OOP in JAVA

Soba Raj Paudel

Course Title: **Object Oriented Programming in Java (3 Cr.)**
Course Code: **CACS204**
Year/Semester: **II/III**
Class Load: **6 Hrs. / Week (Theory: 3 Hrs, Tutorial: 1, Practical: 2 Hrs.)**

Course Description

This course covers preliminary concepts of object-oriented approach in programming with basic skills using Java. Control structures, Classes, methods and argument passing and iteration; graphical user interface basics Programming and documentation style.

Course Objectives

The general objectives of this course are to provide fundamental concepts of Object Oriented Programming and make students familiar with Java environment and its applications.

Course Contents

Unit 1 Introduction to Java **2 Hrs.**

Definition, History of Java, The Internet and Java's Place in IT, Applications and Applets, Java Virtual Machine, Byte Code- not an Executable code, Procedure-Oriented vs. Object-Oriented Programming, Compiling and Running a Simple Program, Setting up your Computer for Java Environment, Writing a Program, Compiling, Interpreting and Running the Program, Handling Common Errors

Unit 2 Tokens, Expressions and Control Structures **5 Hrs.**

Primitive Data Types: Integers, Floating-Point types, Characters, Booleans; User-Defined Data Types, Declarations, Constants, Identifiers, Literals, Type Conversion and Casting, Variables: Variable Definition and Assignment, Default Variable Initializations; Command-Line Arguments, Arrays of Primitive Data Types, Comment Syntax, Garbage Collection, Expressions, Using Operators: Arithmetic, Bitwise, Relational, Logical, Assignment, Conditional, Shift, Ternary, Auto-increment and Auto-decrement; Using Control Statements(Branching: if, switch; Looping: while, do-while, for; Jumping statements: break, continue and return)

Unit 3 Object Oriented Programming Concepts **9 Hrs.**

Fundamentals of Classes: A Simple Class, Creating Class Instances, Adding methods to a class, Calling Functions/Methods; Abstraction, Encapsulation, Using 'this' keyword, Constructors, Default constructors, Parameterized constructors, More on methods: Passing by Value, by Reference, Access Control, Methods that Return Values, Polymorphism and Method Overloading, Recursion; Nested and Inner Classes.

Unit 4 Inheritance & Packaging **3 Hrs.**

Inheritance: Using 'extends' keyword, Subclasses and Superclasses, 'super' keyword usage, Overriding Methods, Dynamic Method Dispatch; The Object class, Abstract and Final Classes, Packages: Defining a Package, Importing a Package: Access Control; Interfaces: Defining an Interface, Implementing and applying interfaces.

Unit 5 Handling Error/Exceptions **2 Hrs.**

Basic Exceptions, Proper use of exceptions, User defined Exceptions, Catching Exception: try, catch; Throwing and re-throwing: throw, throws; Cleaning up using the finally clause.

Unit 6 Handling Strings **2 Hrs.**

Creation, Concatenation and Conversion of a String, Changing Case, Character Extraction, String Comparison, Searching Strings, Modifying Strings, String Buffer.

Unit 7 Threads **3 Hrs.**

Create/Instantiate/Start New Threads: Extending java.lang.Thread, Implementing java.lang.Runnable Interface; Understand Thread Execution, Thread Priorities, Synchronization, Inter-Thread Communication, Deadlock

Unit 8 I/O and Streams **2 Hrs.**

java.io package, Files and directories, Streams: Byte Streams and Character Streams; Reading/Writing Console Input/Output, Reading and Writing files, The Serialization Interface, Serialization & Deserialization.

Unit 9 Understanding Core Packages **3 Hrs.**

Using java.lang Package: java.lang.Math, Wrapper classes and associated methods (Number, Double, Float; Integer, Byte; Short, Long; Character, Boolean); Using java.util package: Core classes (Vector, Stack, Dictionary, Hashtable, Enumerations, Random Number Generation).

Unit 10 Holding Collection of Data **3 Hrs.**

Arrays And Collection Classes/Interfaces, Map/List/Set Implementations: Map Interface, List Interface, Set Interface, Collection Classes: Array List, Linked List, Hash Set and Tree Set; Accessing Collections/Use of An Iterator, Comparator.

Unit 11 Java Applications **8 Hrs.**

About AWT & Swing, About JFrame (a top level window in Swing), Swing components (JLabel, About text component like JTextField, JButton, Event Handling in Swing Applications, Layout Management using Flow Layout, Border Layout, Grid Layout, Using JPanel, Choice components like JCheck Box, JRadio

Button, Borders components, JCombo Box & its events, JList & its events with MVC patterns, Key & Mouse Event Handling, Menus in swing, JText Area, Dialog boxes in swing, JTable for Displaying Data in Tabular form, MDI using JDesktop Pane & JInternal Frame, Using IDE like Netbeans, JBuilder for building java applications using Drag & Drop), Adapter classes

Unit 12Introduction to Java Applets **1 Hr.**

Definition, Applet lifecycle methods, Build a simple applet, Using Applet Viewer, Adding Controls: Animation Concepts.

Unit 13Database Programming using JDBC **2 Hrs.**

Using Connection, Statement & Result Set Interfaces for Manipulating Data with the Databases

Laboratory Works

Laboratory works should be done covering all the topics listed above and a small project work should be carried out using the concept learnt in this course. Project should be assigned on Individual Basis.

Teaching Methods

The general teaching pedagogy includes class lectures, group discussions, case studies, guest lectures, research work, project work, assignments (theoretical and practical), and examinations (written and verbal), depending upon the nature of the topics. The teaching faculty will determine the choice of teaching pedagogy as per the need of the topics.

Evaluation

Examination Scheme				
Internal Assessment		External Assessment		Total
Theory	Practical	Theory	Practical	
20	20 (3 Hrs.)	60 (3 hrs.)	-	100

Text Books

1. Deitel & Dietel, "*Java: How to program*", 9th Edition, Pearson Education, 2011, ISBN: 9780273759768
2. Herbert Schildt, "*Java: The Complete Reference*", Seventh Edition, McGraw-Hill 2006, ISBN: 0072263857

Reference Books

1. Bruce Eckel, "*Thinking in Java*", 4th Edition, Prentice Hall, 2006, ISBN: 0-13-187248-6
2. Cay Horstmann and Grazy Cornell, "*Core Java Volume I-Fundamentals*", Ninth Edition, Prentice Hall, 2012, ISBN: 978-0137081899
3. E. Balagurusamy, "*Programming with Java: A Primer*", 4th Edition, Tata McGraw Hill Publication, India,

Unit-1

Introduction to JAVA

History of java

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades.

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed

to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

The Internet and Java's Place in IT

The Internet helped shoot Java to the forefront of programming, and Java, in turn, had a deep effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security.

Java Applets

An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

Security

Every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work on all computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the **Java Virtual Machine (JVM).** In essence, the original JVM was designed as an interpreter for bytecode. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.

Details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Servlets

A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.

Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container.

Java Buzzwords

Different factors that played an important role in molding the final form of the java language are:

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance no objects.

Robust

The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file.

Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

JVM (Java Virtual Machine)

JVM is a engine that provides runtime environment to drive the Java Code or applications. It converts Java bytecode into machines language. JVM is a part of JRE(Java Run Environment).

In other programming languages, the compiler produces machine code for a particular system. However, Java compiler produces code for a Virtual Machine known as Java Virtual Machine. First, Java code is complied into bytecode. This bytecode gets interpreted on different machines Between host system and Java source, Bytecode is an intermediary language. JVM is responsible for allocating memory space.

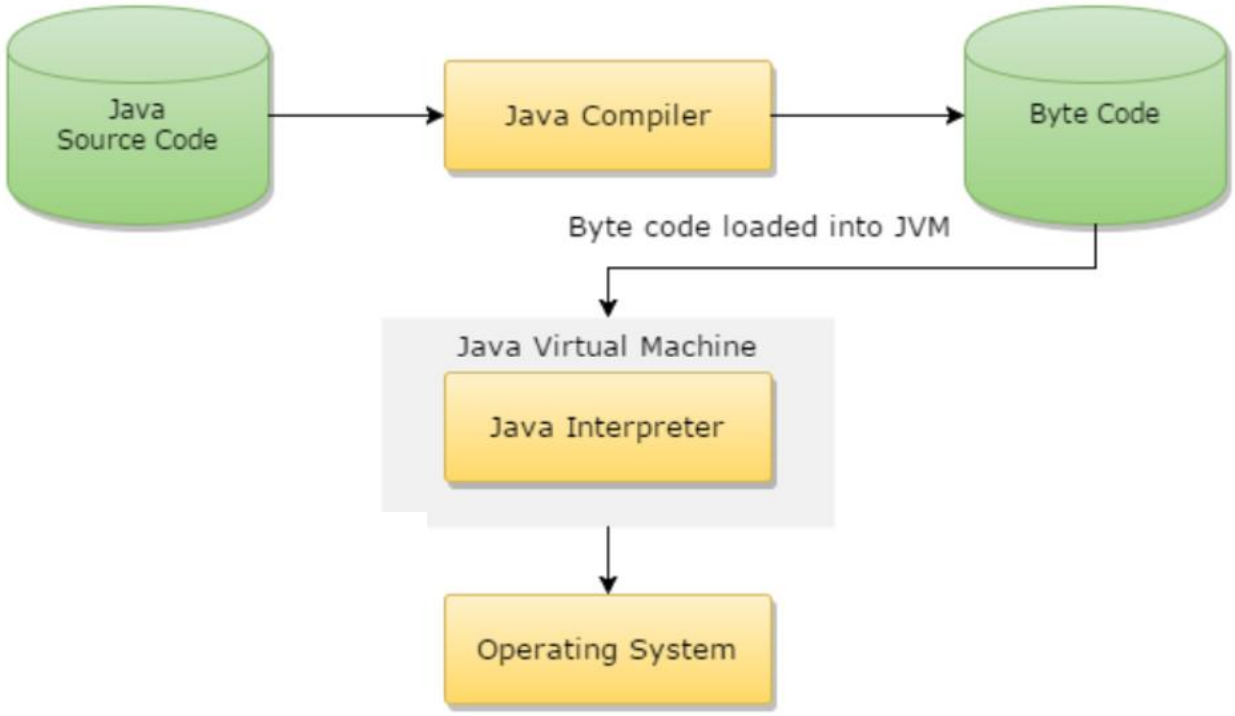
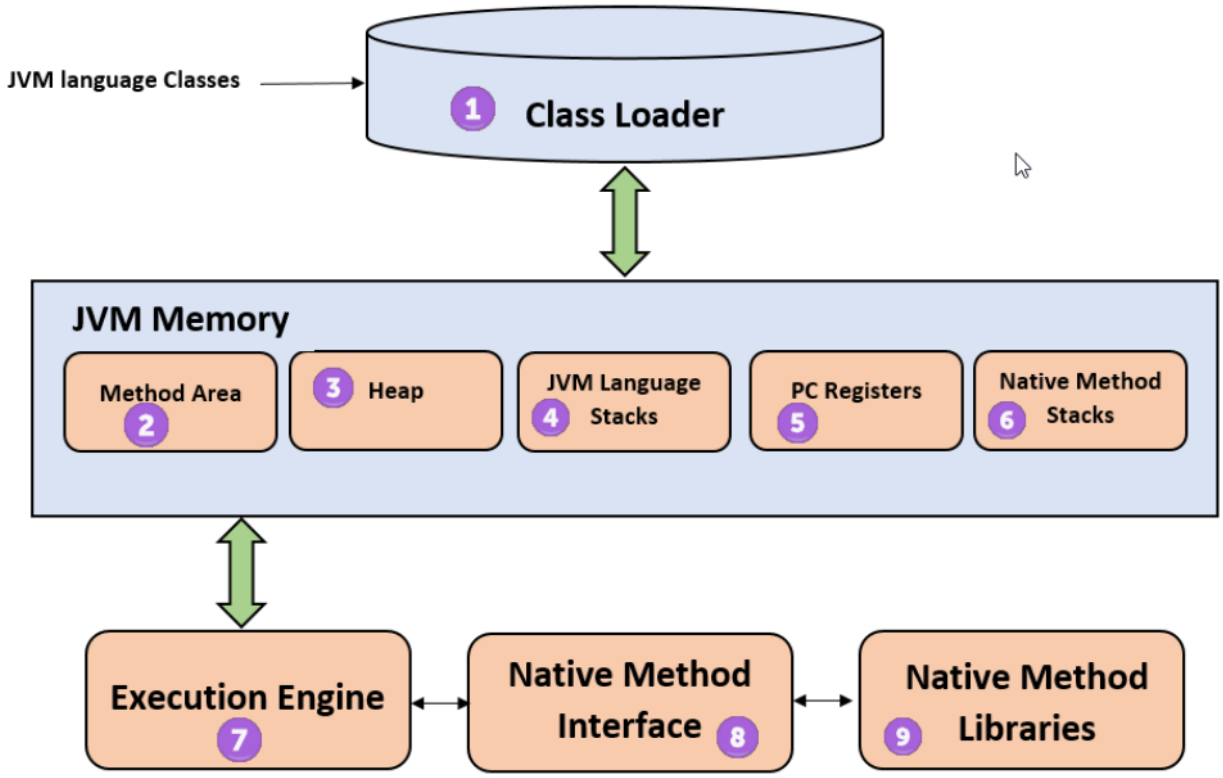


Fig: Diagram of JVM

JVM generates a .class(Bytecode) file, and that file can be run in any OS, but JVM should have in OS because JVM is platform dependent.

JVM Architecture

It contains classloader, memory area, execution engine etc.



1. **ClassLoader**

The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

2. **Method Area**

JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

3. **Heap**

All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

4. **JVM language Stacks**

Java language Stacks store local variables, and it's partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

5. **PC Registers**

PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

6. Native Method Stacks

Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.

7. Execution Engine

It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

8. Native Method interface

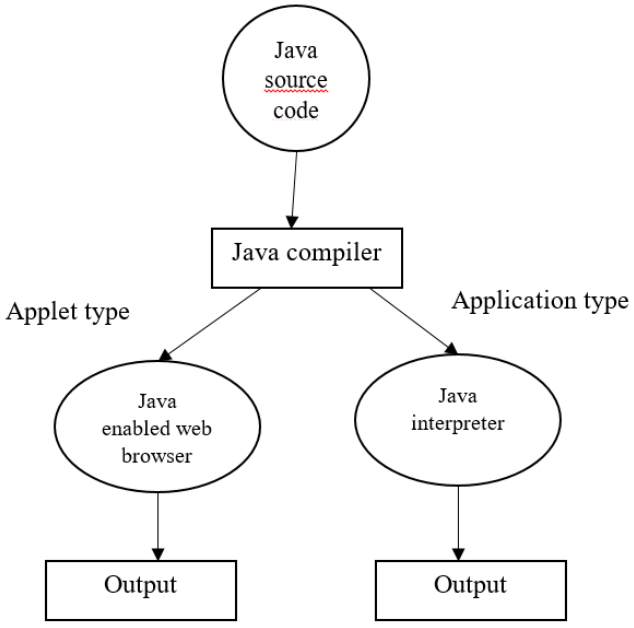
The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.

9. Native Method Libraries

Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine.

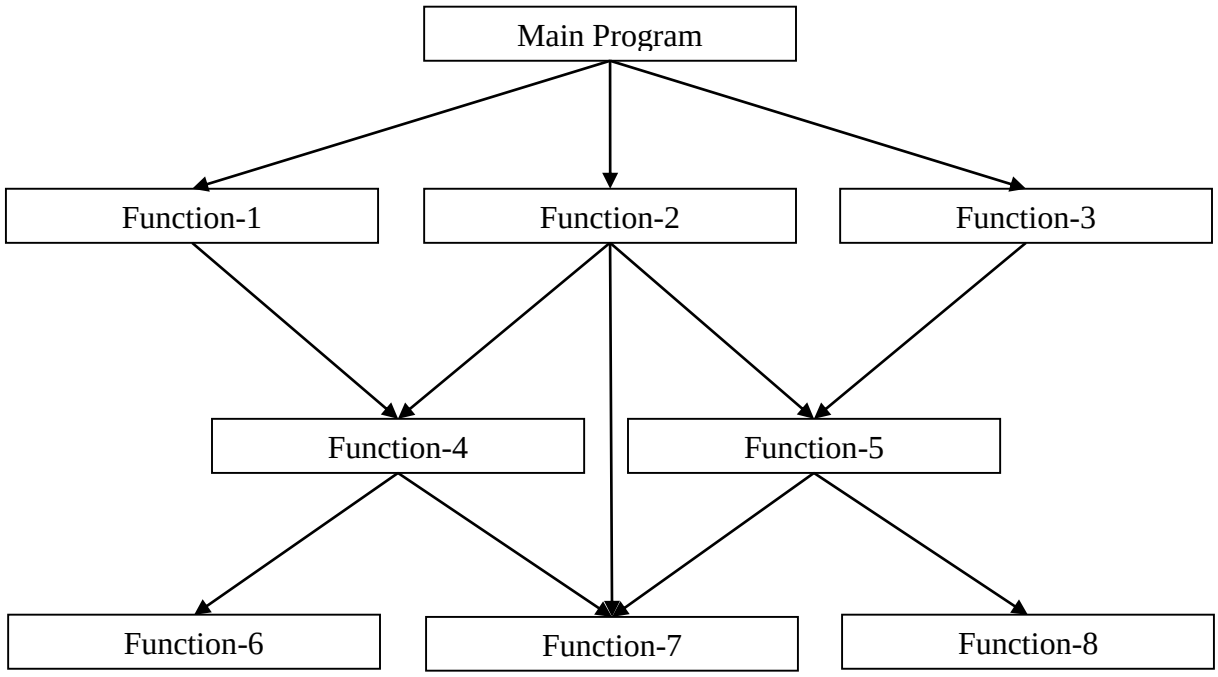
Java Applets and Applications

Java can be used to create two types of program : applications and applets. An application is a program that runs on our computer under the operating system of that computer. That is an application created by java is more or less like one created using C or C++. When used to create applications, java is not much different from any other computer language. Rather, it is java's ability to create applets that makes it important. An applet is an application designed to be transmitted over the internet and executed by a java compatible web browser.



Overview of structural programming approach

Conventional programming, using high level languages such as COBAL, FORTAN known as structural programming language (Procedure Oriented Programming). In procedure oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on function.



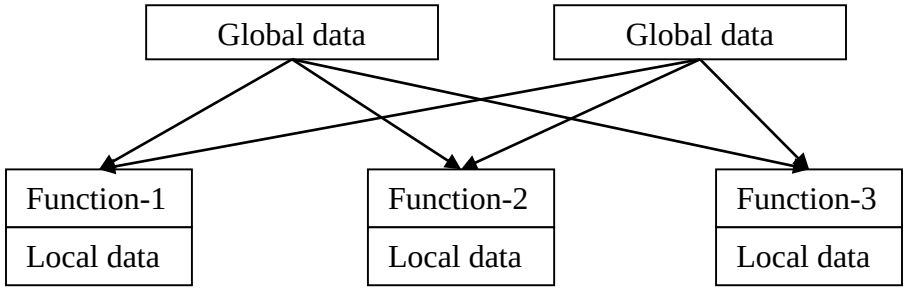
Typical structure of procedure- oriented programs

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or submodule, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Features of structure (procedural) programming are:

1. Emphasis is on doing things.
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data.
4. Data move openly around the system from function to function.
5. Functions transform data from one form to another.

6. Employs top-down approach in program design.



Relationship of data and functions in Structural Programming

Disadvantages of structure (procedure) programming

1. In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions.
2. In a large program it is very difficult to identify what data is used by which function.
3. It doesn't model real world problems very well because functions are action-oriented and do not really correspond to the elements of the problems.

Object oriented approach

Object oriented approach provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The data of an object can be accessed only by the functions associated with that object.

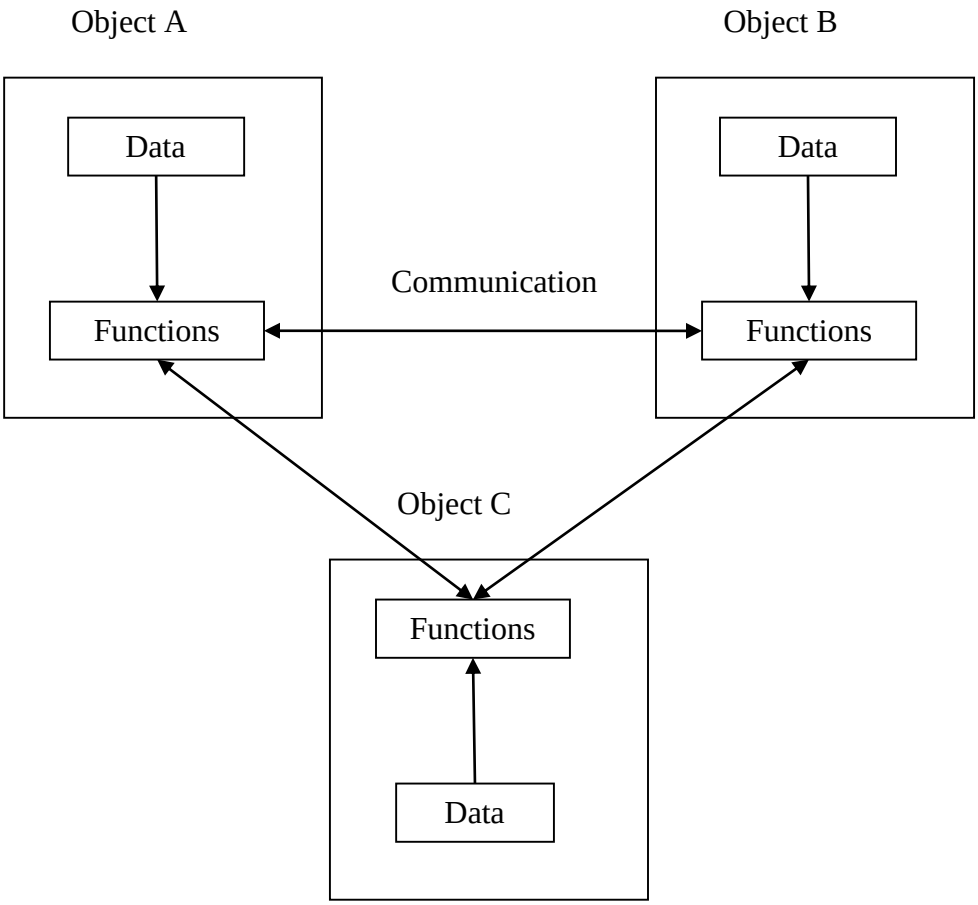
Features of object –oriented programming

1. Emphasis is on data rather than procedure.
2. Programs are divided into objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and cannot be accessed by external functions.
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added whenever necessary.
8. Follows bottom-up approach in program design.

Characteristics of Object Oriented Programming Languages (Elements of OOP)

1. Objects

   Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.



Organization of data and functions in OOP

2. Class

   A class is thus a collection of objects of similar type. For example mango, apple and orange are members of the class fruit. Classes are user defined data types and behave like the built in types of a programming language. In fact, objects are variables of the type class. Once a

class has been defined we can create any number of objects belonging to that class. If fruit has been defined as a class, then the statement

fruit mango;

will create and object ***mango*** belonging to the class ***fruit.***

For example, manager, peon, secretary clerk are the objects of the class employee. Similarly, car, bus, jeep, truck are the objects of the class vehicle. Classes are user defined data type (like a struct in C programming language) and behave much like built in data type (like int, char, float) of programming language.

One of the objects of student can have following values

Name = "Ramesh"
Registration_number = 200876255
Marks = {56, 67, 81, 50, 83}

3. Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding. We can assume encapsulation as a protective wrapper that prevents the data being accessed by other code defined outside the wrapper. By making use of encapsulation we can easily achieve abstraction. For example consider a TV (relating to the real world). It encapsulates hundreds of bits of information about the internal architecture. The user has only one method of affecting this complex encapsulation by opening the TV box.

