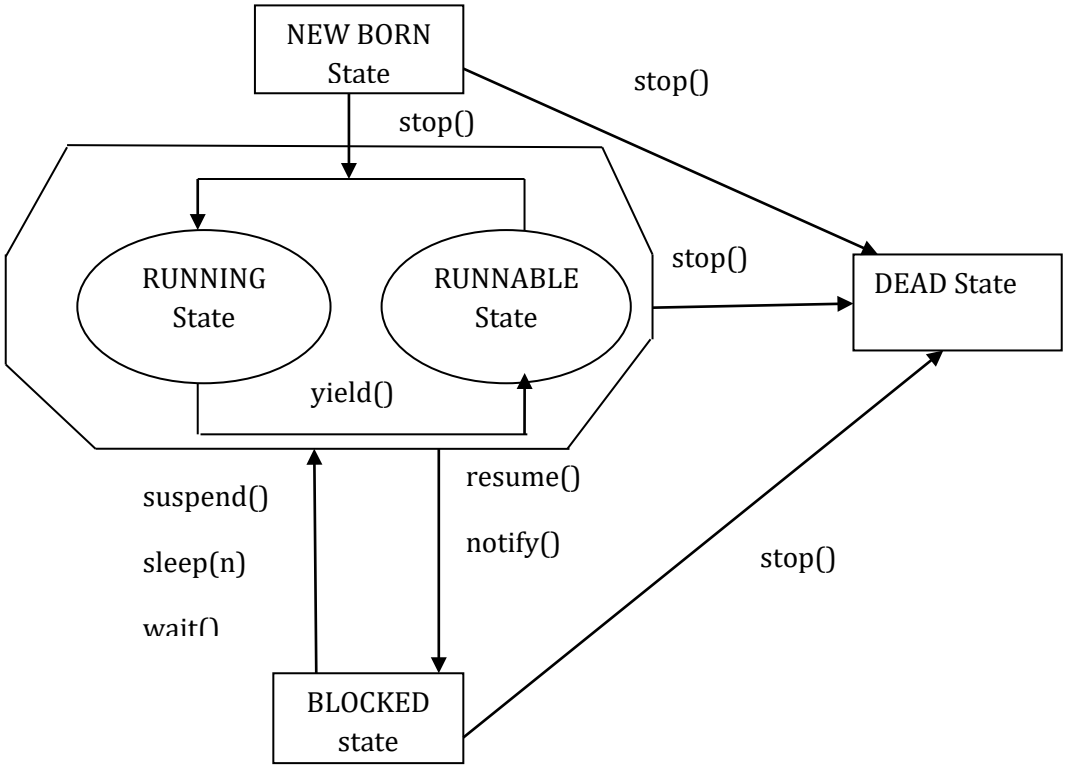
    i. A thread gets suspended using **suspend()** method and go to blocked or waiting state which can only be recalled with the help of **resume()** method.

    ii. A thread has go to sleep mode for a specified period of time using s**leep(time)** method, where default time mentioned in milliseconds

iii. A thread is made to wait for some event to occur using **wait ()** method. In this case a thread can be scheduled to run again using **notify ()** method.

4. **Waiting/Blocked State:** If any thread is prevented to get CPU on runnable state from running state due to suspension or sleep etc. Then those thread can go to Blocked state from where it can takes place furthermore into runnable state.

5. **Dead State:** A runnable thread enters the Dead or terminated state when it completes its task (Mature death) or explicitely get death from other conditions before completes its task (Immature death).



5.5.4. Main Thread

When a Java program execution starts, one thread begins running upto end of execution. This thread is called main thread.

- Child threads are generated from main thread

- This is the last thread which is used to complete program execution.

LECTURE-6: Creation of multiple threads-yield(), suspend(), sleep(n), resume(), wait(), notify(), join(), isAlive().

5.6.1. Thread Creation

Java defines two ways in which this can be accomplished:

- By implementing the Runnable interface.
- By extending the Thread class.

Thread creation by implementing the Runnable interface:

```
class A implements Runnable
{
        public void run()
        {
                System.out.println("Hello GNIT");
         }

        public static void main(String args[])
        {
                A ob=new A();
                Thread t =new Thread(ob);
                t.start();
                 }
}
```

Output: Hello GNIT

Here we explicitly create Thread class object and passing that object of your class that implements Runnable interface so that's why run() method is being executing.