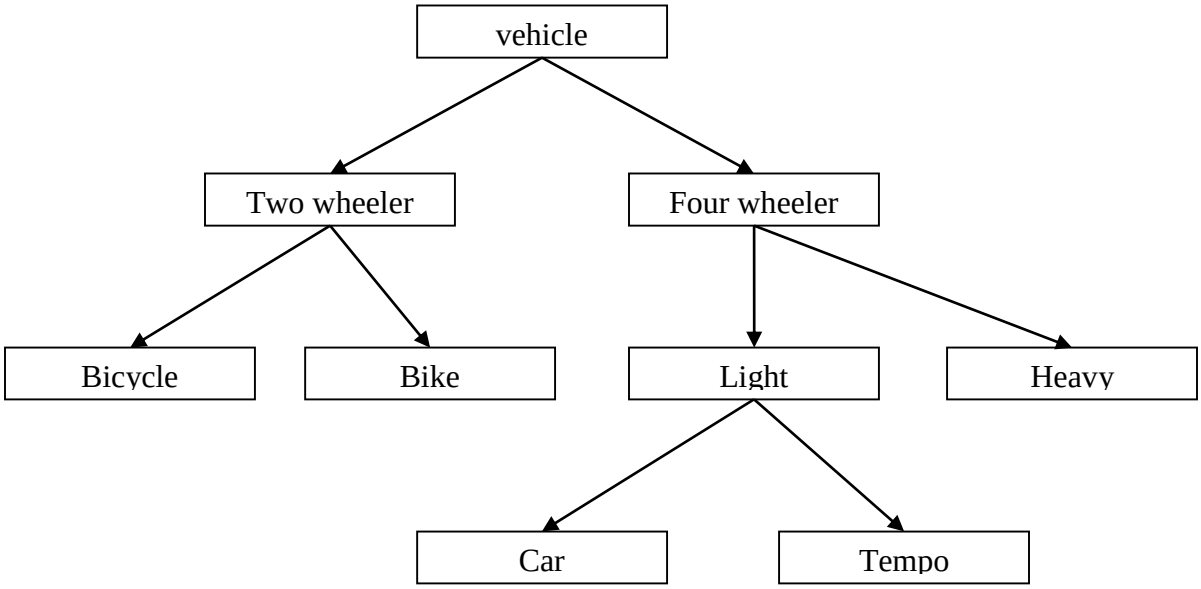
4. Data Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions. The abstraction allows the driver of the vehicle to drive without having detail knowledge of the complexity of the parts. The driver can drive the whole vehicle treating like a single object.

Similarly Operating System like Windows, UNIX provides abstraction to the user. The user can view his files and folders without knowing internal detail of Hard disk like the sector number, track number, cylinder number or head number. Operating System hides the truth about the disk hardware and presents a simple file-oriented interface.

5. Inheritance

   Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. We can use additional features to an existing class without modifying it. This is possible by deriving a new class (derived class) from the existing one (base class).This process of deriving a new class from the existing base class is called inheritance. It provides the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the existing code.

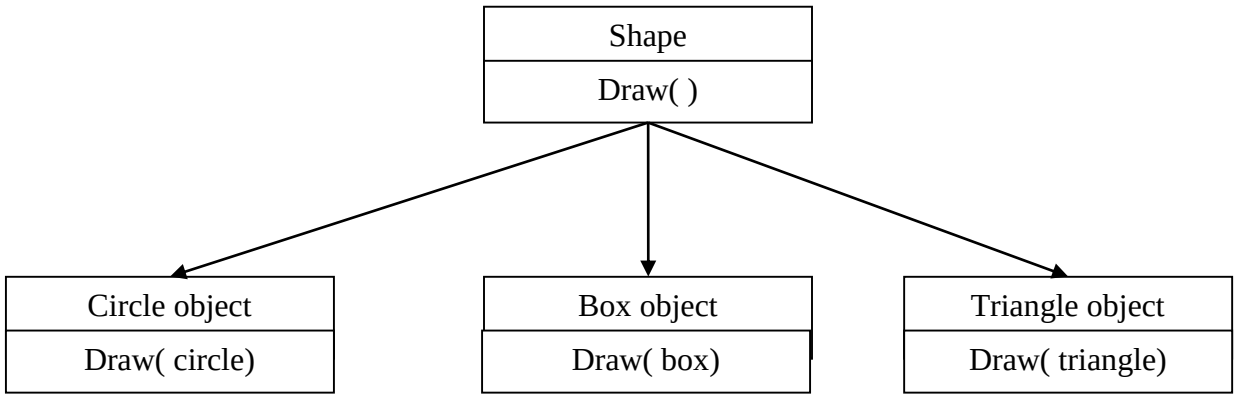


Example of inheritance

   In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

6. Polymorphism

   The word “polymorphism” is derived from Greek word “poly-morphos” which means many forms. Polymorphism is an ability to take more than one form. An operation may exhibit different behaviours in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The different ways of using same function or operator depending on what they are operating on is called polymorphism. When same function name is used in defining different function to operate on different data (type or number of data) then this feature of polymorphism is function overloading.



Polymorphism example

Example of polymorphism in OOP is operator overloading, function overloading. Still another type of polymorphism exist which is achieved at run time also called dynamic binding.

Disadvantages of OOPs

1. Compiler and runtime overhead. Object oriented program required greater processing overhead-demands more resources.
2. An object's natural environment is in RAM as a dynamic entity but traditional data storage in files or databases.
3. Requires the mastery in software engineering and programming methodology.
4. Benefits only in long run while managing large software projects.
5. Re-orientation of software developer to object-oriented thinking.

Application of OOPs

1. User interface design such as windows.
2. Used in real time system or real business system environment.
3. Simulation and modeling.
4. Object-Oriented databases.
5. AI and expert systems.
6. Neural networks and parallel programming.
7. Decision support and office automation system.
8. CAM/CAD systems.

Differences between Structured and Object Oriented Programming Language

Structured Programming Language	Object Oriented Programming Language
1. It is a procedure language emphasize on doing the things only i.e. it focuses on algorithm.	1. It is an object oriented technique in which emphasis is on data rather than procedure or algorithm.
2. Larger programs are divided into smaller programs known as functions.	2. Programs are divided into object.
3. Most of the function shares the global data. Function transforms the data from one to another.	3. Object may communicate with each other through function.
4. Data undervalued.	4. Data are more valued.
5. It is very difficult to add the new function and data once the program is completed.	5. New data and function can easily be added whenever the need arises.
6. It follows top-down approach.	6. It follows bottom-up approach
7. Example C,COBAL,FORTAN etc.	7. Example C++,JAVA,C# etc.

Java Environment

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as Java development kit (JDK) and the classes and methods are part of the Java Standard Library (JSL), also known as the Application Programming Interface (API).

Java Development Kit

The Java Development Kit comes with a collection of tools that are used for developing and running java programs. They include:

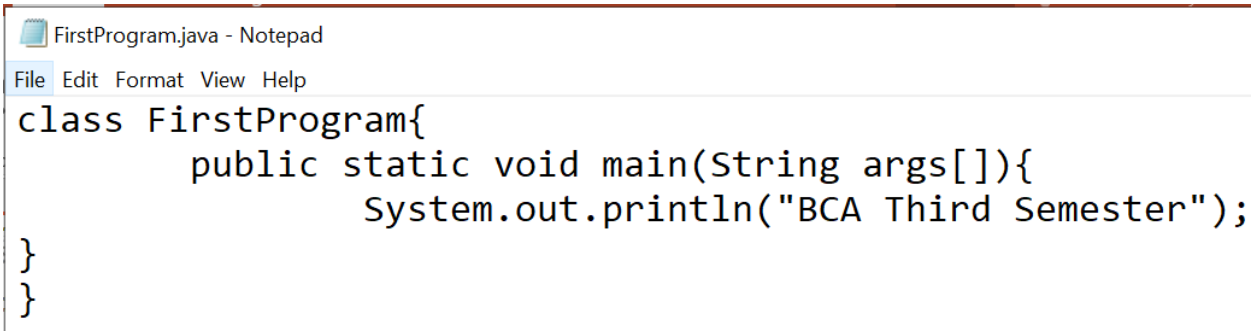
- appletviewer (for viewing Java applets)
- javac (java compiler)
- java (java interpreter)
- javap (java disassembler)
- javah (for C header files)
- javadoc (for creating HTML documents)
- jdb (java debugger)

Setting up your computer for Java Environment

1. In Search, search for and then select: System (Control Panel)
2. Click the **Advanced system settings** link.

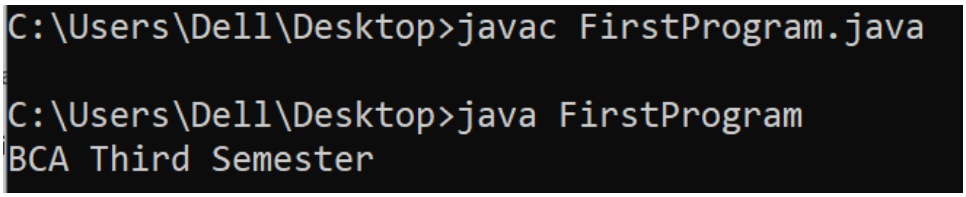
3. Click **Environment Variables**. In the section **System Variables**, find the PATH environment variable and select it. Click **Edit**. If the PATH environment variable does not exist, click New.
4. In the **Edit System Variable** (or **New System Variable**) window, specify the value of the PATHenvironment variable. Click **OK**. Close all remaining windows by clicking **OK**.
5. Reopen Command prompt window, and run your java code.

Compiling and running a simple program



Save the file as FirstProgram.java

Open command prompt and do following



Class declaration

The first line

class FirstProgram

Declares a class, which is an object-oriented construct. Java is a true object-oriented language and therefore, everything must be placed inside a class, class is a keyword and declares that a new class definition follows. FirstProgram is a java identifier that specifies the name of the class to be defined.

Opening brace

Every class definition in java begins with an opening brace "{" and ends with a matching closing brace "}", appearing in the last line in the example.

the main line

The third line

public static void main (String args[])

Defines a method named main. Conceptually, this is similar to the main() function in C. Every Java application program must include the main() method. This is the starting point for the interpreter to begin the execution of the program. A Java application can have any number of classes but only one of them must include a main method to initiate the execution. (Note that Java applets will not use the main method at all).

This line contains a number of keywords, public, static and void.

public: The keyword public is an access specifies that declares the main method as unprotected and therefore making it accessible to all other classes.

static: static declares this method as one that belongs to the entire class and not a part of any objects of the class. The main must always be declared as static since the interpreter uses this method before any objects are created.

void: The type modifier void states that the main method does not return any value (but simply prints some text to the screen).

All parameters to a method are declared inside a pair of parentheses. Here String args[] declares a parameter named args, which contains an array of objects of the class type String.

:

Unit -2

Tokens, Expressions and Control Structures

Data types in Java

Data type is a special keyword used to allocate sufficient memory space for the data, in other words Data type is used for representing the data in main memory (RAM) of the computer. Data type represents the different values to be stored in the variable. Every data type has range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric, values, characters and Boolean values. In general, every programming language is containing three categories of data types. They are

- Fundamental or Primitive data types
- Derived data types
- User defined data types

Primitive types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:

- **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.
- **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an int is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Integer

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit*, which defines the *sign* of an integer value.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least used Java type. Here are some examples of **short** variable declarations:

```
short s;

short t;
```

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Although you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case. The reason is that when **byte** and **short** values are used in an expression, they are *promoted* to **int** when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, **int** is often the best choice when an integer is needed.

long

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed.

Floating point types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating point type. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
Double	64	4.9 e–324 to 1.8e+308
Float	32	1.4 e–045 to 3.4e+038

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.
For example, **float** can be useful when representing dollars and cents.
Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

characters

Java uses *Unicode* to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1,

ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits.

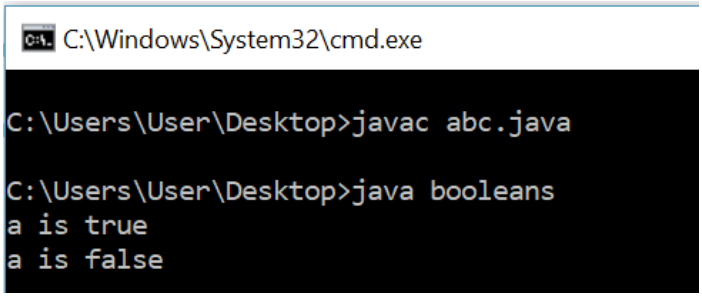
Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a<b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

abc - Notepad

File Edit Format View Help

```
class booleans
{
public static void main(String args[])
{
boolean a=true;
System.out.println("a is "+a);
a=false;
System.out.println("a is "+a);
}
}
```

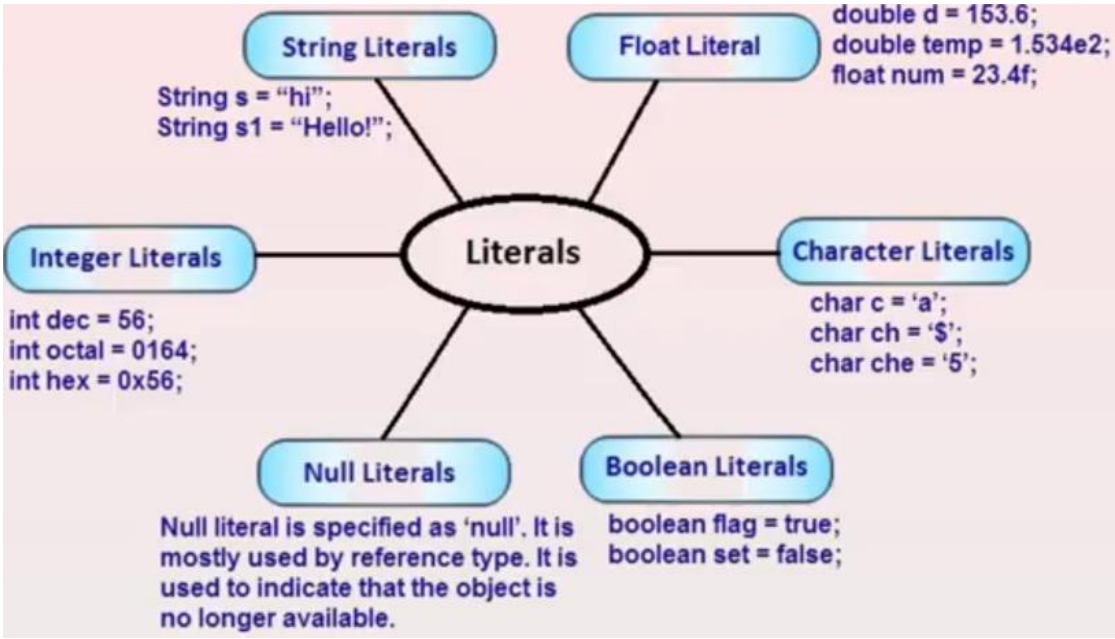


User defined data types

Java allows to create own data types using primitive data types known as composite data type. As composite data types are defined by users these are also called user defined data types. Sometimes some logically related elements needed to be used under a single unit. To handle and serve to such situations, we create own data type through classes.

Literals

Values assigned to variable is called literals.



Identifiers

Identifiers are the name given to variables, methods, classes, packages and interfaces.

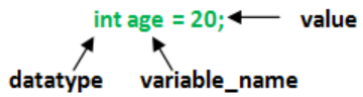
Rules for naming an identifiers

1. The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign) and '**_**' (underscore).
2. Identifiers should **not** start with digits(**[0-9]**). For example "123geeks" is a not a valid java identifier.
3. Java identifiers are **case-sensitive**.
4. Whitespaces are not allowed
5. There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
6. **Reserved Words** can't be used as an identifier. For example "int while = 20;" is an invalid statement as while is a reserved word. There are **53** reserved words in Java.

Variables

A variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.



Types of variables

1. **Local variable**

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

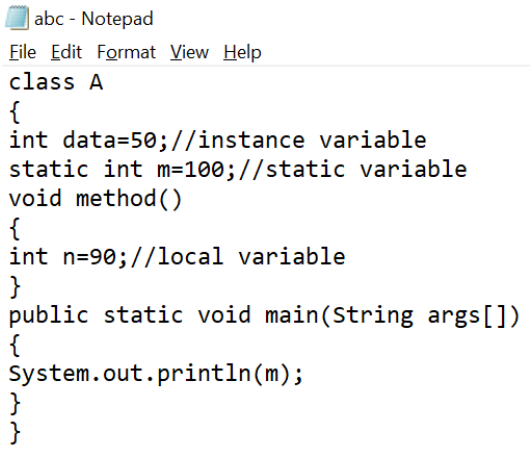
2. **Instance variable**

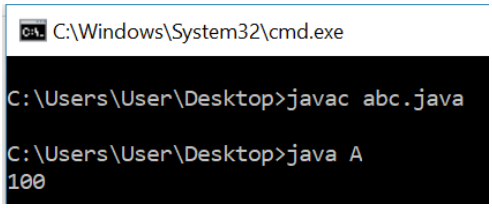
A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static. It is called instance variable because its value is instance specific and is not shared among instances.

3. **Static variable**

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

Example





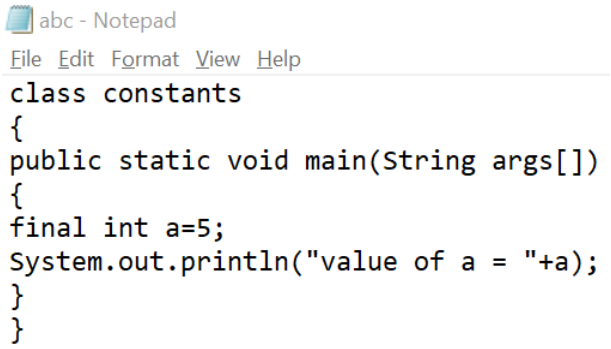
Standard default values

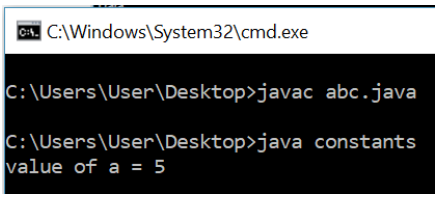
In java, every variable has a default value. If we don’t initialize a variable when it is first created, java provides default value to that variable type automatically.

Type of variable	Default value
byte	Zero: (byte)0
short	Zero: (short)0
int	Zero: 0
long	Zero: 0l
float	0.0f
double	0.0d
char	Null character
boolean	False
reference	null

Constants

Constants are the variables whose value cannot be changed during execution of program. In java we use keyword final to denote a constant.





Type conversion in Java

When we assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

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

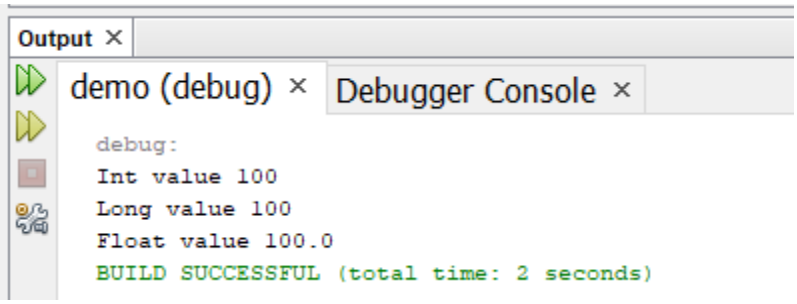
- The two data types are compatible.

- When we assign value of a smaller data type to a bigger data type.

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

byte → short → int → long → float → double

```
public class typeconversion {
    public static void main(String args[])
    {
        int i = 100;

        //automatic type conversion
        long l = i;

        //automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
}
}
```



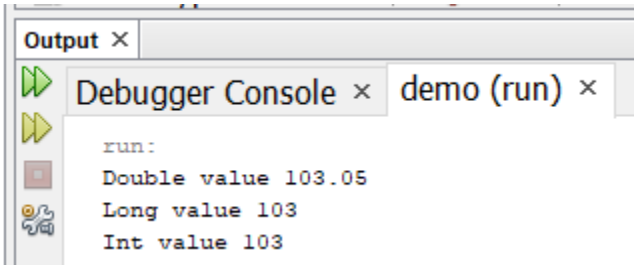
Narrowing or Explicit Conversion

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

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

double → float → long → int → short → byte

```
public class typeconversion {
    public static void main(String args[])
    {
        double d = 103.05;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);

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



Array

Array is an object, which stores a group of elements of the same data type. Array is index based and the first element of an array starts with the index 0. The size of array is fixed. Array is Index based and hence accessing a random element and performing any operations over the elements such as sorting, filling etc, can be easily performed. The size of an array is fixed and hence cannot grow or shrink at runtime.

Types of arrays

Single dimensional array

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

Syntax for declaration

datatype[] array_name;

syntax for instantiation

array_name=new datatype[size];

Declaration and instantiation can be done in a single line

datatype array_name[]=new datatype[size];

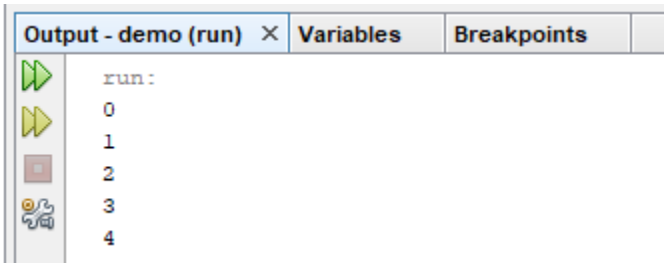
Array initialization

datatype array_name[]= {value1,value2,…..};

int arr[]={45,56,67,78,32};

example

```
public class arrays {
    public static void main(String args[])
    {
 int arr[]=new int[5];
 int i;
 for(i=0;i<arr.length;i++)
     arr[i]=i;
 for(i=0;i<arr.length;i++)
     System.out.println(arr[i]);
}
}
```



Multidimensional array

Multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

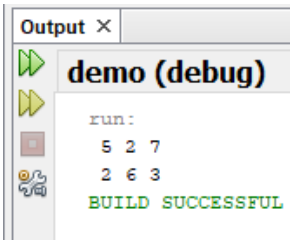
Declaration

datatype array_name[][]=new datatype[][];

int mat[][] = new int[3][4];

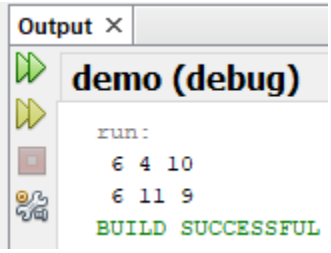
Example

```
public class multidimensionalarray {
    public static void main(String args[])
    {
 int arr[][]={
        {5,2,7},
        {2,6,3}
    };
 int i,j;
 for(i=0;i<2;i++)
 {
     for(j=0;j<3;j++)
     {
         System.out.print(" "+arr[i][j]);
     }
     System.out.println();
    }
    }
}
```



Write a program to display the sum of two matrix

```
public class multidimensionalarray {
    public static void main(String args[])
    {
 int a[][]={{5,2,7},{2,6,3}};
 int b[][]={{1,2,3},{4,5,6}};
 int c[][]=new int[2][3];
 int i,j;
 for(i=0;i<2;i++)
 {
     for(j=0;j<3;j++)
     {
         c[i][j]=a[i][j]+b[i][j];
         System.out.print(" "+c[i][j]);
     }
     System.out.println();
    }
    }
}
```



Comment

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code. Single-line comments start with two forward slashes (//). Any text between // and the end of the line is ignored by Java (will not be executed). Multi-line comments start with /* and ends with */. Any text between /* and */ will be ignored by Java.

Garbage collection

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.

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

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.

How can an object be unreferenced?

There are many ways:

- By nulling the reference

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

- 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
```

- By anonymous object etc.

```
new Employee();
```

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()
{
}
```

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()
{
}
```

Example

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

run:

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

Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions.

Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators

1. Arithmetic operator

Arithmetic operators are used to construct mathematical expressions as in algebra. Java provides all the basic arithmetic operators.

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division (Remainder)

Integer arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value. If a and b are integers, then a=14 and b=4 we have the following results:

```
a+b=18
a-b=10
a*b=56
a/b=3 (decimal part truncated)
a%b=2 (remainder of integer division)
```

Real arithmetic

An arithmetic operation involving only real operands is called real arithmetic. An real operand may assume values either in decimal or exponential notation.

Unlike C and C++, modulus operator % can be applied to the floating point data as well. The floating point modulus operator returns the floating point equivalent of an integer division.

```
public class floatpoint {
    public static void main(String args[])
    {
    float a=20.5f, b=6.4f;
    System.out.println("a = "+a);
    System.out.println("b = "+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("a/b = "+ (a/b));
```

```
}
run:
a = 20.5
b = 6.4
a+b = 26.9
a-b = 14.1
a*b = 131.2
a/b = 3.203125
```

Mixed mode arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed mode arithmetic expression. If either operands is of the real type, then the other operand is converted to real and the real arithmetic is performed. The result will be a real .

```
15/10.0   produces the result 1.5
Whereas
15/10     produces the result 1
```

2. Relational operator

Relational operator are used to compare two values or two expressions.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Example

```
public class relationaloperator {
    public static void main(String args[])
    {
        int a=5,b=3,c=2;
        System.out.println("value of a = " +a);
        System.out.println("value of b = " +b);
        System.out.println("value of c = " +c);
        System.out.println("a>b is "  +(a>b));
        System.out.println("b<c is "  +(b<c));
        System.out.println("a==b is "  +(a==b));
        System.out.println("a!=b is "  +(a!=b));
        System.out.println("a==b+c is "  +(a==b+c));
    }
}
```

run:

```
value of a = 5
value of b = 3
value of c = 2
a>b is true
b<c is false
a==b is false
```

```
a!=b is true
a==b+c is true
```

3. Logical operator

Logical operators are used to combine two or more relational expressions or determine logic between variables or values.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

4. Assignment operator

Assignment operators are used to assign the value of an expression to a variable. We can have multiple assignment expressions also for example;

x = y= z= 20

Here all the three variable x, y, z will be assigned value 20, and the value of whole expressions will be 20.

When the variable on the left hand side of the assignment operator also occurs on right hand side then we can avoid writing the variable twice by using compound assignment operator. For example:

x = x +5

can be also written as x += 5

Here += is a compound assignment operator.

Similarly we have other compound assignment operators:

x -= 5 is equivalent to x = x - 5

y *= 5 is equivalent to y = y * 5

sum / = 5 is equivalent to sum = sum/5

k% = 5 is equivalent to k = k % 5.

5. Increment/Decrement operator

Java has two useful operator increment (++) and decrement (--). These are unary operators because they operate only the single operand. The increment operator (++) increments the value of the variable by one and decrement operator (--) decrements the value of variable by 1.

++x is equivalent to x = x + 1

--x is equivalent to x = x -1

These operators should be used only with variables; they can't be used with constant for expressions. For example the expression ++5 or ++(x+y+z) are invalid.

These operators are of two types:

i. Prefix increment / decrement - operator is written before operand. eg. ++x or --x

ii. Postfix increment/decrement - operator is written after operand eg. x++ or x --

Prefix Increment/decrement:

Here first the value of variable is incremented/decremented then the new value is used in the operation. Let's us take a variable x whose value is 3. The statement y = ++x; means first increment of the value of x by 1, then assign the value of x to y. This single statement is equivalent to these two statements;

x = x + 1;

y = x;

Hence, now the value of x is 4 and value of y is 4.

The statement y = --x; means first decrement the value of x by 1 then assign the value of x to y. This single statement is equivalent to these two statements;

x = x - 1;

y = x

Hence, now the value of x is 3 and value of y is 3 .

Postfix Increment/Decrement:

Here first the value of variable is used in the operation and then increment/decrement is performed. Let us take a variable whose value is 3.

The statement y = x ++; means first the value of x is assigned to y and then x is incremented by 1. This statement is equivalent to these two statements;

y = x ;

x = x + 1;

Hence, now value of x is 4 and value of y is 3.

The statement y = x--; means first the value of x is assigned to y and then x is decremented by 1. This statement is equivalent to these two statements;

y = x;

x = x -1;

Hence, now the value of x is 3 and value of y is 4.

Example

```
public class incrementdecrement {
    public static void main(String args[])
    {
int a=5,b=10;
System.out.println("value of a = "+ a +" value of b = " + b);
System.out.println("value of a = "+ ++a +" value of b = "+ b--);
System.out.println("value of a = "+ a +" value of b = "+ b);
System.out.println("value of a = "+ a-- +" value of b = "+ ++b);
System.out.println("value of a = "+ a +" value of b = "+ b);
System.out.println("value of a = "+ --a +" value of b = "+ --b);
```

```
System.out.println("value of a = "+ a +" value of b = "+ b);
System.out.println("value of a = "+ a++ +" value of b = "+ b++);
System.out.println("value of a = "+ a +" value of b = "+ b);
    }
}
```

run:

```
value of a = 5 value of b = 10
value of a = 6 value of b = 10
value of a = 6 value of b = 9
value of a = 6 value of b = 10
value of a = 5 value of b = 10
value of a = 4 value of b = 9
value of a = 4 value of b = 9
value of a = 4 value of b = 9
value of a = 5 value of b = 10
```

6. Conditional operator (Ternary Operator)

Conditional operator is a ternary operator (? and -) which requires three expressions as operands. This is written as-

Test expression ? expression 1: expression2

First the test expression is evaluated,

If testexpression is true (nonzero), then expression1 is evaluated and it becomes the value of the overall conditional expression.

If test expression if false(zero), then expression2 is evaluated and it becomes the value of overall conditional expression.

For example consider this conditional expression-

a > b ? a : b

Here first the expression a > b is evaluated, if the value is true then the value of variable a becomes the value of conditional expression otherwise the value of b becomes the value of conditional expression.

Example

```
public class conditionaloperator {
    public static void main(String args[])
    {
 int a=5,b=2,max;
 max=a>b?a:b;
 System.out.println("largest value = " + max);
}
}
```

run:

```
largest value = 5
```

7. Bitwise operator

Bitwise operators are used for operations on individual bits. Bitwise operators operate on integers only. The bitwise operators are as-

Bitwise operator	Meaning
&	bitwise AND
|	bitwise OR
~	one's complement
<<	left shift
>>	right shift
>>>	shift right with zero fill
^	bitwise XOR

Example

```
public class bitwiseoperator {
    public static void main(String args[])
    {
        int a=5,b=2;
        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); // ~a
        System.out.println(a<<2); // a<<2
        System.out.println(b>>2); // b>>2
        System.out.println(a>>>2); // a>>>2
    }}
```

run:

```
0
7
7
-6
20
0
1
```

8. Instance of operator

The `instanceof` is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.

Example

```
person  instanceof  student
```

is `true` if the object `person` belongs to the class `student` otherwise it is `false`.

Control statements

Control statements enable us to specify the flow of program control; ie, the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

Branching or **Selection** statement allow our program to choose different paths of execution based upon the outcome of an expression or the state of a variable.

i. if statement

The if statement is used to express conditional expressions. The general form of if statement is;

For only one statement we don't need braces.

```
if(condition)
statement;
```

For more than one statements we need braces;

```
if(condition)
{
statement1;
statement2;
……..
statementN;
}
```

Example

```
public class ifstatement {

    public static void main(String args[])
    {
        int sales=5000;
        float commission=0.0f;
        if(sales>3000)
            commission=sales*0.5f;
        System.out.println(commission);
    }
    }
```

run:

```
2500.0
```

ii. if else

This is a bi-directional conditional control statement. This statement is used to test a condition and take one of the two possible actions. If the condition is true then a single statement or a block of statements is executed (one part of the program), otherwise another single statement or a block of statements is executed (other part of the program).

For single line

```
if( condition)
statement1;
else
statement2;
```

For multiline

```
if(condition)
{
statement;
……..;
}
else
{
statement;
……;
}
```

Example

```
public class ifelse {
     public static void main(String args[])
    {
        int a=5;
        if(a%2==0)
            System.out.println(a+ " is even");
        else
            System.out.println(a+ " is odd");
    }
}
```

run:

```
5 is odd
```

iii. nested if else

If we declare another if…else statement in the block of if or the else block. This is known as nesting if….else statements.

```
if(condition)
 {
     if(condition)
         statement;
          else
        statement;
 }
else
    {
     if(condition)
         Statement;
      else
       statement;
}
```

Example

```
public class nestedif {
public static void main(String args[])
{
 int a=5,b=2,c=7;
 if(a>b)
 {
     if(a>c)
     System.out.println(a+ " is largest");
     else
     System.out.println(c+ " is largest");
 }
 else
 {
     if(b>c)
     System.out.println(b+ " is largest");
     else
     System.out.println(c+ " is largest");
 }
 }
}
```

run:

```
7 is largest
```

iv. if else if ladder

This is the type of nesting in which there is an if….else statement in every else part except the last else part. This type of nesting is frequently used in programs and is also known as else if ladder.

```
if(condition)
statement; else
if(condition)
statement; else
if(condition)
statement;
.
.
.
else   statement;
```

Example

```
public class ifelseifladder {
    public static void main(String args[])
    {
float per=65.5f;
if(per>=75)
    System.out.println("Distinction");
else
    if(per>=60)
        System.out.println("First Division");
else
        if(per>=45)
            System.out.println("Second Division");
else
            System.out.println("Third division");
}
}
```

run:

```
First Division
```

v. switch statement

This is a multi-directional condition control statements. Sometimes there is a need in program to make choice among number of alternatives. For making this choice, we use the switch statements. This can be written as:-

```
switch (expression) {
case value1:
      // statement sequence
        break;
case value2:
      // statement sequence
       break;
.
.
```

```
.
case valueN :
      // statement sequence
       break;
default:
     // default statement sequence
}
```

Example

```
public class switchcase {
    public static void main(String args[])
    {
    int a=3;
switch(a)
{
            case 1:
          System.out.println("Sunday");
          break;
            case 2:
          System.out.println("Monday");
          break;
            case 3:
          System.out.println("Tueday");
          break;
            case 4:
          System.out.println("Wednesday");
          break;
            case 5:
          System.out.println("Thusday");
          break;
            case 6:
          System.out.println("Friday");
          break;
            case 7:
          System.out.println("Saturday");
          break;
            default:
          System.out.println("invalid");
}
    }
    }
```

run:

```
Tueday
```

Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops).

i. While loop

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

The syntax of a **while** loop in C programming language is

```
while(condition)
{
   statement(s);
   loopupdate statement;
}
```

Example

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

run:

```
tick 1
tick 2
tick 3
tick 4
tick 5
tick 6
tick 7
tick 8
tick 9
tick 10
```

ii. do while loop

do...while loop is very similar to while loop. Only difference between these two loops is that, in while loops, test expression is checked at first but, in do...while loop code is executed at first then the condition is checked. So, the code are executed at least once in do...while loops.

The syntax of a **do while** loop is

```
do
{
   statement(s);
   loopupdate statement;
}while(condition);
```

Example

```
public class Dowhileloop {
    public static void main(String args[])
    {
        int i=10;
        do
                {
                    System.out.println("tick "+i);
                    i--;
```

```
                }while(i>=0);
        }

    }
```

run:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
tick 0
```

iii. for loop

For statement makes programming more convenient to count iterations of a loop and works well where the number of iterations of the loop is known before the loop entered. The syntax is as follows:

```
for (initialization; test condition; loop update)
  {
  Statement(s);
}
```

Example

```
public class Forloop {
    public static void main(String args[])
    {
    int num=13,factor=0,i;
    for(i=1;i<=num;i++)
    {
        if(num%i==0)
            factor++;
    }
    if(factor==2)
        System.out.println(num+ " is prime");
    else
        System.out.println(num+ " is composite");
}
}
```

run:

```
13 is prime
```

Jump statements allow our program to execute in a nonlinear fashion.

i. break

break statement is used inside loops and switch statements. Sometimes it becomes necessary to come out of loop before its termination. In such a situation, break statement is used to terminate the loop. When break statement is encountered, loop is terminated and the control is transferred to the statement immediately after the loop.

Example

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

run:

```
1
2
3
4
```

ii. continue

continue statement is used when we want to go to the next iteration of the loop after skipping some statements of the loop. It is generally used with a condition. When continue statement is encountered all the remaining statements after continue in the current iteration are not executed and loop continues with the next iteration.

Example

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

run:

```
1
2
3
```

```
4
6
7
8
9
10
```

iii. return

The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**:

Example

```
public class Returns {
    public static void main(String args[])
    {
        boolean t=true;
        System.out.println("Before the return ");
        if(t)
            return;
            System.out.println("This won't execute ");
    }
}
```

run:

```
Before the return
```

Unit-3

Object Oriented Programming Concept

Class in java

A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object. A class in java can contain:

- fields
- methods
- constructors
- blocks
- nested class and interface

A class is a framework that specifies what data and what methods will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. A class is also called user define data type or programmers defined data type because we can define new data types according to our needs. The general syntax of the class definition is given below:

```
class <class_name>
{
    field;
    method;
}
```

Components of class

1. **Modifiers**: A class can be public or has default access.
2. **class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

Declaration of class

A class is declared by the use of the class keyword. The classes that have been used up to this point are very limited illustrations of its complete form. A simplified form of a class definition is shown below:

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

```
        ……..
        type instance-variableN;
        type methodname1(parameter-list){
              //body of the method
        }
        type methodname2(parameter-list){
              //body of the method
        }
        ……..
        ……..
        type methodnamen(parameter-list){
              //body of the method
        }
}
```

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

Adding methods in java

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it. The most important method in Java is the **main()** method. The general syntax of the method definition is given below:

```
Acces_specifier   return_type   method_name (paremeter list){
//method body
}
```

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.

- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Creating objects

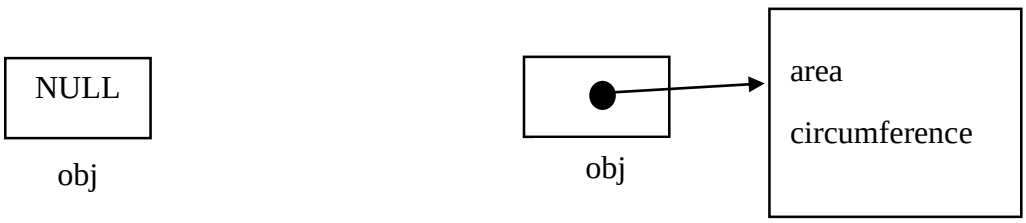
Objects are created using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. The main idea of using **new** is to create the memory that is required to hold an object of the particular type in run time i.e to dynamically allocate the memory at the run time. To create a usable object we must finish two steps:

- Declaring the variable of its type and
- Instantiating the object.

The following example shows the creation of an object of class Area.

```
Area obj;          //declaring the variable of type Area
obj=new Area();  //Instantiating an object
```

The execution of first statement creates the variable that holds reference of the class Area using the name obj. It points nowhere (i.e null) as shown below:



When the second statement is executed then the actual assignment of object reference to the variable is done.

The above two steps are equivalently written as:

```
Area obj=new Area();
```

Example

```
class Circle{
    public void area(int r){
    System.out.println("Area of circle = "+3.14*r*r);
    }
    public void circumference(int r){
    System.out.println("Circumference of circle = "+2*3.14*r);
    }
}
public class CreatingClassAndAddingMethods {
    public static void main(String args[]){
        Circle obj=new Circle();
        obj.area(5);
        obj.circumference(5);
    }
}
```

Output

```
run:
Area of circle = 78.5
Circumference of circle = 31.400000000000002
BUILD SUCCESSFUL (total time: 0 seconds)
```

Method overloading

In java, we can define different methods with the same name but with different parameters (either parameter type or number of parameters). This is one of the examples of polymorphism. If we define many methods with same name but with difference in parameters, then this process is called method overloading. In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters
2. Data type of parameters
3. Sequence of data type of parameters

If two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int,int)

float add(int,int)
```

Example of method overloading

```
public class Area {
    void area(int r){
        System.out.println("Area of circle = "+3.14*r*r);
    }
    void area(int l,int b){
        System.out.println("Area of rectangle = "+l*b);
    }
    public static void main(String args[]){
        Area obj=new Area();
        obj.area(5);
        obj.area(3,6);
    }
}
```

Output

```
run:
Area of circle = 78.5
Area of rectangle = 18
BUILD SUCCESSFUL (total time: 0 seconds)
```

Constructors

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor. 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.

There are basically three rules defined for the constructor.

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

3. A Java constructor cannot be abstract, static, final, and synchronized.

Types of java constructor

There are two types of constructors.

1. No-argument constructor
2. Parameterized constructor

1. **No-argument constructor:** A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates **default constructor (with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.

Example

```
public class A {
    A(){
        System.out.println("This is no argument constructor");
    }
    public static void main(String args[]){
        A obj=new A();
    }
}
```

Output

```
run:
This is no argument constructor
```

Example

```
public class Student {
   String name;
   int rollno;
   void display(){
       System.out.println("Rollno = "+rollno +" Name = "+name);
   }
   public static void main(String args[]){
       Student obj=new Student();
       obj.display();
   }
}
```

Output

```
run:
Rollno = 0 Name = null
```

Example

```
public class A {
    int length;
    int breadth;
    A(){
        length=5;
        breadth=3;
    }
    void area(){
        System.out.println("Area of rectangle = "+length*breadth);
    }
    public static void main(String args[]){
        A obj=new A();
        obj.area();
    }
}
```

Output

```
run:
Area of rectangle = 15
```

2. **Parameterized constructor**

A constructor that has parameter is known as parameterized constructor. If we want to initialize fields of the class with our own values, then we use a parameterized constructor.

Example

```
public class A {
    int length,breadth;
    A(int l,int b){
        length=l;
        breadth=b;
    }
    void area(){
        System.out.println("Area of rectangle = "+length*breadth);
    }
    public static void main(String args[]){
        A obj=new A(4,6);
        obj.area();
    }
}
```

Output

```
run:
Area of rectangle = 24
```

Constructor Overloading

Like methods overloading, we can overload constructors for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters. Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading.

Example

```
public class Area {
    Area(int r){
        System.out.println("Area of circle = "+3.14*r*r);
    }
    Area(int l,int b){
        System.out.println("Area of rectangle = "+l*b);
    }
    public static void main(String args[]){
        Area obj1=new Area(5);
        Area obj2=new Area(3,8);
    }
}
```

Output

```
run:
Area of circle = 78.5
Area of rectangle = 24
```

this keyword

this keyword refers to the current object in a method or constructor. The most common use of the 'this' keyword is to eliminate the confusion between class attributes and parameters with the same name. **this** is a java keyword which act as a reference to the current object within an instance method or a constructor- the object whose method or constructor is being called.

Usage of **this** keyword

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

1. **this can be used to refer current class instance variable.**

```
public class Test
{
    int a;
    int b;

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    void display()
    {
        //Displaying value of variables a and b
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String args[])
    {
        Test obj = new Test(10, 20);
        obj.display();
    }
}
```

Output

```
run:
a = 10  b = 20
```

2. **Using this() to invoke current class constructor**

```
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        this(10, 20);
        System.out.println("Inside  default constructor \n");
    }

    //Parameterized constructor
    Test(int a, int b)
```

```
    {
        this.a = a;
        this.b = b;
        System.out.println("a= "+a+" b= "+b);
    }

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

Output

```
run:
a= 10 b= 20
Inside  default constructor
```

3. Using 'this' keyword to return the current class instance

```
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that returns current class instance
    Test get()
    {
        return this;
    }

    //Displaying value of variables a and b
    void display()
    {
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get().display();
    }
```

```
}
```

Output

```
run:
a = 10   b = 20
```

4. Using 'this' keyword as method parameter

```
class Test
{
    int a;
    int b;

    // Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    // Method that receives 'this' keyword as parameter
    void display(Test obj)
    {
        System.out.println("a = " +obj.a + "  b = " + obj.b);
    }

    // Method that returns current class instance
    void get()
    {
        display(this);
    }

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

Output

```
run:
a = 10   b = 20
```

5. Using 'this' keyword to invoke current class method

```
class Test {
```

```
    void display()
    {
        // calling function show()
        this.show();

       System.out.println("Inside display function");
    }

    void show() {
        System.out.println("Inside show funcion");
    }


    public static void main(String args[]) {
        Test t1 = new Test();
        t1.display();
    }
}
```

Output

```
run:
Inside show funcion
Inside display function
```

6. Using 'this' keyword as an argument in the constructor call

```
class A
{
   B obj;

   // Parameterized constructor with object of B
   // as a parameter
   A(B obj)
   {
         this.obj = obj;

   // calling display method of class B
         obj.display();
   }

}

class B
{
   int x = 5;

   // Default Constructor that create a object of A
   // with passing this as an argument in the
// constructor
   B()
```

```
    {
          A obj = new A(this);
    }

    // method to show value of x
    void display()
    {
          System.out.println("Value of x in Class B : " + x);
    }

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

Output

```
run:
Value of x in Class B : 5
```

Static keyword

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

- Variable
- Method
- Block
- Nested class

Static variable

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

- Static variables are also known as Class Variables.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

Example

```
public class StaticVariable {
      int roll;
      String name;
      static String subject="Java";
      StaticVariable(int r,String n)
```

```
        {
            roll=r;
            name=n;
        }
        void display()
        {
            System.out.println(roll+" "+name+" "+subject);
        }
        public static void main(String args[])
          {
        StaticVariable obj1=new StaticVariable(1,"Anjila");
        StaticVariable obj2=new StaticVariable(2,"Sopin");
        obj1.display();
        obj2.display();
          }
   }
```

Output

```
run:
1 Anjila Java
2 Sopin Java
```

Static method

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

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.
- A static method cannot use non static data member or call non-static method directly.
- The 'this' keyword and 'super' keyword cannot be used in static context.

Example

```
public class StaticMethod {
   static int roll;
      static String name;
      static String subject;
      StaticMethod(int r,String n,String s)
      {
          roll=r;
          name=n;
          subject=s;
      }
      static void display()
      {
          System.out.println(roll+" "+name+" "+subject);
      }
```

```
        public static void main(String args[])
          {
        StaticMethod obj=new StaticMethod(1,"Niruta","Java");
        display();
          }
   }
```

Output

```
1 Niruta Java
```

Static Block

Static block is used for initializing the static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Example

```
   public class StaticBlock {
        static int roll;
        static String name;
        static String subject;
        static
        {
            roll=1;
            name="Rama";
            subject="Java";
        }
        void display()
        {
            System.out.println(roll+" "+name+" "+subject);
        }
        public static void main(String args[])
          {
        StaticBlock obj=new StaticBlock();
        obj.display();
          }
   }
```

Output

```
run:
1 Rama Java
```

Static class

A class can be made **static** only if it is a nested class.

1. Nested static class doesn't need reference of Outer class.
2. A static class cannot access non-static members of the Outer class.

Syntax

```
Class OuterClass
{
   Static class NestedClass
   {

   }
}
```

Example

```
public class OuterClass {
    static class InnerClass
    {
        public void display()
        {
            System.out.println("Inside Nested class");
        }
    }
    public static void main(String args[])
    {
        OuterClass.InnerClass obj=new OuterClass.InnerClass();
        obj.display();
    }
}
```

Output

```
run:
Inside Nested class
```

Passing by value to method

When a parameter is pass-by-value, the caller and the callee method operate on two different variables which are copies of each other. Any changes to one variable don't modify the other. It means that while calling a method, parameters passed to the callee method will be clones of original parameters. Any modification done in callee method will have no effect on the original parameters in caller method.

Example

```
class A
{
int a,b;
    void add(int x,int y)
    {
        a=x+3;
        b=y+3;
    }
}
```

```
public class PassByValue
{
    public static void main(String args[])
    {
        A obj=new A();
        int a=5,b=10;
        System.out.println("Before function call a = "+a+" b = "+b);
        obj.add(a, b);
        System.out.println("After function call a = "+a+" b = "+b);
    }
}
```

Output

```
run:
Before function call a = 5 b = 10
After function call a = 5 b = 10
```

Passing by reference to method

When a parameter is pass-by-reference, the caller and the callee operate on the same object. It means that when a variable is pass-by-reference, the unique identifier of the object is sent to the method. Any changes to the parameter's instance members will result in that change is being made to the original value.

Example

```
public class PassByReference
{
    int a=5,b=10;
    void add(PassByReference ob)
    {
      ob.a=ob.a+3;
     ob.b=ob.b+3;
    }
    public static void main(String args[])
    {
      PassByReference obj=new PassByReference();
      System.out.println("Before function call a = "+obj.a+" b =
"+obj.b);
      obj.add(obj);
      System.out.println("After function call a = "+obj.a+" b =
"+obj.b);
    }
}
```

Output

```
run:
Before function call a = 5 b = 10
After function call a = 8 b = 13
```

Methods that return values

Java method can have zero or more parameters. They may return a value. A method with a void return types means this method cannot return a value. But if we want to return a value from any method then we need to define any return type of the method except void. It returns the value to calling methods.

Example

```
public class MethodReturnValue {
    static int sum(int a,int b)
    {
        return a+b;
    }
    public static void main(String args[])
    {
        System.out.println("sum = " +sum(5,3));
    }
}
```

Output

```
run:
sum = 8
```

Access Control (Access Specifiers)

Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class to specific parts of a program. Access to members of a class can be controlled using the *access modifiers.* There are four access modifiers in Java. They are:

1. public
2. protected
3. default
4. private

If the member (variable or method) is not marked as either *public* or *protected* or *private*, the access modifier for that member will be *default*.

public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Example

```
package pack1;
public class A {
    public void display()
    {
        System.out.println("Object Oriented Programming");
    }
}
package pack2;
import pack1.*;
public class B {
   public static void main(String args[])
   {
       A obj=new A();
       obj.display();
   }
}
```

Output

```
run:
Object Oriented Programming
```

protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Example

```
package pack1;
public class A {
    protected void display()
    {
        System.out.println("Object Oriented Programming");
```

```
    }
}

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

Output

```
run:
Object Oriented Programming
```

default (no specifier)

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public, static, final and the methods in an interface are by default public.