Thread creation by extending the Thread class:

```
class A extends Thread
{
        public void run()
        {
            System.out.println("Hello GNIT...");
        }
        public static void main(String args[])
        {
        A thread_ob=new A();
       thread_ob.start();
        }
   }
```

Output: Hello GNIT

Important methods in Java thread

a) public void start() :

Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.

b) public void run() :

If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.

5.6.2. yield() :

```
class  X extends Thread
```

```
{
 public void run()
  { int p=1;
    while(int p<=6)
     {
       if(p==2)
        {
          yield();
          System.out.println("X:"+p);
        }
       System.out.println("End of X");
      p++;
     }
  }
}
class  Y extends Thread
{
 public void run()
  { int p=1;
    while(int p<=6)
     {
        System.out.println("Y:"+p);
          p++;
        }
   System.out.println("End of Y");
             }
  }
Class Z
{
 Public static void main(String[] args)
 {
   X x=new X();
   Y y=new Y();
   x.start();
   y.start();
 } }
```

OUTPUT:
X: 1
Y: 1
Y: 2
Y: 3
Y: 4
Y: 5
Y: 6
End of Y
X: 2
X: 3
X: 4
X: 5
X: 6
End of X

Thread X gets started and when condition if(i==2) occurs yield() method gets evoked and the control is relinquished from thread XtothreadYwhichruntoitscompletion. After completion the task Thread Y transfer the control to X and thread X executes.

5.6.3. sleep(n) :

```
class  X extends Thread
{
 public void run()
  { int p=1;
    while(int p<=6)
     {
        try {
                if(p==2)
                 sleep(5000);
                   }
         catch(Exception e)
             {                 };
        System.out.println("X:"+p);
```

```
        p++;
       }
   System.out.println("End of X");
    }
  public static void main(String[] args)
  {
    X x=new X();
    x.start();
     }
            }
```

OUTPUT:
X: 1
X: 2
X: 3
X: 4
X: 5
X: 6
End of X

Due to sleep method after printing X: 1 the next o/p X: 2 printed after 5 seconds.

5.6.4. suspend() and resume()

```
class  X extends Thread
{
  public void run()
   { int p=1;
     while(int p<=6)
      {
            System.out.println("X:"+p);
            p++;
      }
     System.out.println("End of X");

   }
```

```
}
class  Y extends Thread
{
 public void run()
  { int p=1;
    while(int p<=6)
     {
          System.out.println("Y:"+p);
          p++;
     }
    System.out.println("End of Y");

  }
}

Class Z
{
 Public static void main(String[] args)
  {
    X x=new X();
    Y y=new Y();
    x.start();
    y.start();
    x.suspend();
    x.resume();
    }
}
```

OUTPUT:

X: 1
X: 2
X: 3
X: 4
X: 5
X: 6
End of X

```
Y: 1
Y: 2
Y: 3
Y: 4
Y: 5
Y: 6
End of Y
```

In above two threads X and Y are created. Thread X is started ahead of Thread Y, but X is suspended using suspend() method causing Thread Y to get hold of the processor allowing it to run and when Thread Y is resumed using resume() method it runs to its completion.

5.6.5. isAlive() and join()

isAlive()methodtests ifanythread isalive or not.Athreadisaliveifithasbeen startedandhasnotyetdied.This method returns true if this thread is alive, false otherwise.

join() method waits for a thread to die. It causes the currently thread to stop executing until the thread it joins with completes its task.

```
class  X extends Thread
{
        public void run()
        {    System.out.println("GNIT:"+isAlive());
       }
         }


class  Z extends Thread
{
 public static void main(String[] args)
        {
                X x=new X();
```

```
                x.start();
                try{
                        x.join();
                        }
                catch(Exception e)
                        {  };

                System.out.println("GNIT:"+x.isAlive());
        }

}
```

OUTPUT:
GNIT: true
GNIT: false

Join() is called from Thread X which stops executing of further statement until X is dead. In this statement:- System.out.println("GNIT:"+isAlive()); Thread X is alive so the value gets printed by isalive() method is true. In this statement :-
System.out.println("GNIT:"+x.isAlive());
Here x.isAlive() returns false as the Thread X is completed.