Example

```
package pack1;
class A {
    void display()
    {
        System.out.println("Object Oriented Programming");
    }
    public static void main(String args[])
    {
        A obj=new A();
        obj.display();
    }
}
```

Output

```
run:
Object Oriented Programming
```

Example

```
package pack1;
class A {
    void display()
    {
```

```
        System.out.println("Object Oriented Programming");
    }
}

package pack2;
import pack1.*;
public class B {
   public static void main(String args[])
   {
       A obj=new A();  //compile time error
       obj.display();  //compile time error
   }
}
```

private

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Using private specifier, we can achieve encapsulation and hide data from the outside world.

Example

```
package pack1;
class A {
    private int x=5;
    private void display()
    {
        System.out.println("Object Oriented Programming");
    }
}
public class PrivateSpecifier
{
    public static void main(String args[])
    {
        A obj=new A();
        System.out.println(obj.x);  //compile Time Error
        obj.display();  //compile Time Error
    }
}
```

In this example, we have created two classes A and PrivateSpecifier. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

Access Modifiers	Default	Private	Protected	Public
Accessible inside the class	Yes	Yes	Yes	Yes
Accessible within the subclass inside the same package	Yes	No	Yes	Yes
Accessible outside the package	No	No	No	Yes

Accessible within the subclass outside the package	No	No	Yes	Yes

Nested and Inner Classes

A class defined within another class is known as nested class. The scope of the nested class is bounded by the scope of its enclosing class. We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

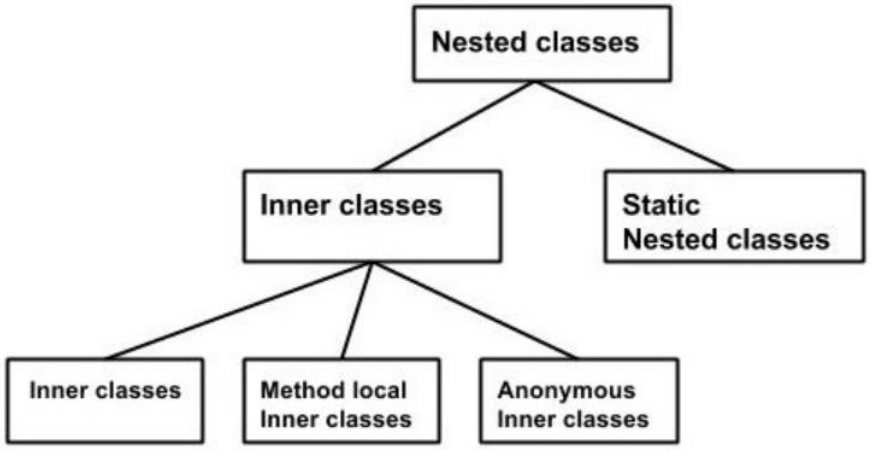
Syntax

```
class Outer   //class Outer members
{
      class Inner   //class Inner members
      {

      }
}
```

Types of Nested classes

There are two types of nested classes that can be created in java.

1. Non-static nested class (inner class)
2. Static nested class



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 we 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
{
    int x=5;
  private class Inner
    {
        void display()
        {
            System.out.println("value of x = "+x);
        }
    }
    void show()
    {
        Inner obj=new Inner();
        obj.display();
    }
}
public class InnerClass
{
public static void main(String args[])
{
    Outer ob=new Outer();
    ob.show();
}
}
```

Output

```
run:
value of x = 5
```

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.

Example

```
public class MethodLocalInnerClass
{
void show()
    {
    int x=5;
  class Inner
    {
       void display()
        {
            System.out.println("value of x = "+x);
        }
```

```
    }
  Inner ob=new Inner();
  ob.display();
    }
public static void main(String args[])
{
    MethodLocalInnerClass obj=new MethodLocalInnerClass();
    obj.show();
}
}
```

Output

```
run:
value of x = 5
```

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 we need to override the method of a class or an interface.

Example

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

Output

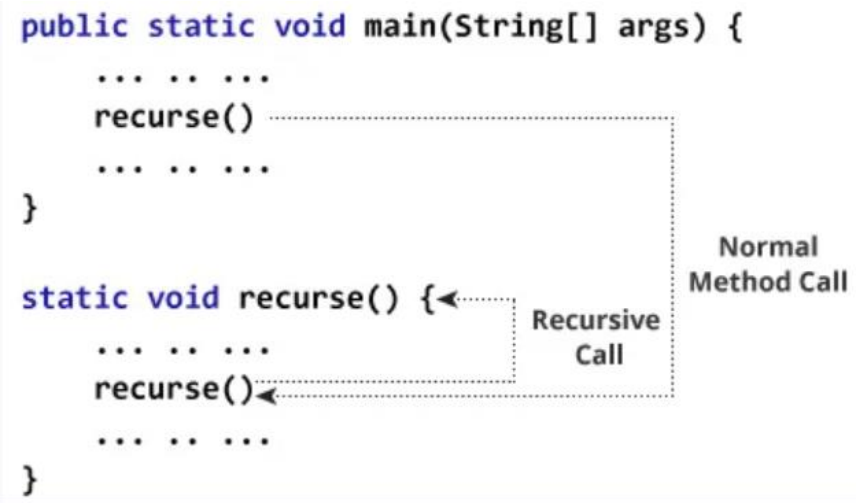
```
run:
This is an example of anonymous inner class
```

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.

Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

How recursion works?



Working of Java Recursion

In the above example, we have called the `recurse()` method from inside the `main` method. (normal method call). And, inside the `recurse()` method, we are again calling the same `recurse()` method. This is a recursive call.

Example: Factorial of a number using recursion

```
import java.util.Scanner;
class Factorial
```

```
{
int fact(int n)
{
    if(n==1)
        return 1;
    else
        return (n*fact(n-1));
}
}
public class Recursion
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        Factorial obj=new Factorial();
        int n;
        System.out.println("Enter a number ");
        n=sc.nextInt();
        System.out.println("Factorial of "+n+" is "+ obj.fact(n));
    }
}
```

Output

```
run:
Enter a number
5
Factorial of 5 is 120
```

Let's trace the evaluation of fact(5)

fact(5) =
5*fact(4) =
5*(4*fact(3)) =
5*(4*(3*fact(2))) =
5*(4*(3*(2*fact(1)))) =
5*(4*(3*(2*(1*fact(0))))) =
5*(4*(3*(2*(1*1)))) =
5*(4*(3*(2*1))) =
5*(4*(3*2)) =
5*(4*6) =
5*24 =
120

Example: Fibonacci series using recursion

```
import java.util.Scanner;
class Fibonacci
{
int fib(int n)
{
    if(n<=1)
        return n;
    else
        return (fib(n-1)+fib(n-2));
}
}
public class Recursion
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        Fibonacci obj=new Fibonacci();
        int i,n;
        System.out.println("Enter a number ");
        n=sc.nextInt();
        for(i=0;i<n;i++)
        {
        System.out.print(obj.fib(i)+"\t");
        }
    }
}
```

Output

```
run:
Enter a number
5
0       1       1       2       3
```

Unit-4

Inheritance and Packaging

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends keyword

`extends` is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

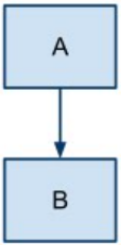
```
class base
{
    .....
    .....
}
class Sub extends base
{
    .....
    .....
}
```

Types of inheritances

1. Single inheritance
2. Multi-level inheritance
3. Hierarchical inheritance

1. Single inheritance

When a single class gets derived from its base class, then this type of inheritance is termed as single inheritance.



Syntax

```
class A
{
    .....
    .....
}
class B extends A
{
    .....
    .....
}
```

Example

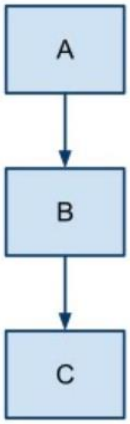
```
import java.util.Scanner;
class A
{
    String name;
    int age;
    double salary;
    Scanner sc=new Scanner(System.in);
    void get()
    {
        System.out.println("Enter name, age and salary");
        name=sc.nextLine();
        age=sc.nextInt();
        salary=sc.nextDouble();
    }
}
class B extends A
{
    void display()
    {
        System.out.println("Name = "+name);
        System.out.println("Age = "+age);
        System.out.println("Salary = "+salary);

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

Output

```
run:
Enter name, age and salary
Subal
27
48000
Name = Subal
Age = 27
Salary = 48000.0
```

2. Multilevel Inheritance

In this type of inheritance, a derived class gets created from another derived class and can have any number of levels.



Syntax

```
class A
{
    .....
    .....
}
class B extends A
{
    .....
    .....
}
class C extends B
{
    .....
    .....
}
```

Example

```
import java.util.Scanner;
class A
{
    String name;
    Scanner sc=new Scanner(System.in);
    void getName()
    {
        System.out.println("Enter Name ");
        name=sc.nextLine();
    }
```

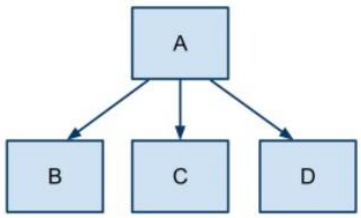
```
    void displayName()
    {
        System.out.println("Name = "+name);
    }
}
class B extends A
{
    int age;
    void getAge()
    {
        System.out.println("Enter Age ");
        age=sc.nextInt();
    }
    void displayAge()
    {
        System.out.println("Age = "+age);
    }
}
class C extends B
{
    double salary;
    void getSalary()
    {
        System.out.println("Enter Salary ");
        salary=sc.nextDouble();
    }
    void displaySalary()
    {
        System.out.println("Address = "+salary);
    }
}
public class Multilevel
{
public static void main(String args[])
{
    C obj=new C();
    obj.getName();
    obj.getAge();
    obj.getSalary();
    obj.displayName();
    obj.displayAge();
    obj.displaySalary();
}
}
```

Output

```
run:
Enter Name
Subin
Enter Age
26
Enter Salary
43000
Name = Subin
Age = 26
Address = 43000.0
```

3. Hierarchical inheritance

In this type, two or more classes inherit the properties of one existing class. When more than one classes are derived from a single base class, such inheritance is known as Hierarchical Inheritance.



Syntax

```
class A
{
    .....
    .....
}
class B extends A
{
    .....
    .....
}
class C extends A
{
    .....
    .....
}
class D extends A
{
    .....
    .....
}
```

Example

```
import java.util.Scanner;
class A
{
    String name;
```

```
    int age;
    double salary;
    Scanner sc=new Scanner(System.in);
}
class B extends A
{
    void getName()
    {
        System.out.println("Enter Name ");
        name=sc.nextLine();
    }
    void displayName()
    {
        System.out.println("Name = "+name);
    }
}
class C extends A
{
    void getAge()
    {
        System.out.println("Enter Age ");
        age=sc.nextInt();
    }
    void displayAge()
    {
        System.out.println("Age = "+age);
    }
}
class D extends B
{
    void getSalary()
    {
        System.out.println("Enter Salary ");
        salary=sc.nextDouble();
    }
    void displaySalary()
    {
        System.out.println("Address = "+salary);
    }
}
public class Hierarchical
{
public static void main(String args[])
{
    B obj1=new B();
    C obj2=new C();
    D obj3=new D();
    obj1.getName();
    obj2.getAge();
```

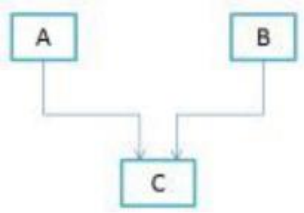
```
    obj3.getSalary();
    obj1.displayName();
    obj2.displayAge();
    obj3.displaySalary();
}
}
```

Output

```
run:
Enter Name
Subal
Enter Age
23
Enter Salary
45000
Name = Subal
Age = 23
Address = 45000.0
```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and we call if from child class object, there will be ambiguity to call the method of A and B class. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.




```
class A
{
    void display()
    {
        System.out.println("This is display function in class A");
    }
}
class B
{
    void display()
    {
        System.out.println("This is display function in class B");
    }
}
```

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

‘super’ keyword

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usages of super keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1. super can be used to refer immediate parent class instance variable.

When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

Example

```
class A
{
    int num=5;
}
class B extends A
{
int num=10;
void display()
{
    System.out.println("Value of num = "+super.num);
    System.out.println("Value of num = "+num);
}
public static void main(String args[])
{
    B obj=new B();
    obj.display();
```

```
}
}
```

Output

```
run:
Value of num = 5
Value of num = 10
```

2. super can be used to invoke immediate parent class method.

The super keyword can be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Example

```
class A
{
    void display()
    {
        System.out.println("Display function in class A");
    }
}
class B extends A
{
    void display()
    {
        System.out.println("Display function in class B");
    }
    void show()
    {
        super.display();
        display();
    }
public static void main(String args[])
{
    B obj=new B();
    obj.show();
}
}
```

Output

```
run:
Display function in class A
Display function in class B
```

3. super() can be used to invoke immediate parent class constructor.

When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class. So the order of execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super() (this invokes the no-argument constructor of parent class) as the first statement in the constructor of child class.

Example

```
class A
{
    A(int x)
    {
        System.out.println("x = "+x);
    }
}
class B extends A
{
    B(int m,int n)
    {
        super(m);   //invokes parent class constructor
        System.out.println("y = "+n);
    }
public static void main(String args[])
{
    B obj=new B(5,10);
}
}
```

Output

```
run:
x = 5
y = 10
```

Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usages of method overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

- Method overriding is used for runtime polymorphism.

Example

```
class A
{
    void display()
    {
        System.out.println("Display function in class A");
    }
}
class B extends A
{
    void display()
    {
        System.out.println("Display function in class B");
    }
public static void main(String args[])
{
    A obj1=new A();
    A obj2=new B();
    obj1.display();
    obj2.display();
}
}
```

Output

```
run:
Display function in class A
Display function in class B
```

Difference between method overloading and method overriding in java

Method overloading	Method overriding
1. In the method overloading, methods or functions must have the same name and different signatures.	1. In the method overriding, methods or functions must have the same name and same signatures.
2. Method overloading is a compile time polymorphism.	2. Method overriding is a run time polymorphism.
3. It help to rise the readability of the program.	3. While it is used to grant the specific implementation of the method which is already provided by its parent class or super class.
4. It is occur within the class.	4. While it is performed in two classes with inheritance relationship.
5. In java, method overloading can't be performed by changing return type of	5. Return type must be same or covariant in method overriding.

the method only. Return type can be same or different in method overloading. But we must have to change the parameter.	

Example
`class A`
`{`
`  int add(int a,int b)`
`  {`
`  return a+b;`
`  }`
`  int add(int a,int b,int c)`
`  {`
`  return a+b+c;`
`  }`
`}`
	Example
	`class A`
	`{`
	`  void display()`
	`  {`
	`  System.out.println("parent class");`
	`  }`
	`}`
	`class B extends A`
	`{`
	`  void display()`
	`  {`
	`  System.out.println("child class");`
	`  }`
	`}`

Runtime polymorphism or dynamic method dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. This is the example of run-time polymorphism. Runtime polymorphism in java is achieved by method overriding in which a child class overrides a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method. This method call resolution happens at runtime and is termed as dynamic method dispatch mechanism.

```
class A
{
    void display()
    {
        System.out.println("Display function in class A");
    }
}
class B extends A
{
    void display()
    {
        System.out.println("Display function in class B");
    }
public static void main(String args[])
{
    A obj1=new A();    // class A reference and object
    A obj2=new B();    // class A reference but class B object
    obj1.display();    // runs the method of class A
    obj2.display();    // runs the method of class B
}}
```

Output

```
run:
Display function in class A
Display function in class B
```

In the above example, we can see that even though obj2 is type of A it runs the display method in the B class. The reason for this is in compile time the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since A class has the method display. Then, at the runtime, it runs the method specific for that object.

Difference between compile time polymorphism and run time polymorphism

Compile time polymorphism	Run time polymorphism
1. In Compile time Polymorphism, the call is resolved by the compiler.	1. In Run time Polymorphism, the call is not resolved by the compiler.
2. It is also known as Static binding, Early binding and overloading as well.	2. It is also known as Dynamic binding, Late binding and overriding as well.
3. Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	3. Method overriding is the runtime polymorphism having same method with same parameters or signature, but associated in different classes.
4. It is achieved by function overloading and operator overloading.	4. It is achieved by virtual functions and pointers.
5. It provides fast execution because the method that needs to be executed is known early at the compile time.	5. It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime.
6. Compile time polymorphism is less flexible as all things execute at compile time.	6. Run time polymorphism is more flexible as all things execute at run time.

The Object class in java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java. The Object class is beneficial if you want to refer any object whose type you don't know.

Object class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class, then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore, the Object class

methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Abstract class

A class which is declared with the abstract keyword is known as an abstract class in java. It can have abstract and non-abstract methods (method with the body).

If we declare a class as abstract, then it is necessary for you to subclass it so as to instantiate the object of that class i.e. we cannot instantiate object of abstract class without creating its subclass. The keyword abstract is used to create a class as an abstract class and it can contain abstract methods. Likewise, if you need your method to be always overridden before it can be used, you can declare the method as an abstract method using abstract keyword and without the method

definition i.e. end it with semicolon. Remember, if you declare some method as an abstract method, you must declare your class as abstract.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method

Example

```
abstract class Bank
{
abstract int getRateOfInterest();
}
class NBL extends Bank
{
    int getRateOfInterest()
        {
            return 7;
        }
}
class EBL extends Bank
{
     int getRateOfInterest()
        {
            return 8;
        }
}
class Abstraction
{
public static void main(String args[])
{
Bank obj1=new NBL();
System.out.println("Interest Rate of Nepal Bank Limited is :
"+obj1.getRateOfInterest()+"%");
Bank obj2=new EBL();
System.out.println("Interest Rate of Everest Bank Limited is :
"+obj2.getRateOfInterest()+"%");
}
}
```

Output

```
run:
Interest Rate of Nepal Bank Limited is : 7%
Interest Rate of Everest Bank Limited is : 8%
```

Interface in java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods. The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling

An interface is declared by using interface keyword. It provides total abstraction, means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface Bank
{
int getRateOfInterest();
}
class NBL implements Bank
{
   public int getRateOfInterest()
        {
            return 7;
        }
}
class EBL implements Bank
{
   public int getRateOfInterest()
        {
            return 8;
        }
}
class Abstraction
{
public static void main(String args[])
```

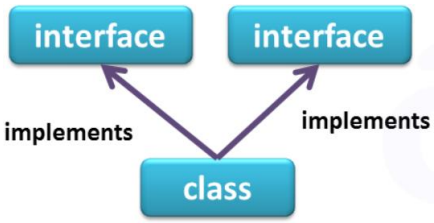
```
{
Bank obj1=new NBL();
System.out.println("Interest Rate of Nepal Bank Limited is :
"+obj1.getRateOfInterest()+"%");
Bank obj2=new EBL();
System.out.println("Interest Rate of Everest Bank Limited is :
"+obj2.getRateOfInterest()+"%");
}
}
```

Output

```
run:
Interest Rate of Nepal Bank Limited is : 7%
Interest Rate of Everest Bank Limited is : 8%
```

Multiple inheritance in java by interface

If a class implements multiple interface, or an interface extends multiple interfaces then it is known as multiple inheritance.



Example

```
interface A
{
    void circle();
}
interface B
{
    void rectangle();
}
class C implements A,B
{
    public void circle()
    {
        System.out.println("Draw Circle");
    }
    public void rectangle()
    {
        System.out.println("Draw Rectangle");
    }
```

```
    public static void main(String args[])
    {
        C obj=new C();
        obj.circle();
        obj.rectangle();
    }}
```

Output

```
run:
Draw Circle
Draw Rectangle
```

Difference between abstract class and interface

Abstract class	Interface
1. Abstract class can **have abstract and non-abstract** methods.	1. Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2. Abstract class **doesn't support multiple inheritance**.	2. Interface **supports multiple inheritance**.
3. Abstract class **can have final, non-final, static and non-static variables**.	3. Interface has **only static and final variables**.
4. Abstract class **can provide the implementation of interface**.	4. Interface **can't provide the implementation of abstract class**.
5. The **abstract keyword** is used to declare abstract class.	5. The **interface keyword** is used to declare interface.
6. An **abstract class** can extend another Java class and implement multiple Java interfaces.	6. An **interface** can extend another Java interface only.
7. An **abstract class** can be extended using keyword "extends".	7. An **interface** can be implemented using keyword "implements".
8. A Java **abstract class** can have class members like private, protected, etc.	8. Members of a Java interface are public by default.

Final classes

In Java, the **`final` keyword** can be used while declaring an entity. Using the final keyword means that the value can't be modified in the future. The java final keyword can be used in many contexts. Final can be:

1. variable: to create a constant variable
2. method: to prevent method overriding
3. class: to prevent inheritance

when a class is declared with final keyword, it is called a final class. A final class cannot be extended (inherited). A method or a class is declared to be final using the final keyword. Though a final class cannot be extended, it can extend other classes. In simple words, a final class can be a sub class but not a super class.

Example

```
final class A
{
  final int a=5;
 final void display()
 {
     System.out.println("Value of a = "+a);
 }
}
public class FinalClass
{
    public static void main(String args[])
    {
        A obj=new A();
        obj.display();
    }
}
```

Output

```
run:
Value of a = 5
```

There are two uses of final class:

1. To prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float etc. are final classes. We cannot extend them.
2. To create an immutable class like the predefined String class. We cannot make a class immutable without making it final.

Difference between abstract class and final class

Abstract class	Final class
1. Uses the “abstract” key word.	1. Uses the “final” key word.
2. This helps to achieve abstraction.	2. This helps to restrict other classes from accessing its properties and methods.

3. For later use, all the abstract methods should be overridden	3. Overriding concept does not arise as final class cannot be inherited
4. A few methods can be implemented and a few cannot	4. All methods should have implementation
5. Cannot create immutable objects (infact, no objects can be created)	5. Immutable objects can be created (eg. String class)
6. Abstract class methods functionality can be altered in subclass	6. Final class methods should be used as it is by other classes
7. Can be inherited	7. Cannot be inherited
8. Cannot be instantiated	8. Can be instantiated

Package

A **java package** is a group of similar types of classes, interfaces and sub-packages. **Package** in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. There are two types of package:

- User defined package: The package we create is called user-defined package.
- Built-in package: The possible java build in packages are: java.lang, java.io, java.awt, java.math, java.sql, java.util, java.time, java.rmi, java.security, java.net, java.nio, java.javax, java.text, java.applet.

Advantages of Java Package

1) Preventing naming conflicts: We can define two classes with the same name in different packages so to avoid name collision, we can use packages
2) Making searching/locating and usage of classes, interfaces, enumerations and annotations easier.
3) Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
4) Packages can be considered as data encapsulation (or data-hiding).

Creating a package

Creating a package in java is quite easy. Simply include a package command followed by name of the package as the first statement in java source file.

Syntax:

```
package PackageName;
```

Here `PackageName` is the name of the package.

Example

```
package FirstPack;
public class A
{
public static void main(String args[])
{
    System.out.println("Welcome to First Package");
}
}
```

Output

```
run:
Welcome to First Package
```

Importing a package

Java has import statement that allows us to import an entire package or use only certain classes and interfaces defined in the package. The general form of import statement is:

```
import packagename.ClassName;   //  To import a certain class only
import packagename.*;           //   To import the whole package
```

For example

```
import java.util.Date; // imports only Date class
import java.io.*;      //  imports everything inside java.io package
```

How to access package from another package?

The **`import`** keyword is used to import built-in and user-defined packages into our java source file so that our class can refer to a class that is in another package by directly using its name. There are three ways to access the package from outside the package.

1. fully qualified name.
2. import package.*;
3. import package.classname

1. using fully qualified name

If we use fully qualified name, then only declared a class of this package will be accessible. Now there is no need to import. But we need to use fully qualified name every time when we are accessing the class or interface. It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class. But this is not a good practice.

Example

```
package pack1;
public class A
{
    public void display()
    {
        System.out.println("pack1 inside A class");
    }
}


package pack2;
public class B
{
    public static void main(String args[])
    {
        pack1.A obj=new pack1.A();
        obj.display();
    }
}
```

Output

```
run:
pack1 inside A class
```

2. Using packagename.classname (import only the class you want to use)

If we import package.classname then only declared class of this package will be accessible. Here import only that class that we want to use.

Example

```
package pack1;
public class A
{
    public void display()
    {
        System.out.println("pack1 inside A class");
    }
}


package pack2;
import pack1.A;
public class B
{
    public static void main(String args[])
```

```
    {
        A obj=new A();
        obj.display();
    }
}
```

Output

```
run:
pack1 inside A class
```

3. Import all the classes from the particular package

If we use packagename.* then all the classes and interfaces of this package will be accessible but not subpackages. The import keyword is used to make the classes and interfaces of another package accessible to the current package.

Example

```
package pack1;
public class A
{
    public void display()
    {
        System.out.println("pack1 inside A class");
    }
}
```

```
package pack2;
import pack1.*;
public class B
{
    public static void main(String args[])
    {
        A obj=new A();
        obj.display();
    }
}
```

Output

```
run:
pack1 inside A class
```

Sub package in Java

Package inside the package is known as subpackage. The packages that come lower in the naming hierarchy are called "**sub package**" of the corresponding package higher in the hierarchy i.e. the package that we are putting into another package is called "**sub package".** The naming convention defines how to create a unique package name so that packages that are widely used with unique namespaces. This allows packages to be easily managed. Suppose we have a file called "A.java". and want to store it in a sub package **"subpackage"**, which stays inside package **"pack1"**. The "A" class should look something like this:

Example

```
package pack1.subpackage;
public class A
{
    public void display()
    {
        System.out.println("pack1 inside A class");
    }
}
```

```
package pack2;
import pack1.subpackage.*;
public class B
{
    public static void main(String args[])
    {
        A obj=new A();
        obj.display();
    }
}
```

Output

```
run:
pack1 inside A class
```

Unit-5

Handling Errors/Exceptions

Exception

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

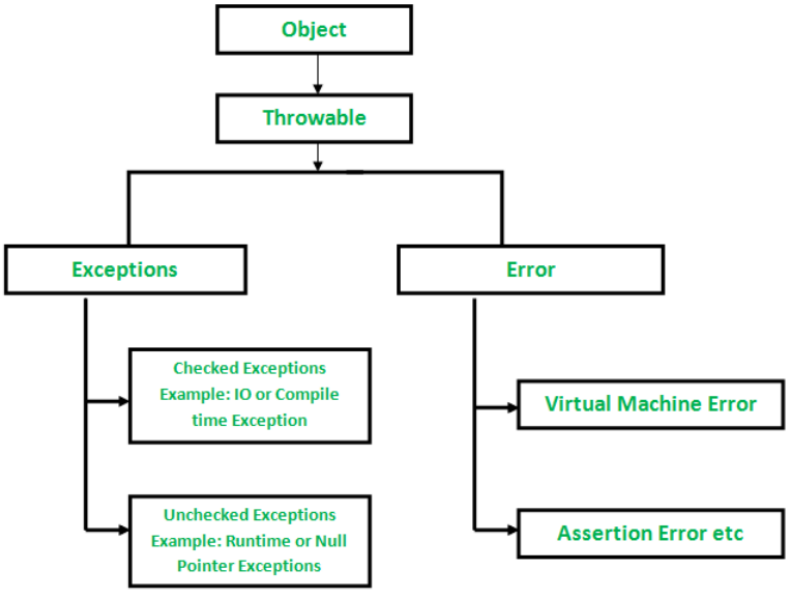
Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

For example, if we use FileReader class in our program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;
public class Checked
{
    public static void main(String args[])
    {
        File file=new File("D://a.txt");
        FileReader fr=new FileReader(file);
    }
}
```

Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

```
public class Unchecked
{
    public static void main(String args[])
    {
        int a[]={1,2,3,4};
```

```
        System.out.println(a[5]);
    }
}
```

Error

Error" is a subclass of the built-in class "Throwable". Errors are the critical conditions that occur due to the lack of the system resources, and it can not be handled by the code of the program. Errors can not be recovered by any means because they can not be created, thrown, caught or replied. Errors are caused due to the catastrophic failure which usually can not be handled by your program.

Errors are always of unchecked type, as compiler do not have any knowledge about its occurrence. Errors always occur in the run-time environment. The error can be explained with the help of an example, the program has an error of stack overflow, out of memory error, or a system crash error, this kind of error are due to the system.

The code is not responsible for such errors. The consequence of the occurrence of the error is that the program gets terminated abnormally.

Error Vs Exception in Java

1. Error occur only when system resources are deficient whereas, an exception is caused if a code has some problem.
2. An error can never be recovered whereas, an exception can be recovered by preparing the code to handle the exception.
3. An error can never be handled but, an exception can be handled by the code if the code throwing an exception is written inside a try and catch block.
4. If an error has occurred, the program will be terminated abnormally. On the other hand, If the exception occurs the program will throw an exception, and it is handled using the try and catch block.
5. Errors are of unchecked type i.e. error are not in the knowledge of compilers whereas, an exception is classified as checked and unchecked.
6. Errors are defined in java.lang.Error package whereas, an exception is defined java.lang.Exception.

Difference between Error and Exception

Errors	Exceptions
1. Recovering from Error is not possible.	**1.** We can recover from exceptions by either using try-catch block or throwing exceptions back to caller.
2. All errors in java are unchecked type.	**2.** Exceptions include both checked as well as unchecked type.
3. Errors are mostly caused by the environment in which program is running.	**3.** Program itself is responsible for causing exceptions.
4. Errors occur at runtime and not known to the compiler.	**4.** All exceptions occurs at runtime but checked exceptions are known to compiler while unchecked are not.
5. They are defined in java.lang.Error package.	**5.** They are defined in java.lang.Exception package
6. Examples : `java.lang.StackOverflowError`, `java.lang.OutOfMemoryError`	**6.** Examples : Checked Exceptions : `SQLException`, `IOException` `Unchecked Exceptions :` `ArrayIndexOutOfBoundException`, `NullPointerException`, `ArithmeticException.`

Advantages of Exception handling

1. ***Separating Error-Handling Code from "Regular" Code:-***
   Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere.
2. ***Propagating Errors Up the Call Stack:-***
   A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods.
3. ***Grouping and Differentiating Error Types:-***
   Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

```
try{
   //statements that may cause an exception
}
```

catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

```
try
{
     //statements that may cause an exception
}
catch (exception(type) e(object))
{
     //error handling code
}
```

Example 1

```
import java.util.Scanner;
public class HandlingException
{
    public static void main(String args[])
    {
         Scanner sc=new Scanner(System.in);
         int a,b;
         System.out.println("Enter two numbers ");
         a=sc.nextInt();
         b=sc.nextInt();
         try
         {
          System.out.println(a/b);
         }
         catch(ArithmeticException e)
         {
             System.out.println("We shouldnot divide a number by
Zero");
         }
    }
}
```

Output

```
Enter two numbers
3 0
We shouldnot divide a number by Zero
```

Example 2

```
public class HandlingException
{
    public static void main(String args[])
    {
        try
        {
            int a[]={1,4,3,4};
           System.out.println(a[5]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
    }
}
```

Output

```
run:
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4
```

Multiple Catch Blocks

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Example 1

```
public class MultipleCatchBlock
{
    public static void main(String args[])
    {
        try
        {
            int a[]=new int[5];
            a[2]=3/0;
            System.out.println(a[7]);
        }
        catch(ArithmeticException e)
        {
```

```
            System.out.println("Divide by Zero Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBounds Exception
occurs");
        }
    }
}
```

Output

```
run:
Divide by Zero Exception occurs
```

Example 2

```
public class MultipleCatchBlock
{
    public static void main(String args[])
    {
        try
        {
            int a[]=new int[5];
            a[2]=6/3;
            System.out.println(a[7]);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by Zero Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBounds Exception
occurs");
        }
    }
}
```

Output

```
run:
ArrayIndexOutOfBounds Exception occurs
```

Nested try catch block

The try block within a try block is known as nested try block in java.

Syntax

```
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
```

Example

```
public class NestedTryCatch
{
    public static void main(String args[])
    {
    try
    {
        try
        {
            System.out.println("going to divide");
            int b =39/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        try
        {
```

```
            int a[]=new int[5];
            a[5]=4;
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
  }
    catch(Exception e)
    {
        System.out.println("handeled");
    }
 }
}
```

Output

```
going to divide
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
```

throw Exception

The `throw` keyword in Java is used for explicitly throwing a single exception. This can be from within a method or any block of code. Both checked and unchecked exceptions can be thrown using the `throw` keyword.

When an exception is thrown using the `throw` keyword, the flow of execution of the program is stopped and the control is transferred to the nearest enclosing try-catch block that matches the type of exception thrown. If no such match is found, the default exception handler terminates the program.

Syntax

```
throw exception;
```

Example

```
import java.util.Scanner;
public class ThrowException
{
    void validate(int age)
    {
        if(age<16)
            throw new ArithmeticException("Not valid");
```

```
        else
            System.out.println("Could vote");
    }
    public static void main(String args[])
    {
        ThrowException obj=new ThrowException();
        Scanner sc=new Scanner(System.in);
        int a;
        System.out.println("Enter age ");
        a=sc.nextInt();
        obj.validate(a);
    }
}
```

Output

```
run:
Enter age
10
Exception in thread "main" java.lang.ArithmeticException: Not valid
```

throws Exception

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Example

```
import java.io.*;
class A
{
 void method()throws IOException
 {
  throw new IOException("device error");
 }
}
class ThrowsException
{
   public static void main(String args[])
   {
    try
    {
       A obj=new A();
       obj.method();
    }
    catch(Exception e)
    {
        System.out.println("Exception Handled");
```

```
    }
  }
}
```

Output

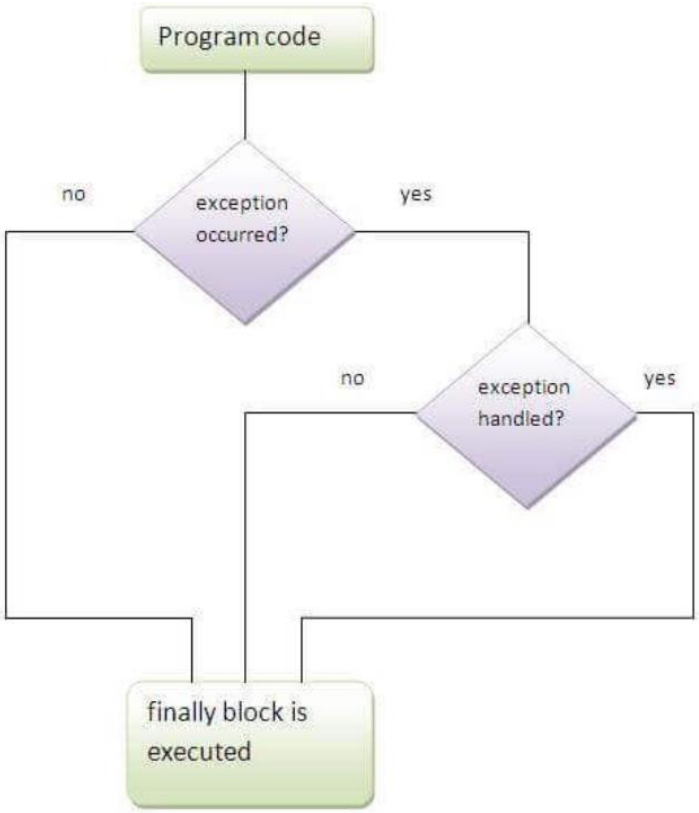
```
run:
Exception Handled
```

Difference between throw and throws

throw	throws
1. throw keyword is used to throw an exception explicitly.	1. throws keyword is used to declare one or more exceptions, separated by commas.
2. Only single exception is thrown by using throw.	2. Multiple exceptions can be thrown by using throws.
3. throw keyword is used within the method.	3. throws keyword is used with the method signature.
4. Syntax wise throw keyword is followed by the instance variable.	4. Syntax wise throws keyword is followed by exception class names.
5. Checked exception cannot be propagated using throw only.Unchecked exception can be propagated using throw.	5. For the propagation checked exception must use throws keyword followed by specific exception class name.

finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.



Example

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

      try
      {
         int a = 10;
         int b = 0;
         int result = a/b;
      }
      catch(Exception e)
      {
         System.out.println("Error: "+ e.getMessage());
      }
      finally
      {
         System.out.println("Finished.");
      }
   }
}
```

Output

```
run:
Error: / by zero
Finished.
```

Properties of finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. we should place those statements in this block that must be executed always.
2. finally, block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for exception handling, however if we place a finally block then it will always run after the execution of try block.
3. In normal case when there is no exception in try block then the finally block is executed after try block. However, if an exception occurs then the catch block is executed before finally block.
4. An exception in the finally block, behaves exactly like any other exception.
5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.

Unit-6

Handling Strings

Introduction

String is an **object** that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package. It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.

There are two ways to create String object:

1. **By string literal**

   Java String literal is created by using double quotes. For Example:

   ```
   String s="welcome";
   ```

   Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

2. **By new keyword**

   We can create a new string object by using **new** operator that allocates memory for the object. Each time we create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as **string constant pool** inside the heap memory.

   ```
   String s=new String("Welcome");
   ```

Java String Methods

There are a lot of built in java string methods, some of major string methods are listed below,

- String length()
- String compareTo()
- String concat()

- String IsEmpty()
- String Trim()
- String toLowerCase()
- String toUpper()

String length

The java String length() method tells the length of the string. It returns count of total number of characters present in the String. For example:

Example

```
public class StringLength
{
    public static void main(String args[])
    {
        String s="Object Oriented Programming in Java";
        System.out.println("Length of given string is
"+s.length());
    }
}
```

Output

```
run:
Length of given string is 35
```

String compares

String class provides equals() and equalsIgnoreCase() methods to compare two strings. These methods compare the value of string to check if two strings are equal or not. It returns true if two strings are equal and false if not.

Example

```
public class StringEquals
{
    public static void main(String args[])
    {
        String s1="Anjila";
        String s2="Anjila";
        String s3="Niruta";
        String s4="ANJILA";
        System.out.println(s1.equals(s2));
        System.out.println(s2.equals(s3));
        System.out.println(s1.equals(s4));
        System.out.println(s1.equalsIgnoreCase(s4));
    }}
```

Output

```
run:
true
false
false
true
```

String class implements comparable interface, which provides compareTo() and compareToIgnoreCase() methods and it compares two strings lexicographically. Both strings are converted into Unicode value for comparison and return an integer value which can be greater than, less than or equal to zero. If strings are equal then it returns zero or else if returns either greater or less than zero.

Example

```
public class StringCompareTo
{
    public static void main(String args[])
    {
        String s1="Anjila";
        String s2="Anjila";
        String s3="Niruta";
        String s4="ANJILA";
        System.out.println(s1.compareTo(s2));
        System.out.println(s2.compareTo(s3));
        System.out.println(s1.compareTo(s4));
        System.out.println(s1.compareToIgnoreCase(s4));
    }
}
```

Output

```
run:
0
-13
32
0
```

String concatenation

String concatenation is very basic operation in java. String can be concatenated by using "+" operator or by using concat() method. The java String concat() method combines a specific string at the end of another string and ultimately returns a combined string. It is like appending another string.

Example

```
public class StringConcat
{
    public static void main(String args[])
    {
        String s1="Hello";
        String s2="World";
        System.out.println("using + operator "+(s1+s2));
        System.out.println("using concat method "+
s1.concat(s2));
    }
}
```

Output

```
run:
using + operator HelloWorld
using concat method HelloWorld
```

String IsEmpty()

Java String IsEmpty() method checks whether a String is empty or not. This method returns true if the given string is empty, else it returns false. In other words we can say that this method returns true if the length of the string is 0.

Example

```
public class StringIsEmpty
{
    public static void main(String args[])
    {
        String s1="";
        String s2="JAVA";
        System.out.println(s1.isEmpty());
        System.out.println(s2.isEmpty());
    }
}
```

Output

```
run:
true
false
```

String Trim()

The java string trim() method removes the leading and trailing spaces. It checks the Unicode value of space character (“u0020”) before and after the string. If it exists, then removes the spaces and return the omitted string.

Example

```
public class StringTrim
{
    public static void main(String args[])
    {
        String s="  JAVA  ";
        System.out.println(s+" Programming Language");
        System.out.println(s.trim()+" Programming Language");
    }
}
```

Output

```
run:
  JAVA   Programming Language
JAVA Programming Language
```

String toLowerCase()

The java string toLowerCase() method converts all the characters of the String to lower case.

Example

```
public class StringLower
{
    public static void main(String args[])
    {
        String s="OOP in Java";
        System.out.println(s.toLowerCase());
    }
}
```

Output

```
run:
oop in java
```

String toUpperCase()

The java string toUpperCase() method converts all the characters of the string to upper case.

Example

```
public class StringUpper
{
    public static void main(String args[])
    {
        String s="OOP in Java";
        System.out.println(s.toUpperCase());
    }
}
```

Output

```
run:
OOP IN JAVA
```

Character Extraction

The process of extracting particular character of any symbol from given string is called character extracting. There are a lot of methods of for extracting character from given string some of major character extraction methods are listed below.

- charAt()
- getChars()
- getBytes()
- toCharArray()

String charAt()

The Java string `charAt(int index)` method returns the character at the specified index in a string. The index value that we pass in this method should be between 0 and (length of string-1).

Example

```
public class CharAtExample
{
    public static void main(String args[])
    {
        String s="Object Oriented Programming in Java";
        char c1=s.charAt(0);
        char c2=s.charAt(14);
        char c3=s.charAt(15);
        char c4=s.charAt(31);
```

```
            System.out.println("character at 0 index is "+c1);
            System.out.println("character at 14th index is "+c2);
            System.out.println("character at 15th index is "+c3);
            System.out.println("character at 31th index is "+c4);
        }
    }
```

Output

```
run:
character at 0 index is O
character at 14th index is d
character at 15th index is
character at 31th index is J
```

getChars()

If we need to extract more than one character at a time, we can use the getChars() method. It's general form is:

```
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
```

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd-1. The array that will receive the character is specified by target. The index within target at which the substring will be copied is passed to targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

Example

```
public class GetCharsExample
{
    public static void main(String args[])
    {
        String s1="Object Oriented Programming in Java";
        int start=16;
        int end=30;
        char s2[]=new char[end-start];
        s1.getChars(start, end, s2,0);
        System.out.println(s2);
    }
}
```

Output

```
run:
Programming in
```

getBytes()

`getBytes()` extract characters from String object and then convert the characters in a byte array. It has following syntax.

```
byte[] getBytes()
```

Example

```
public class GetBytes
{
    public static void main(String args[])
    {
        String s="OOP in Java";
        byte[] b=s.getBytes();
        for(byte i:b)
        {
            System.out.println(i);
        }
    }
}
```

Output

```
run:
79
79
80
32
105
110
32
74
97
118
97
```

toCharArray()

It is an alternative of getChars() method. toCharArray() convert all the characters in a String object into an array of characters. It is the best and easiest way to convert string to character array. It has following syntax.

```
char[] toCharArray()
```

Example

```
public class ToCharArray
{
    public static void main(String args[])
    {
        String s="OOP in Java";
        char ch[]=s.toCharArray();
        System.out.println(ch);
    }
}
```

Output

```
run:
OOP in Java
```

Searching Strings

Searching a string for finding the location of either a character or group of characters (substring) is a common task. We can search for a particular letter in a string using the indexOf() method of the String class. This method returns a position index of a word within the string if found. Otherwise it returns -1.

Example

```
public class SearchingStrings
{
    public static void main(String args[])
    {
        String s="OOP in Java";
        int index;
        System.out.println("Index of the letter J =
"+s.indexOf('J'));
    }
}
```

Output

```
run:
Index of the letter J = 7
```

Following example shows how we can search a word within a String object using indexOf() method which returns a position index of a word within the string if found. Otherwise it returns -1.

Example

```
public class SearchingStrings
{
    public static void main(String args[])
    {
        String s="BCA 3rd Semester";
        int id=s.indexOf("Semester");
        if(id==-1)
            System.out.println("Semester not found");
        else
            System.out.println(" Found at index " +id);
    }
}
```

Output

```
run:
 Found at index 8
```

String Buffer class in Java

StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	Creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	Creates a string buffer with the specified string.
StringBuffer(int capacity)	Creates an empty string buffer with the specified capacity on length.

Methods used in String Buffer class

Some of the most used methods in String Buffer class for updating given string in java are listed below:

- length() and capacity()
- append()
- insert()
- reverse()
- delete() and deleteCharAt()
- replace()

length() and capacity()

the length of a StringBuffer can be found by the length() method, while the total allocated capacity can be found by the capacity() method. The default capacity of the buffer is 16. If the number of character increases from its current capacity, then number of characters are added to their default value.

Example

```
public class LengthCapacity
{
    public static void main(String args[])
    {
        StringBuffer s=new StringBuffer("BCA 3rd Semester");
        int l=s.length();
        int c = s.capacity();
        System.out.println("Length of given string = "+l);
        System.out.println("capacity of given string = "+c);
    }
}
```

Output

```
run:
Length of given string = 16
capacity of given string = 32
```

append()

It is used to add text at the end of the existence text.

Example

```
public class AppendExample
{
    public static void main(String args[])
    {
        StringBuffer s=new StringBuffer("BCA 3rd Semester");
        System.out.println(s);
        s.append(" OOP in Java");
```

```
        System.out.println(s);
    }
}
```

Output

```
run:
BCA 3rd Semester
BCA 3rd Semester OOP in Java
```

insert()

It is used to insert text at the specified index position. Here index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

Example

```
public class AppendExample
{
    public static void main(String args[])
    {
        StringBuffer s=new StringBuffer("OOP in Java BCA 3rd
Semester");
        System.out.println(s);
        s.insert(12, "of ");
        System.out.println(s);
    }
}
```

Output

```
run:
OOP in Java BCA 3rd Semester
OOP in Java of BCA 3rd Semester
```

reverse()

It can reverse the characters within a StringBuffer object using reverse(). This method returns the reversed object on which it was called.

Example

```
public class ReverseExample
{
    public static void main(String args[])
    {
        StringBuffer s=new StringBuffer("OOP in Java");
```

```
        System.out.println(s);
        System.out.println(s.reverse());
    }
}
```

Output

```
run:
OOP in Java
avaJ ni POO
```

delete() and deleteCharAt()

It can delete characters within a StringBuffer by using the method delete() and deleteCharAt(). The delete() method deletes a sequence of characters from the invoking object. Here start index specifies the index of the first character to remove, and end index specifies an index one past the last character to remove. Thus, the substring deleted runs from start index to endindex-1. The resulting StringBuffer object is returned. The deleteCharAt() method deletes the character at the index specified by location. It returns the resulting StringBuffer object. These methods are:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deletCharAt(int loc)
```

Example

```
public class DeleteAndDeleteCharAtExample
{
    public static void main(String args[])
    {
        StringBuffer s=new StringBuffer("OOP in Java");
        System.out.println(s);
        s.delete(2,7);
        System.out.println(s);
        StringBuffer str= new StringBuffer("BCA 3rd Semester");
        str.deleteCharAt(2);
        System.out.println(str);
    }
}
```

Output

```
run:
OOP in Java
OOJava
BC 3rd Semester
```

replace()

The replace() method replaces the given string from the specified beginIndex and endIndex. It can replace one set of characters with another set inside a StringBuffer object by calling replace(). The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex-1 is replaced.