LECTURE-6:Thread priorities &Thread synchronization

Thread priorities:

Java thread priority means operating system determine the order in which threads arescheduled. 3 kind of properties are there in Java. Java priorities are in the range between **MIN_PRIORITY (value= 1)** and **MAX_PRIORITY (value=10)**. By default, every thread is given priority **NORM_PRIORITY (value=5)**.Threads with higher priority are more important to a program and should be allocated process or time before lower-priority threads.

```
class  X extends Thread
{
 public void run()
```

```
  { int p=1;
    while(int p<=6)
     {
          System.out.println("X:"+p);
          p++;
     }
     System.out.println("End of X");

   }
}
class  Y extends Thread
{
 public void run()
  { int p=1;
    while(int p<=6)
     {
          System.out.println("Y:"+p);
          p++;
     }
     System.out.println("End of Y");

   }
}

Class Z
{
 Public static void main(String[] args)
  {
    X x=new X();
    Y y=new Y();
    y.setPriority(10);
    x.setPriority(1);
    x.start();
    y.start();

    }
```

```
}
```

OUTPUT:

```
Y: 1
Y: 2
Y: 3
Y: 4
Y: 5
Y: 6
End of Y
X: 1
X: 2
X: 3
X: 4
X: 5
X: 6
End of X
```

Priority of Thread y is greater than the priority of Thread x. Although x is started before y but due to highest priority y thread executes first.

Thread synchronization

- When two or more threads want to access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization.
- Here Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object'slock.

```
Class A
{
  void abc( int i)
```

```
  {
     Thread t = Thread.currentThread();
     for(int x=1;x<=5;x++)
         {
                  System.out.println(t.getName()+"==="+(i+x));
                        }

         }
}

Class B extends Thread
{
   A ob=new A();
   public void run()
    {
       ob.abc(10);
    }

}

Class Z
{
 public static void main(String[] args)
  {
    B b=new B();
    Thread t1= new Thread(b);
     Thread t2= new Thread(b);
     t1.setName("Thread 1");
     t2.setName("Thread 2");
    t1.start();
    t2.start();
        }
}
```

OUTPUT:
Thread 1: 11

Thread 2: 11
Thread 1: 12
Thread 2: 12
Thread 1: 13
Thread 2: 13
Thread 1: 14
Thread 2: 14
Thread 1: 15
Thread 2: 15

If we write **synchronized void abc(int i)** then O/P will be

OUTPUT:
Thread 1: 11
Thread 1: 12
Thread 1: 13
Thread 1: 14
Thread 1: 15
Thread 2: 11
Thread 2: 12
Thread 2: 13
Thread 2: 14
Thread 2: 15

LECTURE-7:Interthread communication, deadlocks for threads

5.7.1. Interthread Communication (wait() & notify() method)

Synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methods of Object Class:

wait() : This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method.

General syntax for calling wait() method is like this:

```
synchronized( lockObject )
{
  while( ! condition )
  {
    lockObject.wait();
  }

  //take the action here;
}
```

notify() : This method wakes up the first thread that called wait( ) on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier's synchronized block has completed. So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

```
synchronized(lockObject)
{
  //establish_the_condition;

  lockObject.notify();

  //any additional code if needed
}
```

5.7.2. Thread Deadlock

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

LECTURE-8: Applet Programming-Basics, applet life cycle, difference between application & applet programming

5.8.1. Applet Programming-Basics:

- An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

- There are some important differences between an applet and a standalone Java application, including the following –

- An applet is a Java class that extends the java.applet.Applet class.

- A main() method is not invoked on an applet, and an applet class will not define main().

- Applets are designed to be embedded within an HTML page.

- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.

- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.

- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.

- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.

- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

5.8.2. Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet

init- This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.

start- This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.

stop - This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

destroy- This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

paint − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

LECTURE-9:Parameter passing in applets.

5.9.1.Param Tag :

The <param> tag is a sub tag of the <applet> tag. The <param> tag contains two attributes: name and value which are used to specify the name of the parameter and the value of the parameter respectively. For example, the param tags for passing name and age parameters looks as shown below:

```
<param name=”name” value=”GNIT” />
<param name=”age” value=”7”/>
```

Now, these two parameters can be accessed in the applet program using the getParameter() method of the Applet class.

5.9.2. getParameter() Method:

The getParameter() method of the Applet class can be used to retrieve the parameters passed from the HTML page. The syntax of getParameter() method is as follows:
String getParameter(String param-name)
Let’s look at a sample program which demonstrates the <param> HTML tag and the getParameter() method:

======