Example

```
public class ReplaceExample
{
    public static void main(String args[])
    {
        StringBuffer s= new StringBuffer("BCA 3rd Semester");
        System.out.println(s);
        s.replace(4, 7, "Third");
        System.out.println(s);
    }
}
```

Output

```
run:
BCA 3rd Semester
BCA Third Semester
```

delete()

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

Example

```
public class DeleteExample
{
    public static void main(String args[])
    {
        StringBuffer s= new StringBuffer("BCA 3rd Semester");
        System.out.println(s);
       s.delete(4, 7);
        System.out.println(s);
    }
}
```

Output

```
run:
BCA 3rd Semester
BCA  Semester
```

Unit-7

Threads

Introduction

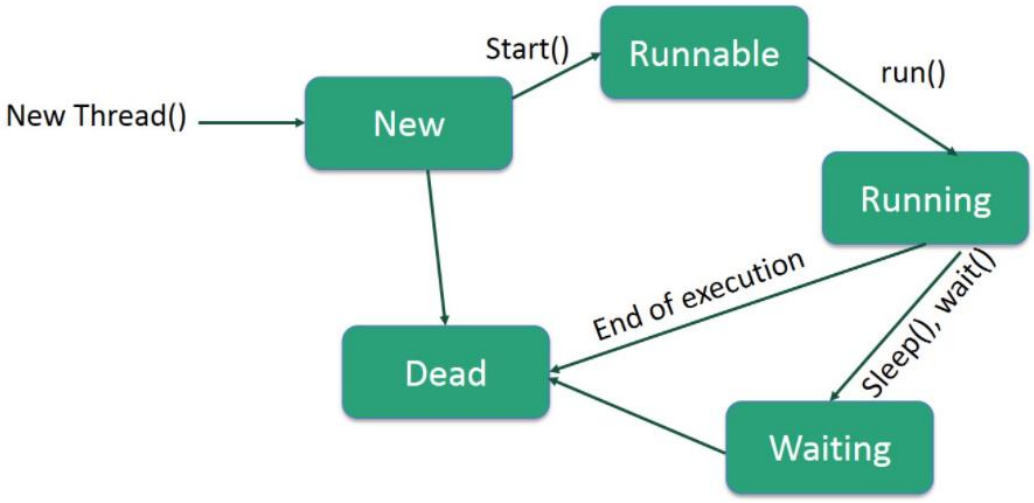
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle −

- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread methods

Following is the list of important methods available in the Thread class.

S.N.	Methods and description
1	**public void start()** starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

S.N.	Methods and description
1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds.
3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

How to create a thread?

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

1. **Creating Thread by extending Thread class**
   Create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1:

You will need to override **run( )** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −

public void run( )

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

void start( );

Example

```
class MultiThread extends Thread
{
    public void run()
    {
        for(int i=65; i<=90; ++i)
        System.out.print((char)i+" ");
    }
}
class ThreadExample
{
    public static void main(String args[])
    {
        MultiThread obj=new MultiThread();
        obj.start();
        for(int i=1; i<=20; ++i)
        System.out.print(i+" ");
    }
}
```

Output

run:

A B C 1 2 D 3 E F G 4 H I 5 6 7 8 9 J K L M N 10 O 11 12 P Q R S T U V W X Y Z 13 14 15 16 17 18 19 20

2. Creating Thread by implementing Runnable interface.

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

Step 1

As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method −

```
public void run( )
```

Step 2

As a second step, you will instantiate a **Thread** object using the following constructor −

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

```
void start();
```

Example

```
class MultiThread implements Runnable
{
    public void run()
    {
        for(int i=65; i<=90; ++i)
        System.out.print((char)i+" ");
    }
}
class ThreadExample
{
    public static void main(String []arg)
    {
        MultiThread obj=new MultiThread();
        Thread t=new Thread(obj);
        t.start();
        for(int i=1; i<=20; ++i)
        System.out.print(i+" ");
    }
}
```

Output

run:

A B C 1 D 2 E 3 F 4 G H 5 I 6 7 8 9 J K L M 10 N O P Q R S T U 11 V 12 W X 13 Y Z 14 15 16 17 18 19 20

Synchronization

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

Example

Without Synchronization

```
class MultiplicationTable
{

    void printTable(int n)
    {
       for(int i=1;i<=10;i++)
       {
         System.out.println(n+"x"+i+"="+n*i);
         try
         {
          Thread.sleep(400);
         }
         catch(Exception e)
         {
             System.out.println(e);
         }
        }
    }
}

class Thread1 extends Thread
{
```

```
    MultiplicationTable t;
    Thread1(MultiplicationTable t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}
class Thread2 extends Thread
{
    MultiplicationTable t;
    Thread2(MultiplicationTable t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
}

class WithoutSynchronization
{
    public static void main(String args[])
    {
        MultiplicationTable obj = new MultiplicationTable();
        Thread1 t1=new Thread1(obj);
        Thread2 t2=new Thread2(obj);
        t1.start();
        t2.start();
    }
 }
```

Output

```
run:
5x1=5
100x1=100
5x2=10
100x2=200
5x3=15
100x3=300
5x4=20
100x4=400
5x5=25
100x5=500
5x6=30
100x6=600
5x7=35
100x7=700
5x8=40
100x8=800
5x9=45
100x9=900
5x10=50
100x10=1000
```

Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization

Example

```
class MultiplicationTable
{
    synchronized void printTable(int n)
    {
       for(int i=1;i<=10;i++)
       {
         System.out.println(n+"x"+i+"="+n*i);
         try
         {
          Thread.sleep(400);
         }
         catch(Exception e)
         {
             System.out.println(e);
```

```
            }
        }
    }
}

class Thread1 extends Thread
{
    MultiplicationTable t;
    Thread1(MultiplicationTable t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}
class Thread2 extends Thread
{
    MultiplicationTable t;
    Thread2(MultiplicationTable t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
}

class Synchronization
{
    public static void main(String args[])
    {
        MultiplicationTable obj = new MultiplicationTable();
        Thread1 t1=new Thread1(obj);
        Thread2 t2=new Thread2(obj);
        t1.start();
        t2.start();
    }
 }
```

Output

```
run:
5x1=5
5x2=10
5x3=15
5x4=20
5x5=25
5x6=30
5x7=35
5x8=40
5x9=45
5x10=50
100x1=100
100x2=200
100x3=300
100x4=400
100x5=500
100x6=600
100x7=700
100x8=800
100x9=900
100x10=1000
```

Using Synchronized Block

If we have to synchronize access to an object of a class or we only want a part of a method to be synchronized to an object then we can use synchronized block for it.

Example

```
class MultiplicationTable
{
    void printTable(int n)
    {
       for(int i=1;i<=10;i++)
       {
         System.out.println(n+"x"+i+"="+n*i);
         try
         {
          Thread.sleep(400);
         }
         catch(Exception e)
```

```
            {
                System.out.println(e);
            }
        }
    }
}

class Thread1 extends Thread
{
    MultiplicationTable t;
    Thread1(MultiplicationTable t)
    {
        this.t=t;
    }
    public void run()
    {
        synchronized(t)
        {
        t.printTable(5);
        }
    }
}
class Thread2 extends Thread
{
    MultiplicationTable t;
    Thread2(MultiplicationTable t)
    {
        this.t=t;
    }
    public void run()
    {
        synchronized(t)
        {
        t.printTable(100);
        }
    }
}

class Synchronization
{
    public static void main(String args[])
    {
        MultiplicationTable obj = new MultiplicationTable();
        Thread1 t1=new Thread1(obj);
        Thread2 t2=new Thread2(obj);
        t1.start();
        t2.start();
    }
 }
```

Output

```
run:
5x1=5
5x2=10
5x3=15
5x4=20
5x5=25
5x6=30
5x7=35
5x8=40
5x9=45
5x10=50
100x1=100
100x2=200
100x3=300
100x4=400
100x5=500
100x6=600
100x7=700
100x8=800
100x9=900
100x10=1000
```

Difference between synchronized keyword and synchronized block

When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution. Synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

Inter-Thread Communication

Java provide benefits of avoiding thread pooling using inter-thread communication.
The `wait()`, `notify()`, and `notifyAll()` methods of Object class are used for this purpose. These method are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context

- `wait()` tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- `notify()` wakes up a thread that called wait() on same object.
- `notifyAll()` wakes up all the thread that called wait() on same object.

Difference between wait() and sleep()

wait()	sleep()
1. called from synchronised block	1. no such requirement
2. monitor is released	2. monitor is not released
3. gets awake when notify() or notifyAll() method is called.	3. does not get awake when notify() or notifyAll() method is called
4. not a static method	4. static method
5. wait() is generaly used on condition	5. sleep() method is simply used to put your thread on sleep.

Example

```
class Customer
{
    int amount=10000;
    synchronized void withdraw(int amount)
    {
        System.out.println("going to withdraw...");
        if(this.amount<amount)
        {
            System.out.println("Less balance; waiting for
deposit...");
            try
            {
                wait();
            }
            catch(Exception e)
            {
            }
        }
        this.amount=this.amount-amount;
        System.out.println("withdraw completed...");
    }
    synchronized void deposit(int amount)
    {
        System.out.println("going to deposit...");
        this.amount=this.amount+amount;
        System.out.println("deposit completed... ");
        notify();
    }
}
class InterProcessCommunication
```

```
{
    public static void main(String args[])
    {
        final Customer c=new Customer();
        new Thread()
        {
            public void run()
            {
                c.withdraw(15000);
            }
        }.start();
        new Thread()
        {
            public void run()
            {
                c.deposit(10000);
            }
        }.start();
    }
}
```

Output

```
run:
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed...
```

Thread deadlock

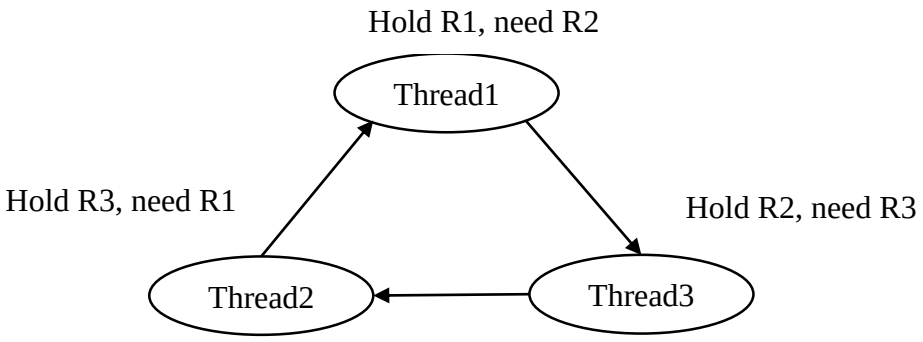


Figure: Deadlock condition

Deadlock is a situation of complete lock, when no thread can complete its execution because lack of resources. In the above figure, Thread1 is holding a resource R1, and need another resource R2

to finish execution, but R2 is locked by Thread2, which needs R3, which in turn is locked by Thread3. Hence none of them can finish and are stuck in a deadlock.

Example

```
public class ThreadDeadlock {

    String str1 = "Java";
    String str2 = "Python";

    Thread t1 = new Thread("My Thread 1"){
        public void run(){
            while(true){
                synchronized(str1){
                    try{
                        Thread.sleep(100);
                    }
                    catch(InterruptedException e){
                        e.printStackTrace();
                    }
                    synchronized(str2){
                        System.out.println(str1 + str2);
                    }
                }
            }
        }
    };

    Thread t2 = new Thread("My Thread 2"){
        public void run(){
            while(true){
                synchronized(str2){
                    synchronized(str1){
                        System.out.println(str2 + str1);
                    }
                }
            }
        }
    };

    public static void main(String a[]){
        ThreadDeadlock obj = new ThreadDeadlock();
        obj.t1.start();
        obj.t2.start();
    }
}
```

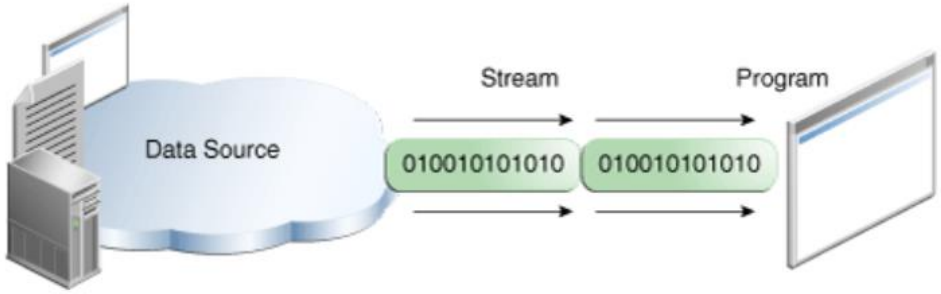
Unit-8

I/O and Streams

Introduction

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays. Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways. No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data.

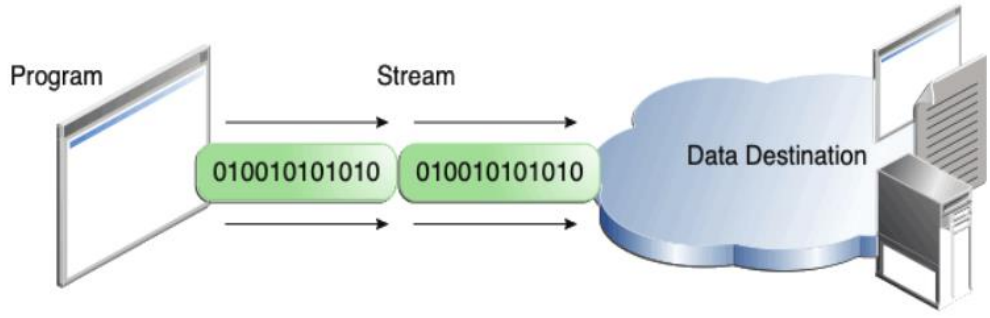
A program uses an *input stream* to read data from a source, one item at a time.



Reading information into a program.

Reading information into a program.

A program uses an output stream to write data to a destination, one item at time:



Writing information from a program.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Byte Stream

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

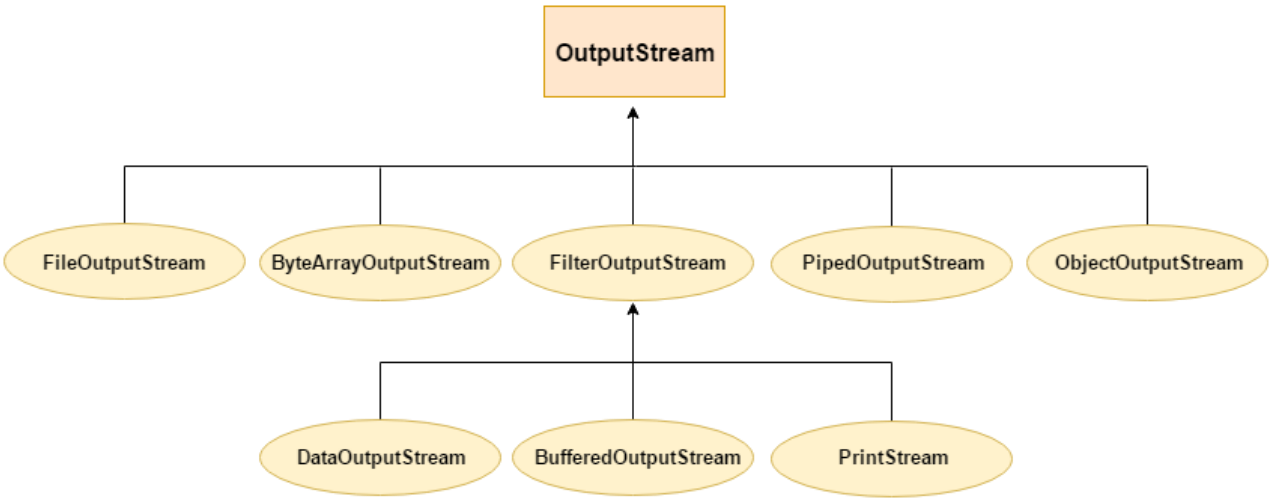
OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



FileOutStream

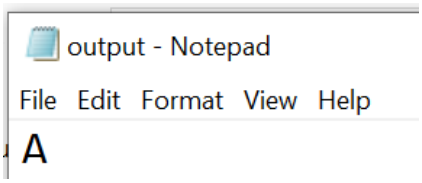
Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

Example

```
import java.io.*;
public class WritingtoFile
{
   public static void main(String args[]) throws IOException
   {
      FileOutputStream out = null;
      try {
         out = new FileOutputStream("D:\\output.txt");
         int a=65;
         out.write(a);
      }
      finally
      {
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Output



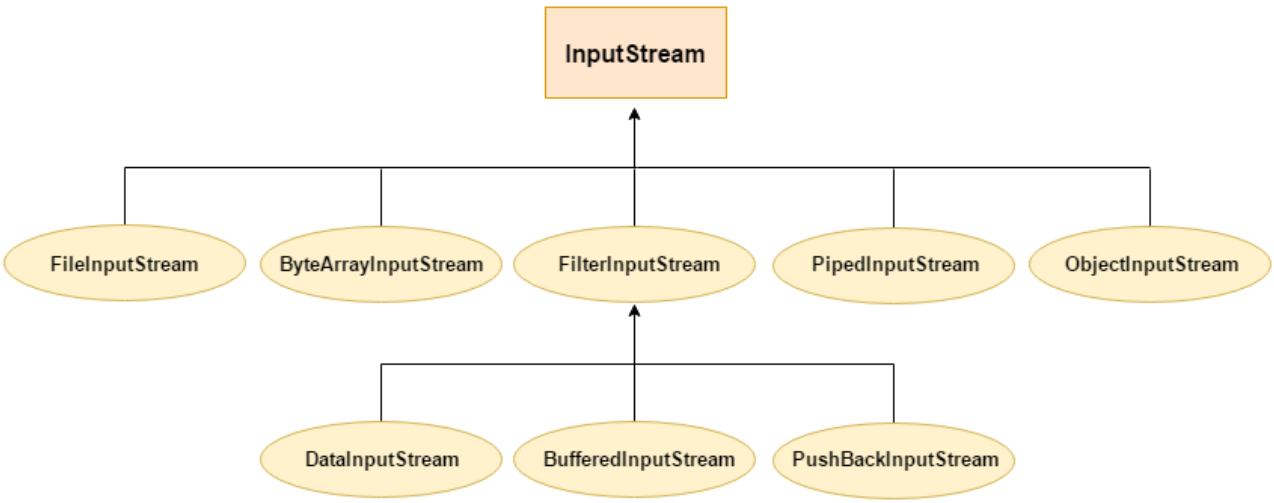
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



FileInputStream

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Example

```
import java.io.*;
public class ReadingFromFile
{
   public static void main(String args[]) throws IOException
```

```
        {
            FileInputStream out = null;
            try {
                out = new FileInputStream("D:\\output.txt");
                System.out.println(out.read());
            }
            finally
            {
                if (out != null) {
                    out.close();
                }
            }
        }
    }
```

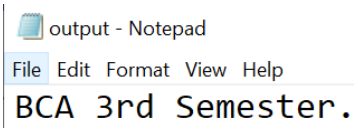
Output

```
run:
65
```

Write a program to copy the content of one file to another file

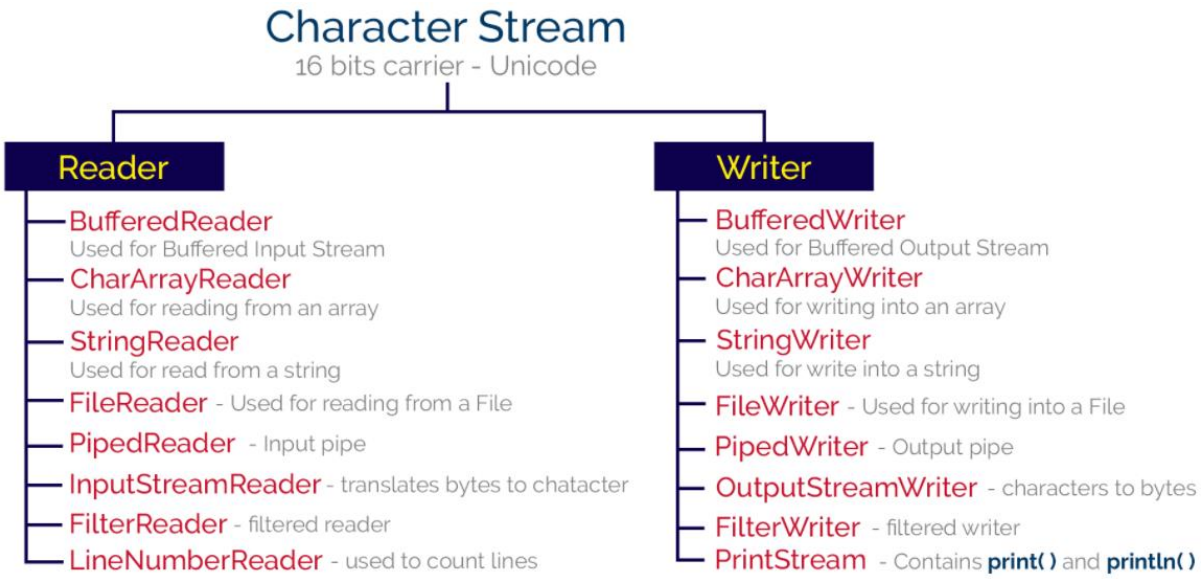
```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("D:\\input.txt");
            out = new FileOutputStream("D:\\output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Output



Java Character Stream Class

Character streams is used to perform input and output for 16-bit Unicode. Though there are many classes related to character streams but the most frequently used classes are FileReader and FileWriter.

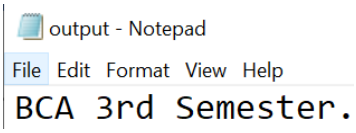


Example

```
import java.io.*;
public class CopyFile {
   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("D:\\input.txt");
         out = new FileWriter("D:\\output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
```

```
            }
            if (out != null) {
               out.close();
            }
         }
      }
   }
```

Output



Reading Console Input

We use the object of BufferedReaderclass and Scanner class to take inputs from the keyword.

Scanner Class

Scanner is a class in java.util package used for obtaining the input of primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if we want an input method for scenarios where time is a constraint like in competitive programming.

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort().
- To read strings, we use nextLine().
- To read a single character, we use next.charAt(0).

Example

```
import java.util.Scanner;
public class ScannerExample
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.print("Name: ");
        String name=sc.nextLine();
        System.out.print("Gender: ");
        char gender=sc.next().charAt(0);
```

```
            System.out.print("Age: ");
            int age=sc.nextInt();
            System.out.print("Mobile No: ");
            long mobileno=sc.nextLong();
            System.out.print("GPA: ");
            double gpa=sc.nextDouble();
            System.out.println("\nName: "+name);
            System.out.println("Gender: "+gender);
            System.out.println("Age: "+age);
            System.out.println("Mobileno: "+mobileno);
            System.out.println("GPA: "+gpa);
        }
    }
```

Output

```
Name: Priya
Gender: F
Age: 24
Mobileno: 9845123412
GPA: 3.9
```

BufferedReader class

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes. In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

Example

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class BufferedReaderExample {
    public static void main(String[] args)
        throws IOException
    {
        BufferedReader reader = new BufferedReader
        (
        new InputStreamReader(System.in));
        System.out.print("Name: ");
        String name = reader.readLine();
```

```
            System.out.println("Name: "+name);
        }
    }
```

Output

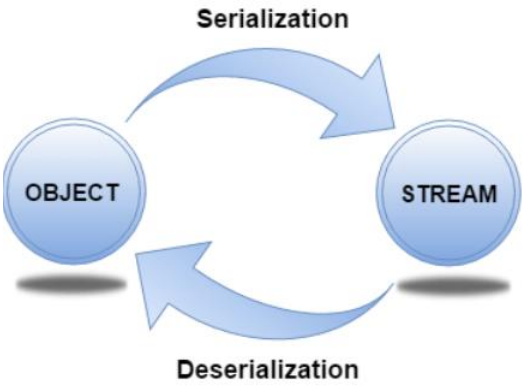
```
run:
Name: Priya Gautam
Name: Priya Gautam
```

Serialization and Deserialization

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface.

The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

The ObjectInputStream class contains **readObject()** method for deserializing an object.



Example

```
import java.io.*;
public class SerializeExample
{
    public static void main(String args[])
    {
        Student s=new Student();
        s.rollno=5;
        s.name="Bimala";
        try
        {
            FileOutputStream                                fout=new
FileOutputStream("d:\\Student.ser");
```

```
            ObjectOutputStream                              out=new
ObjectOutputStream(fout);
            out.writeObject(s);
            out.close();
            fout.close();
            System.out.println("Serialized  data  is  saved  in
employee.ser");
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
        try
        {
            FileInputStream                                 fin=new
FileInputStream("D:\\Student.ser");
            ObjectInputStream in=new ObjectInputStream(fin);
            s=(Student)in.readObject();
            System.out.println("Rollno = "+s.rollno);
            System.out.println("Name = "+s.name);
            in.close();
            fin.close();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Output

```
run:
Serialized data is saved in employee.ser
Rollno = 5
Name = Bimala
```

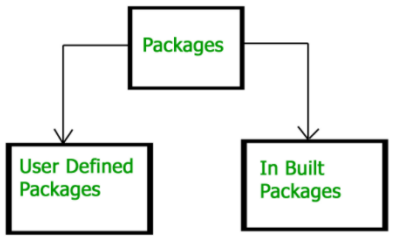
Unit-9

Understanding Core Packages

Introduction

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

Types of package



Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) **java.io:** Contains classed for supporting input / output operations.
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) **java.applet:** Contains classes for creating Applets.
5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
6) **java.net:** Contain classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

Example

```
package myPackage;
public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

```
import myPackage.MyClass;
public class PrintName
{
    public static void main(String args[])
    {
        String name="Birendra Multiple Campus";
        MyClass obj=new MyClass();
        obj.getNames(name);
    }
}
```

Output

```
run:
Birendra Multiple Campus
```

java.lang package

Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

- `Object`, the ultimate superclass of all classes in Java
- `Thread`, the class that controls each thread in a multithreaded program
- `Throwable`, the superclass of all error and exception classes in Java
- Classes that encapsulate the primitive data types in Java
- Classes for accessing system resources and other low-level entities
- `Math`, a class that provides standard mathematical methods
- `String`, the class that is used to represent strings

Because the classes in the `java.lang` package are so essential, the `java.lang` package is implicitly imported by every Java source file. In other words, you can refer to all of the classes and interfaces in `java.lang` using their simple names.

Class Summary

Class	Description
Boolean	The Boolean class wraps a value of the primitive type `boolean` in an object.
Byte	The `Byte` class wraps a value of primitive type `byte` in an object.
Character	The `Character` class wraps a value of the primitive type `char` in an object.
Character.Subset	Instances of this class represent particular subsets of the Unicode character set.
Character.UnicodeBlock	A family of character subsets representing the character blocks in the Unicode specification.
Class<T>	Instances of the class `Class` represent classes and interfaces in a running Java application.

ClassLoader	A class loader is an object that is responsible for loading classes.
ClassValue<T>	Lazily associate a computed value with (potentially) every type.
Compiler	The `Compiler` class is provided to support Java-to-native-code compilers and related services.
Double	The `Double` class wraps a value of the primitive type `double` in an object.
Enum<E extends Enum<E>>	This is the common base class of all Java language enumeration types.
Float	The `Float` class wraps a value of primitive type `float` in an object.
InheritableThreadLocal<T>	This class extends `ThreadLocal` to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
Integer	The `Integer` class wraps a value of the primitive type `int` in an object.
Long	The `Long` class wraps a value of the primitive type `long` in an object.
Math	The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
Number	The abstract class `Number` is the superclass of classes `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`.
Object	Class `Object` is the root of the class hierarchy.
Package	`Package` objects contain version information about the implementation and specification of a Java package.
Process	The **`ProcessBuilder.start()`** and **`Runtime.exec`** methods create a native process and return an instance of a subclass of `Process` that can be used to control the process and obtain information about it.
ProcessBuilder	This class is used to create operating system processes.
ProcessBuilder.Redirect	Represents a source of subprocess input or a destination of subprocess output.
Runtime	Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running.
RuntimePermission	This class is for runtime permissions.
SecurityManager	The security manager is a class that allows applications to implement a security policy.
Short	The Short class wraps a value of primitive type short in an object.
StackTraceElement	An element in a stack trace, as returned by Throwable.getStackTrace().
StrictMath	The class StrictMath contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
String	The String class represents character strings.
StringBuffer	A thread-safe, mutable sequence of characters.
StringBuilder	A mutable sequence of characters.
System	The System class contains several useful class fields and methods.
Thread	A thread is a thread of execution in a program.
ThreadGroup	A thread group represents a set of threads.
ThreadLocal<T>	This class provides thread-local variables.
Throwable	The Throwable class is the superclass of all errors and exceptions in the Java language.

Void	The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.

Java.lang.Math class

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Field

Following are the fields for **java.lang.Math** class −

- **static double E** − This is the double value that is closer than any other to e, the base of the natural logarithms.
- **static double PI** − This is the double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Class Methods

1	static double abs(double a) This method returns the absolute value of a double value.
2	static float abs(float a) This method returns the absolute value of a float value.
3	static int abs(int a) This method returns the absolute value of an int value.
4	static long abs(long a)

	This method returns the absolute value of a long value.
5	static double acos(double a) This method returns the arc cosine of a value; the returned angle is in the range 0.0 through pi.
6	static double asin(double a) This method returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2.
7	static double atan(double a) This method returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2.
8	static double atan2(double y, double x) This method returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).
9	static double cbrt(double a) This method returns the cube root of a double value.
10	static double ceil(double a) This method returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
11	static double copySign(double magnitude, double sign) This method returns the first floating-point argument with the sign of the second floating-point argument.
12	static float copySign(float magnitude, float sign) This method returns the first floating-point argument with the sign of the second floating-point argument.
13	static double cos(double a) This method returns the trigonometric cosine of an angle.
14	static double cosh(double x)

	This method returns the hyperbolic cosine of a double value.
15	static double exp(double a) This method returns Euler's number e raised to the power of a double value.
16	static double expm1(double x) This method returns e^x -1.
17	static double floor(double a) This method returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
18	static int getExponent(double d) This method returns the unbiased exponent used in the representation of a double.
19	static int getExponent(float f) This method returns the unbiased exponent used in the representation of a float.
20	static double hypot(double x, double y) This method returns sqrt(x^2 +y^2) without intermediate overflow or underflow.
21	static double IEEEremainder(double f1, double f2) This method computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
22	static double log(double a) This method returns the natural logarithm (base e) of a double value.
23	static double log10(double a) This method returns the base 10 logarithm of a double value.
24	static double log1p(double x) This method returns the natural logarithm of the sum of the argument and 1.
25	static double max(double a, double b)

	This method returns the greater of two double values.
26	static float max(float a, float b) This method returns the greater of two float values.
27	static int max(int a, int b) This method returns the greater of two int values.
28	static long max(long a, long b) This method returns the greater of two long values.
29	static double min(double a, double b) This method returns the smaller of two double values.
30	static float min(float a, float b) This method returns the smaller of two float values.
31	static int min(int a, int b) This method returns the smaller of two int values.
32	static long min(long a, long b) This method returns the smaller of two long values.
33	static double nextAfter(double start, double direction) This method returns the floating-point number adjacent to the first argument in the direction of the second argument.
34	static float nextAfter(float start, double direction) This method returns the floating-point number adjacent to the first argument in the direction of the second argument.
35	static double nextUp(double d) This method returns the floating-point value adjacent to d in the direction of positive infinity.
36	static float nextUp(float f)

	This method returns the floating-point value adjacent to f in the direction of positive infinity.
37	static double pow(double a, double b) This method returns the value of the first argument raised to the power of the second argument.
38	static double random() This method returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
39	static double rint(double a) This method returns the double value that is closest in value to the argument and is equal to a mathematical integer.
40	static long round(double a) This method returns the closest long to the argument.
41	static int round(float a) This method returns the closest int to the argument.
42	static double scalb(double d, int scaleFactor) This method returns $d \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set.
43	static float scalb(float f, int scaleFactor) This method return $f \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the float value set.
44	static double signum(double d) This method returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
45	static float signum(float f) This method returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero.

46	static double sin(double a) This method returns the hyperbolic sine of a double value.
47	static double sinh(double x) This method Returns the hyperbolic sine of a double value.
48	static double sqrt(double a) This method returns the correctly rounded positive square root of a double value.
49	static double tan(double a) This method returns the trigonometric tangent of an angle.r
50	static double tanh(double x) This method returns the hyperbolic tangent of a double value.
51	static double toDegrees(double angrad) This method converts an angle measured in radians to an approximately equivalent angle measured in degrees.
52	static double toRadians(double angdeg) This method converts an angle measured in degrees to an approximately equivalent angle measured in radians.
53	static double ulp(double d) This method returns the size of an ulp of the argument.
54	static double ulp(float f) This method returns the size of an ulp of the argument.

Example

```
public class JavaMathExample
{
    public static void main(String[] args)
    {
        double x = 28;
        double y = 4;
```

```
        // return the maximum of two numbers
        System.out.println("Maximum number of x and y is: "
+Math.max(x, y));

        // return the square root of y
        System.out.println("Square root of y is: " +
Math.sqrt(y));

        //returns 28 power of 4 i.e. 28*28*28*28
        System.out.println("Power of x and y is: " +
Math.pow(x, y));

        // return the logarithm of given value
        System.out.println("Logarithm of x is: " +
Math.log(x));
        System.out.println("Logarithm of y is: " +
Math.log(y));

        // return the logarithm of given value when base is 10
        System.out.println("log10 of x is: " + Math.log10(x));
        System.out.println("log10 of y is: " + Math.log10(y));

        // return the log of x + 1
        System.out.println("log1p of x is: " +Math.log1p(x));

        // return a power of 2
        System.out.println("exp of a is: " +Math.exp(x));

        // return (a power of 2)-1
        System.out.println("expm1 of a is: " +Math.expm1(x));
    }
}
```

Output

```
run:
Maximum number of x and y is: 28.0
Square root of y is: 2.0
Power of x and y is: 614656.0
Logarithm of x is: 3.332204510175204
Logarithm of y is: 1.3862943611198906
log10 of x is: 1.4471580313422192
log10 of y is: 0.6020599913279624
log1p of x is: 3.367295829986474
exp of a is: 1.446257064291475E12
expm1 of a is: 1.446257064290475E12
```

Wrapper Classes

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

Example

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

```
        {
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);/* converting int into
    Integer explicitly */
        Integer j=a; /*autoboxing, now compiler will write
    Integer.valueOf(a) internally */
        System.out.println(a+" "+i+" "+j);
        }
    }
```

Output

```
run:
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Example

```
class Unboxing
{
    public static void main(String[] args)
    {
    //Converting Integer to int
    Integer a=new Integer(3);
    int i=a.intValue(); /*converting Integer to int explicitly
*/
    int j=a; /*unboxing, now compiler will write a.intValue()
internally */
    System.out.println(a+" "+i+" "+j);
    }
}
```

Output

```
run:
20 20 20
```

Implementation

Example

```
class WrappingUnwrapping
{
    public static void main(String args[])
    {
        //  byte data type
        byte a = 1;
```

```
        // wrapping around Byte object
        Byte byteobj = new Byte(a);

        // int data type
        int b = 10;

        //wrapping around Integer object
        Integer intobj = new Integer(b);

        // float data type
        float c = 18.6f;

        // wrapping around Float object
        Float floatobj = new Float(c);

        // double data type
        double d = 250.5;

        // Wrapping around Double object
        Double doubleobj = new Double(d);

        // char data type
        char e='a';

        // wrapping around Character object
        Character charobj=e;

        //  printing the values from objects
        System.out.println("Values of Wrapper objects
(printing as objects)");
        System.out.println("Byte object byteobj:  " +
byteobj);
        System.out.println("Integer object intobj:  " +
intobj);
        System.out.println("Float object floatobj:  " +
floatobj);
        System.out.println("Double object doubleobj:  " +
doubleobj);
        System.out.println("Character object charobj:  " +
charobj);

        // objects to data types (retrieving data types from
objects)
        // unwrapping objects to primitive data types
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
```

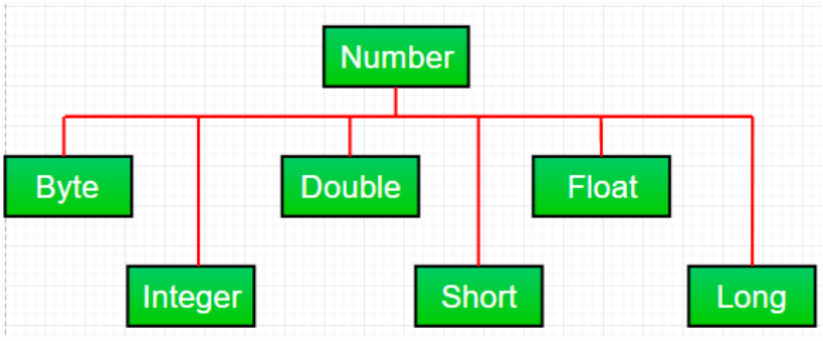
```
            char cv = charobj;

            // printing the values from data types
            System.out.println("Unwrapped values (printing as data
    types)");
            System.out.println("byte value, bv: " + bv);
            System.out.println("int value, iv: " + iv);
            System.out.println("float value, fv: " + fv);
            System.out.println("double value, dv: " + dv);
            System.out.println("char value, cv: " + cv);
        }
    }
```

Output

```
Values of Wrapper objects (printing as objects)
Byte object byteobj:  1
Integer object intobj:  10
Float object floatobj:  18.6
Double object doubleobj:  250.5
Character object charobj:  a
Unwrapped values (printing as data types)
byte value, bv: 1
int value, iv: 10
float value, fv: 18.6
double value, dv: 250.5
char value, cv: a
```

Java.lang.Number Class

Most of the time, while working with numbers in java, we use primitive data types. But, Java also provides various numeric wrapper sub classes under the abstract class Number present in *java.lang* package. There are mainly **six** sub-classes under Number class.These sub-classes define some useful methods which are used frequently while dealing with numbers.



These classes “wrap” the primitive data type in a corresponding object. Often, the wrapping is done by the compiler. If you use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class for you. Similarly, if you use a Number object when a primitive is expected, the compiler unboxes the object for you. This is also called Autoboxing and Unboxing.

Why to use a Number class object over primitive data?

- Constants defined by the number class, such as MIN_VALUE and MAX_VALUE, that provide the upper and lower bounds of the data type are very much useful.
- Number class object can be used as an argument of a method that expects an object (often used when manipulating collections of numbers).
- Class methods can be used for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

1. **xxx xxxValue()** : Here xxx represent primitive number data types (byte, short, int, long, float, double). This method is used to convert the value of this Number object to the primitive data type specified.
   Example

```
public class Test
{
    public static void main(String[] args)
    {
        // Creating a Double Class object with value
        // "6.9685"
        Double d = new Double("6.9685");

        // Converting this Double(Number) object to
        // different primitive data types
        byte b = d.byteValue();
        short s = d.shortValue();
        int i = d.intValue();
        long l = d.longValue();
        float f = d.floatValue();
        double d1 = d.doubleValue();

        System.out.println("value of d after converting it to
byte : " + b);
        System.out.println("value of d after converting it to
short : " + s);
        System.out.println("value of d after converting it to
int : " + i);
        System.out.println("value of d after converting it to
long : " + l);
        System.out.println("value of d after converting it to
float : " + f);
```

```
        System.out.println("value of d after converting it to
double : " + d1);
    }
}
```

Output

```
run:
value of d after converting it to byte : 6
value of d after converting it to short : 6
value of d after converting it to int : 6
value of d after converting it to long : 6
value of d after converting it to float : 6.9685
value of d after converting it to double : 6.9685
```

2. **int compareTo(NumberSubClass referenceName)**

This method is used to compare this Number object to the argument specified. However, two different types cannot be compared, so both the argument and the Number object that invoke the method should be of the same type. The referenceName could be a Byte, Double, Integer, Float, Long, or Short.

```
Returns :
the value 0 if the Number is equal to the argument.
the value 1 if the Number is less than the argument.
the value -1 if the Number is greater than the argument.
```

Example

```
public class Test
{
    public static void main(String[] args)
    {
        // creating an Integer Class object with value "10"
        Integer i = new Integer("10");

        // comparing value of i
        System.out.println(i.compareTo(7));
        System.out.println(i.compareTo(11));
        System.out.println(i.compareTo(10));
    }
}
```

Output

```
run:
1
-1
0
```

3. **boolean equals(Object obj)**

This method determine whether this Number object is equal to the argument.

Returns :
The method returns true if the argument is not null and is an object of the same type and with the same numeric value, otherwise false.

Example

```
public class Test
{
    public static void main(String[] args)
    {
        // creating a Short Class object with value "15"
        Short s = new Short("15");

        // creating a Short Class object with value "10"
        Short x = 10;

        // creating an Integer Class object with value "15"
        Integer y = 15;

        // creating another Short Class object with value
        // "15"
        Short z = 15;

        //comparing s with other objects
        System.out.println(s.equals(x));
        System.out.println(s.equals(y));
        System.out.println(s.equals(z));
    }
}
```

Output

```
run:
false
false
true
```

4. **int parseInt(String s, int radix)**

This method is used to get the primitive data type of a String. Radix is used to return decimal(10), octal(8), or hexadecimal(16) etc representation as output.

Example

```
public class Test
{
    public static void main(String[] args)
    {
        // parsing different strings
        int z = Integer.parseInt("15",8);
        int a = Integer.parseInt("A", 16);
```

```
        long l = Long.parseLong("2158611234",10);
        System.out.println(z);
        System.out.println(a);
        System.out.println(l);
    }
}
```

Output

```
run:
13
10
2158611234
```

5. **String toString()**

There are two variants of toString() method. They are used to get String representation of a number. The other variants of these methods are

`Integer.toBinaryString(int i)`
`Integer.toHexString(int i)`
`Integer.toOctalString(int i)`

which will return binary, hexa-decimal, octal string representation of specified integer(i) respectively.

Example

```
public class Test
{
    public static void main(String[] args)
    {
        // demonstrating toString() method
        Integer x = 12;

        System.out.println(x.toString());

        // demonstrating toString(int i) method
        System.out.println(Integer.toString(12));

        System.out.println(Integer.toBinaryString(152));
        System.out.println(Integer.toHexString(152));
        System.out.println(Integer.toOctalString(152));
    }
}
```

Output

```
run:
12
12
10011000
98
230
```

6. **Integer valueOf()**

There are three variants of valueOf() method. All these three methods return an Integer object holding the value of a primitive integer.

Example

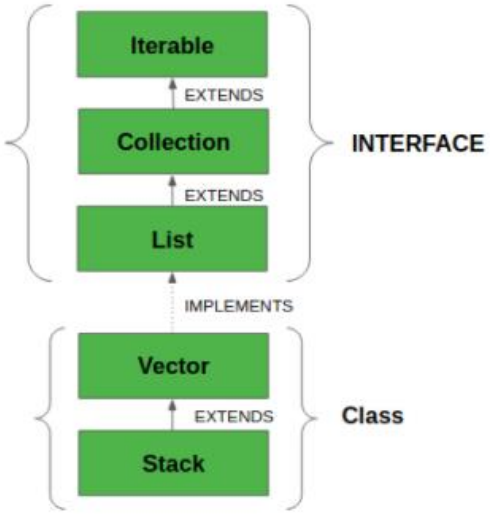
```
public class Test
{
    public static void main(String[] args)
    {
        // demonstrating valueOf(int i) method
        System.out.println("Demonstrating valueOf(int i)
method");
        Integer i =Integer.valueOf(50);
        Double d = Double.valueOf(9.36);
        System.out.println(i);
        System.out.println(d);

        // demonstrating valueOf(String s) method
        System.out.println("Demonstrating valueOf(String s)
method");
        Integer n = Integer.valueOf("333");
        Integer m = Integer.valueOf("-255");
        System.out.println(n);
        System.out.println(m);

        // demonstrating valueOf(String s,int radix) method
        System.out.println("Demonstrating (String s,int radix)
method");
        Integer y = Integer.valueOf("333",8);
        Integer x = Integer.valueOf("-255",16);
        Long l = Long.valueOf("51688245",16);
        System.out.println(y);
        System.out.println(x);
        System.out.println(l);
    }
}
```

Output

```
run:
Demonstrating valueOf(int i) method
50
9.36
Demonstrating valueOf(String s) method
333
-255
Demonstrating (String s,int radix) method
219
-597
1365803589
```

java.util.Stack class

Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector. The below diagram shows the **hierarchy of the Stack class**:



The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Methods in Stack class

Method	Description
`boolean empty()`	Tests if this stack is empty.
`Object peek()`	Looks at the object at the top of this stack without removing it from the stack.
`Object pop()`	Removes the object at the top of this stack and returns that object as the value of this function.
`Object push(Object element)`	Pushes an item onto the top of this stack.
`int search(Object element)`	Returns the 1-based position where an object is on this stack.

Example

```
import java.io.*;
import java.util.*;

class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop Operation:");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top: " +
element);
    }
```

```
    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int
element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position:
" + pos);
    }


    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

Output

```
run:
Pop Operation:
4
3
2
1
0
Element on stack top: 4
Element is found at position: 3
Element not found
```

java.util.Random class

Random class is used to generate pseudo-random numbers in java. An instance of this class is thread-safe. The instance of this class is however cryptographically insecure. This class provides various method calls to generate different random data types such as float, double, int.

Constructors:

- **Random()**: Creates a new random number generator
- **Random(long seed)**: Creates a new random number generator using a single long seed

Example

```
import java.util.Random;
public class Test
{
    public static void main(String[] args)
    {
        Random random = new Random();
        System.out.println(random.nextInt(10));
        System.out.println(random.nextBoolean());
        System.out.println(random.nextDouble());
        System.out.println(random.nextFloat());
        System.out.println(random.nextGaussian());
  }
}
```

Output

```
run:
7
false
0.8978113615371615
0.18495381
0.600991105724803
```

Unit-10

Holding Collection of Data

Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using the object property *length*. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Array can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array. In case of primitive data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.

One-Dimensional Arrays :

```
type var_name[];
```

OR

```
type[] var_name;
```

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc. or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.

Example

```
// both are valid declarations
int intArray[];
or int[] intArray;

byte byteArray[];
short shortsArray[];
boolean booleanArray[];
long longArray[];
float floatArray[];
double doubleArray[];
char charArray[];
```

```
// an array of references to objects of
// the class MyClass (a class created by
// user)
myClass myClassArray[];

Object[]  ao,        // array of Object
Collection[] ca;  // array of Collection
                     // of unknown type
```

Instantiating an Array in Java

When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array like this:The general form of *new* as it applies to one-dimensional arrays appears as follows:
var-name = new type [size];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of array variable that is linked to the array. That is, to use *new* to allocate an array, you must specify the type and number of elements to allocate.

Example

```
int intArray[];    //declaring array

intArray = new int[20];  // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```

Array literal

In a situation, where the size of the array and variables of array are already known, array literals can be used.

```
 int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

 // Declaring array literal
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java

Accessing Java Array Elements

Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

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

Multidimensional array

Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other array. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension. Examples:

```
int[][] intArray = new int[10][20]; //a 2D array or matrix
int[][][] intArray = new int[10][20][10]; //a 3D array
```

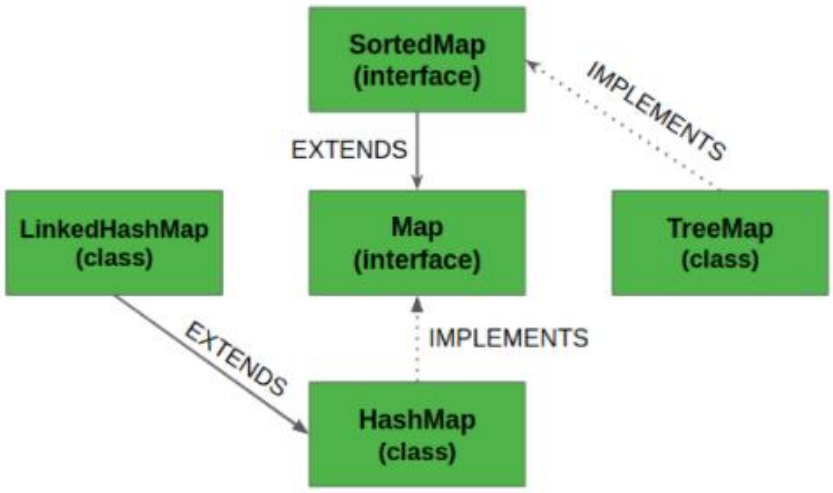
Example

```
class multiDimensional
{
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // 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();
        }
    }
}
```

Map interface

The Map interface present in java.util package represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types.
Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some examples are:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).



MAP Hierarchy in Java

Characteristics of a Map Interface

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap and LinkedHashMap, but some do not like the TreeMap.
2. The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable order, while HashMap does not.
3. There are two interfaces for implementing Map in java. They are, Map and SortedMap, and three classes: HashMap, TreeMap and LinkedHashMap.

Methods in Map Interface

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some examples are:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).
-

Methods	Description
public put(Object key, Object value)	This method inserts an entry in the map
public void putAll(Map map)	This method inserts the specified map in this map
public Object remove(Object key)	It is used to delete an entry for the specified key
public Set keySet()	It returns the Set view containing all the keys

public Set entrySet()	It returns the Set view containing all the keys and values
void clear()	It is used to reset the map
public void putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified
public Object get(Object key)	It returns the value for the specified key
public boolean containsKey(Object key)	It is used to search the specified key from this map

Example

```
import java.util.*;
class MapExample{
 public static void main(String args[]){
  Map<Integer,String> map=new HashMap<Integer,String>();
  map.put(100,"Ashika");
  map.put(101,"Sarmila");
  map.put(102,"Pratima");
  //Elements can traverse in any order
  for(Map.Entry m:map.entrySet())
      {
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Output

```
run:
100 Ashika
101 Sarmila
102 Pratima
```

HashMap

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap methods

Method	Description
void clear()	It is used to remove all of the mappings from this map.

boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
int size()	This method returns the number of entries in the map.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.

Example

```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new
HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Subal");  //Put elements in Map
   map.put(2,"Sanvi");
   map.put(3,"Sonu");
   map.put(4,"Laxmi");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
```

```
}
}
```

Output

```
run:
Iterating Hashmap...
1 Subal
2 Sanvi
3 Sonu
4 Laxmi
```

List Interface

The List interface provides a way to store the ordered collection. It is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements.

- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- In addition to the methods defined by **Collection**, List defines some of its own, which are summarized in the following table.
- Several of the list methods will throw an UnsupportedOperationException if the collection cannot be modified, and a ClassCastException is generated when one object is incompatible with another.

List interface methods

1	**void add(int index, Object obj)** Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	**boolean addAll(int index, Collection c)** Inserts all elements of **c** into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.

3	**Object get(int index)** Returns the object stored at the specified index within the invoking collection.
4	**int indexOf(Object obj)** Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
5	**int lastIndexOf(Object obj)** Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
6	**ListIterator listIterator( )** Returns an iterator to the start of the invoking list.
7	**ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index.
8	**Object remove(int index)** Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
9	**Object set(int index, Object obj)** Assigns obj to the location specified by index within the invoking list.
10	**List subList(int start, int end)** Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Example

```
import java.util.*;
public class ListInterfaceExample
{
    public static void main(String args[])
    {
```

```
        List<Integer>l1=new ArrayList<Integer>();
        l1.add(0,5);
        l1.add(1,7);
        System.out.println(l1);
        List<Integer>l2=new ArrayList<Integer>();
        l2.add(6);
        l2.add(9);
        l2.add(12);
        l1.addAll(1,l2);
        System.out.println(l1);
        l1.remove(1);
        System.out.println(l1);
        l1.set(0, 11);
        System.out.println(l1);
    }
}
```

Output

```
run:
[5, 7]
[5, 6, 9, 12, 7]
[5, 9, 12, 7]
[11, 9, 12, 7]
```

Search

List provides methods to search element and returns its numeric position. Following two methods are supported by list for this operation:

- **int indexOf(Object o)**: This method returns first occurrence of given element or -1 if element is not present in list.
- **int lastIndexOf(Object o)**: This method returns the last occurrence of given element or -1 if element is not present in list.

Example

```
import java.util.*;
public class ListInterfaceExample
{
    public static void main(String args[])
    {
        List<String>l=new ArrayList<String>();
        l.add("Mahima");
        l.add("Nikita");
        l.add("Niru");
        System.out.println("first index of Mahima:
"+l.indexOf("Mahima"));
```

```
        System.out.println("last index of Mahima:
"+l.lastIndexOf("Mahima"));
        System.out.println("first index of Nabin:
"+l.indexOf("Nabin"));
    }
}
```

Output

```
run:
first index of Mahima: 0
last index of Mahima: 0
first index of Nabin: -1
```

Iteration

ListIterator(extends Iterator) is used to iterate over list element. List iterator is bidirectional iterator.

Example

```
import java.util.*;
public class ListInterfaceExample
{
    public static void main(String args[])
    {
        List<String>names=new ArrayList<String>();
        names.add("Mahima");
        names.add("Nikita");
        names.add("Niru");
        ListIterator litr=names.listIterator();
        System.out.println("Traversing list in forward direction");
        while(litr.hasNext())
        {
            System.out.println(litr.next());
        }
        System.out.println("Traversing list in backward direction");
        while(litr.hasPrevious())
        {
            System.out.println(litr.previous());
        }
    }
}
```

Output

```
run:
Traversing list in forward direction
Mahima
Nikita
Niru
Traversing list in backward direction
Niru
Nikita
Mahima
```

Range-view

List Interface provides a method to get the List view of the portion of given List between two indices. Following is the method supported by List for range view operation.

List subList(int fromIndex, int toIndex): This method returns List view of specified list between fromindex(inclusive) and toindex(exclusive).

Example

```
import java.util.*;
public class SubListExample
{
    public static void main(String args[])
    {
        List<String>names=new ArrayList<String>();
        names.add("C");
        names.add("C++");
        names.add("Java");
        names.add("PHP");
        names.add("DotNet");
        List<String>range=new ArrayList<String>();
        range=names.subList(1, 3);
        System.out.println(range);
    }
}
```

Output

```
run:
[C++, Java]
```

Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

Set also adds a stronger contract on the behaviour of the equals and hash Code operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

Methods of Set Interface

1	**add( )** Adds an object to the collection.
2	**clear( )** Removes all objects from the collection.
3	**contains( )** Returns true if a specified object is an element within the collection.
4	**isEmpty( )** Returns true if the collection has no elements.
5	**iterator( )** Returns an Iterator object for the collection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the collection.
7	**size( )** Returns the number of elements in the collection.

Example

```
import java.util.*;
public class SetExample
{
    public static void main(String args[])
    {
```

```
            Set<String> s = new HashSet<String>();
            // Adding elements to the Set
            // using add() method
            s.add("Gita");
            s.add("Sheela");
            s.add("Nabina");
            s.add("Shailu");
            s.add("Sheela");
            System.out.println(s);
        }
    }
```

Output

```
run:
[Sheela, Nabina, Gita, Shailu]
```

Collection Classes

ArrayList is a part of **collection framework** and is present in java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by the size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.

Constructors in the ArrayList

ArrayList(): This constructor is used to build an empty array list.

ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from the collection c.

ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified.

Example

```
import java.util.*;
public class ArrayListExample
{
    public static void main(String args[])
    {
        ArrayList<String>s=new ArrayList<>();
        s.add("Devendra");
        s.add("Bindu");
        s.add("Hari");
```

```
            s.add("Binayak");
            s.add("Soba Raj");
            ListIterator litr=s.listIterator();
            while(litr.hasNext())
                System.out.println(litr.next());
        }
    }
```

Output

```
run:
Devendra
Bindu
Hari
Binayak
Soba Raj
```

Linked List

Linked List is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

Constructors in the LinkedList

LinkedList(): This constructor is used to create an empty linked list.

LinkedList(Collection C): This constructor is used to create an ordered list which contains all the elements of a specified collection, as returned by the collection's iterator.

Example

```
import java.util.*;
public class LinkedListExample
{
    public static void main(String args[])
    {
        LinkedList<String>s=new LinkedList<>();
        s.add("D");
        s.add("C");
        s.addLast("G");
        s.addFirst("A");
        s.add("E");
        s.add(1, "B");
        System.out.println(s);
```

```
            boolean status=s.contains("G");
            if(status)
                System.out.println("List Contains Element 'E' ");
            else
                System.out.println("List Doesnot contains Element
    'E' ");
            s.remove(1);
            System.out.println(s);
            s.remove("D");
            System.out.println(s);
            s.removeLast();
            System.out.println(s);
            s.removeFirst();
            System.out.println(s);
        }
    }
```

Output

```
run:
[A, B, D, C, G, E]
List Contains Element 'E'
[A, D, C, G, E]
[A, C, G, E]
[A, C, G]
[C, G]
```

Hash Set

The **HashSet** class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets.

- Implements Set Interface.
- The underlying data structure for HashSet is Hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements **Serializable** and **Cloneable** interfaces.

Example

```
import java.util.*;
public class HashSetExample
```

```
{
    public static void main(String args[])
    {
        HashSet<String>s=new HashSet<>();
        s.add("Devendra");
        s.add("Bindu");
        s.add("Hari");
        s.add("Binayak");
        s.add("Soba Raj");
        s.add("Binayak");  //adding duplicate elements
        System.out.println(s);
        s.remove("Soba Raj");
        System.out.println(s);
        System.out.println("\nIterating over list: ");
        Iterator<String> itr=s.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }
}
```

Output

```
[Soba Raj, Hari, Devendra, Bindu, Binayak]
[Hari, Devendra, Bindu, Binayak]

Iterating over list:
Hari
Devendra
Bindu
Binayak
BUILD SUCCESSFUL (total time: 0 seconds)
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.

- Java TreeSet class maintains ascending order.

Example

```
import java.util.*;
public class TreeSetExample
{
    public static void main(String args[])
    {
        TreeSet<String>s=new TreeSet<>();
        s.add("Devendra");
        s.add("Bindu");
        s.add("Hari");
        s.add("Binayak");
        s.add("Soba Raj");
        s.add("Binayak");  //adding duplicate elements
        System.out.println(s);
        s.remove("Soba Raj");
        System.out.println(s);
        System.out.println("\nIterating over list: ");
        Iterator<String> itr=s.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }
}
```

Output

```
run:
[Binayak, Bindu, Devendra, Hari, Soba Raj]
[Binayak, Bindu, Devendra, Hari]

Iterating over list:
Binayak
Bindu
Devendra
```

Unit-11

Java Applications

Java AWT

Java AWT (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.

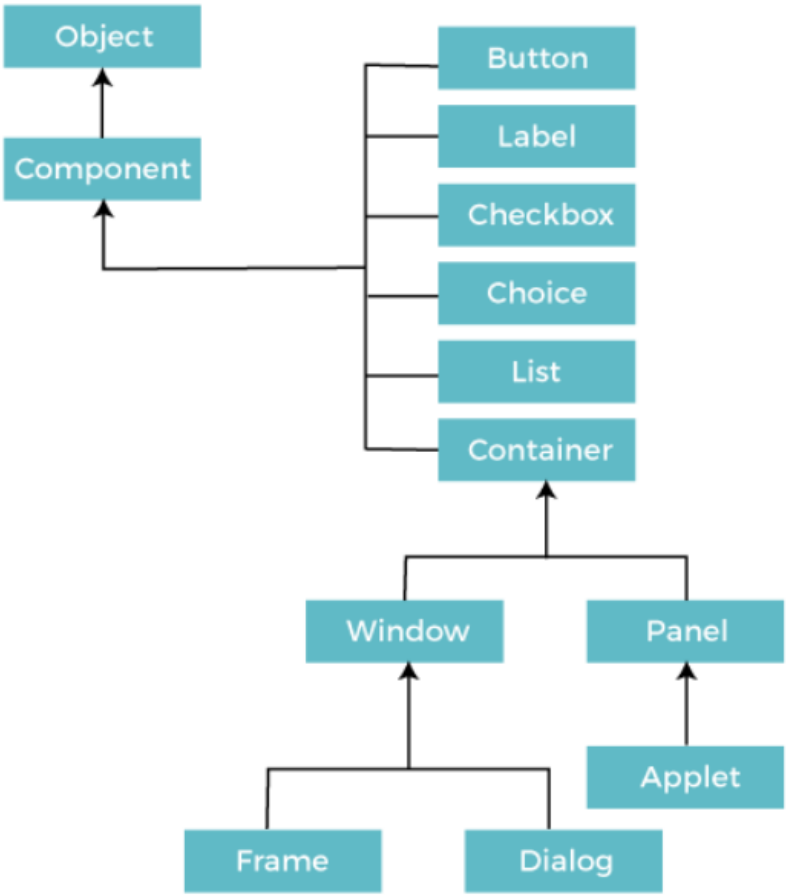
Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Components

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

Useful Methods of Component Class

Method	Description
public void add(Component c)	Inserts a component on this component.
public void setSize(int width,int height)	Sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	Defines the layout manager for the component.
public void setVisible(boolean status)	Changes the visibility of the component, by default false.

Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as **Frame, Dialog** and **Panel**. It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

Types of containers:

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

Java Swing

Java Swing is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

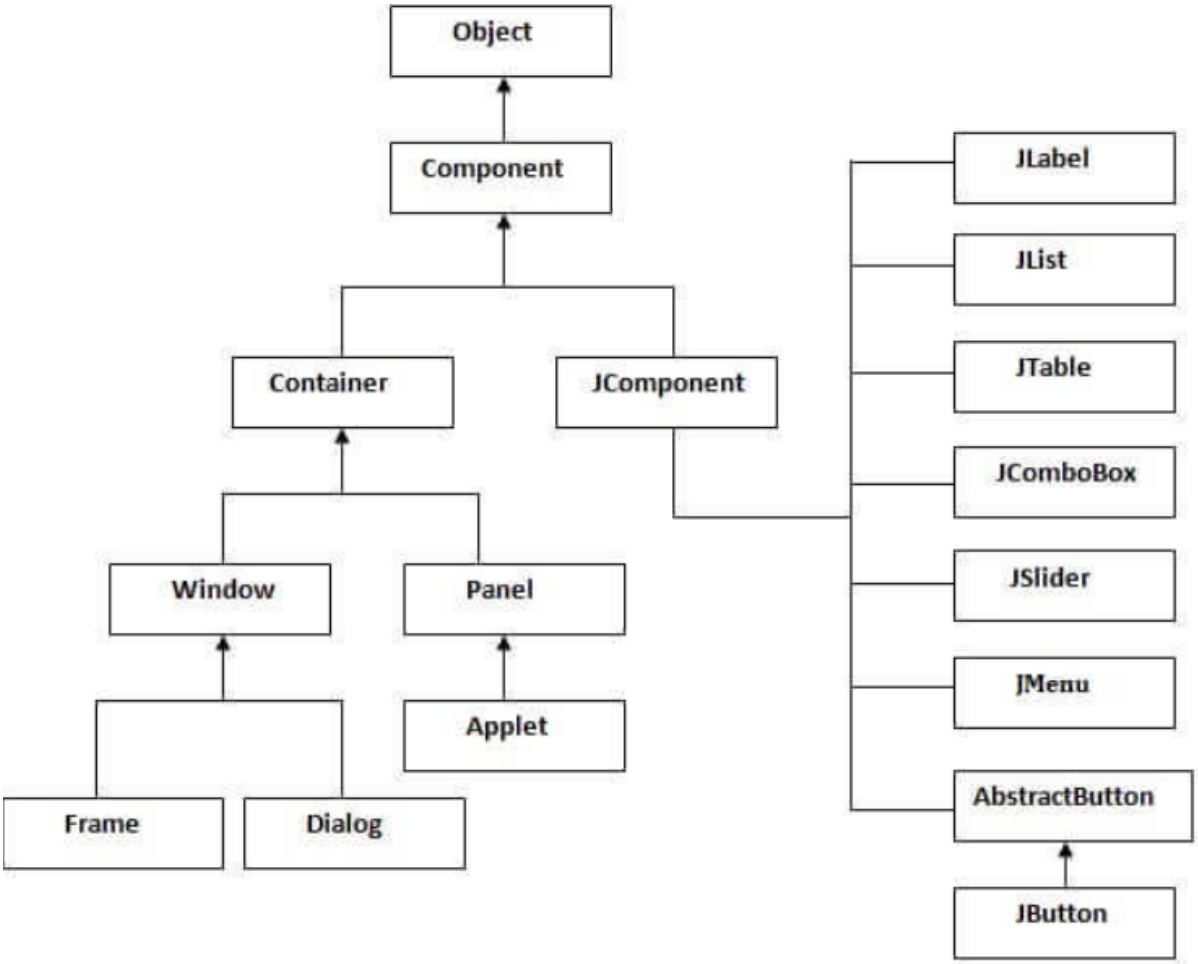
The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing

Java AWT	Java Swing
1. AWT components are **platform-dependent**	1. Java swing components are **platform-independent**
2. AWT components are **heavyweight**.	2. Swing components are **lightweight**.
3. AWT **doesn't support pluggable look and feel**.	3. Swing **supports pluggable look and feel**.
4. AWT provides **less components** than Swing.	4. Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5. AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	5. Swing **follows MVC**.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



JFrame

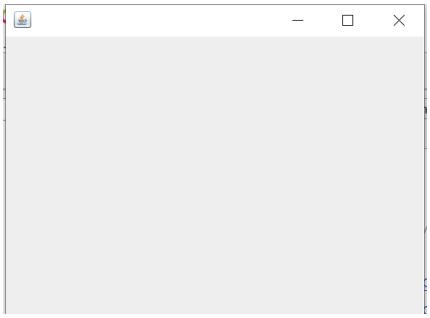
The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

Frame by inheritance

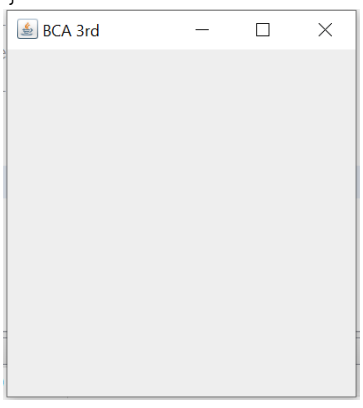
```
import javax.swing.*;
public class FrameByInheritance extends JFrame //inheriting JFrame
{
JFrame f;
FrameByInheritance()
{
setSize(400,500);
setLayout(null);
```

```
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public static void main(String[] args)
{
new FrameByInheritance();
}
}
```



Frame by Association

```
import javax.swing.*;
public class FrameByObject
{
public static void main(String[] args)
{
JFrame f=new JFrame("BCA 3rd");//creating instance of JFrame
f.setSize(500,300);//500 width and 300 height
f.setLayout(null);//using no layout managers
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);//making the frame visible
}
}
```



JPanel

JPanel, a part of Java Swing package, is a container that can store a group of components. The main task of JPanel is to organize components, various layouts can be set in JPanel which provide

better organisation of components, however it does not have a title bar.

Constructor of JPanel are :

1. **JPanel()** : creates a new panel with flow layout
2. **JPanel(LayoutManager l)** : creates a new JPanel with specified layoutManager
3. **JPanel(boolean isDoubleBuffered)** : creates a new JPanel with a specified buffering strategy
4. **JPanel(LayoutManager l, boolean isDoubleBuffered)** : creates a new JPanel with specified layoutManager and a specified buffering strategy

Commonly used functions :

1. **add(Component c)** : adds component to a specified container
2. **setLayout(LayoutManager l)** : sets the layout of the container to specified layout manager
3. **updateUI()** : resets the UI property with a value from the current look and feel.
4. **setUI(PanelUI ui)** : sets the look and feel object that renders this component.
5. **getUI()** : returns the look and feel object that renders this component.
6. **paramString()** : returns a string representation of this JPanel.
7. **getUIClassID()** : returns the name of the Look and feel class that renders this component.
8. **getAccessibleContext()** : gets the AccessibleContext associated with this JPanel.

Example

```
import java.awt.*;
import javax.swing.*;
class JPanelExample extends JFrame{
     static JFrame f;
     static JButton b,b1,b2;
        public static void main(String[] args)
        {
        f=new JFrame("panel");
        b=new JButton("Bibas");
        b1=new JButton("Asha");
        b2=new JButton("Rojana");
        JPanel p=new JPanel();
        p.add(b);
        p.add(b1);
        p.add(b2);
        p.setBounds(40,80,300,300);
        p.setBackground(Color.red);
        f.add(p);
        f.setSize(400,500);
        f.setLayout(null);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
        }
    }
```

Output

JLabel

JLabel is a class of java Swing . JLabel is used to display a short string or an image icon. JLabel can display text, image or both . JLabel is only a display of text or image and it cannot get focus . JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centered but the user can change the alignment of label.

Constructor of the class are :

1. **JLabel() :** creates a blank label with no text or image in it.
2. **JLabel(String s) :** creates a new label with the string specified.
3. **JLabel(Icon i) :** creates a new label with a image on it.
4. **JLabel(String s, Icon i, int align) :** creates a new label with a string, an image and a specified horizontal alignment

Commonly used methods of the class are :

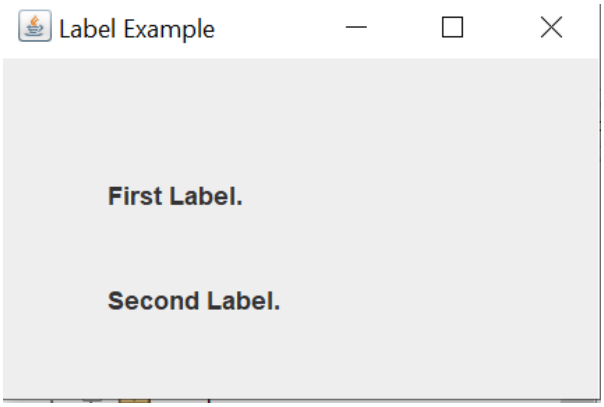
1. **getIcon() :** returns the image that the label displays
2. **setIcon(Icon i) :** sets the icon that the label will display to image i
3. **getText() :** returns the text that the label will display
4. **setText(String s) :** sets the text that the label will display to string s

Example

```
import javax.swing.*;
class JLabelExample
{
public static void main(String args[])
    {
    JFrame f= new JFrame("Label Example");
    JLabel l1,l2;
    l1=new JLabel("First Label.");
    l1.setBounds(50,50, 100,30);
    l2=new JLabel("Second Label.");
    l2.setBounds(50,100, 100,30);
    f.add(l1); f.add(l2);
    f.setSize(300,300);
    f.setLayout(null);
```

```
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
}
}
```

Output



JTextField

TextField is a part of javax.swing package. The class JTextField is a component that allows editing of a single line of text. JTextField inherits the JTextComponent class and uses the interface SwingConstants. The constructor of the class are :

1. **JTextField()** : constructor that creates a new TextField
2. **JTextField(int columns)** : constructor that creates a new empty TextField with specified number of columns.
3. **JTextField(String text)** : constructor that creates a new empty text field initialized with the given string.
4. **JTextField(String text, int columns)** : constructor that creates a new empty textField with the given string and a specified number of columns .
5. **JTextField(Document doc, String text, int columns)** : constructor that creates a textfield that uses the given text storage model and the given number of columns.

Methods of the JTextField are:

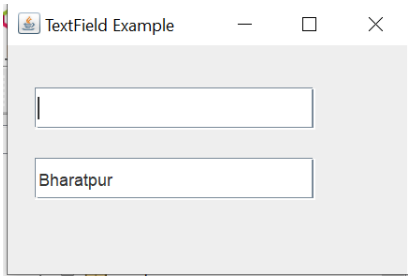
1. **setColumns(int n)** :set the number of columns of the text field.
2. **setFont(Font f)** : set the font of text displayed in text field.
3. **addActionListener(ActionListener l)** : set an ActionListener to the text field.
4. **int getColumns()** :get the number of columns in the textfield.

Example

```
import javax.swing.*;
class JTextFieldExample
{
```

```
public static void main(String args[])
    {
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField();
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("Bharatpur");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
    }
    }
```

Output



JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

Constructor of the class are

Constructor	Description
JButton()	It creates a button with no text and icon.
JButton(String s)	It creates a button with the specified text.
JButton(Icon i)	It creates a button with the specified icon object.

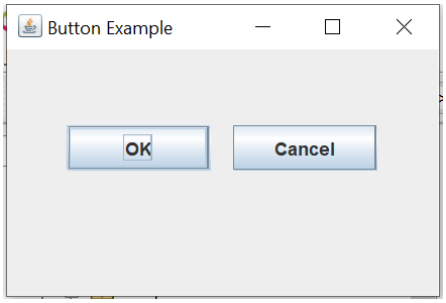
Commonly used methods of AbstractButton are

Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Example

```
import javax.swing.*;
public class JButtonExample {
public static void main(String[] args) {
    JFrame f=new JFrame("Button Example");
    JButton b1=new JButton("Ok");
    JButton b2=new JButton("Cancel");
    b1.setBounds(50,100,95,30);
    b2.setBounds(150,100,95,30);
    f.add(b1);
    f.add(b2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
}
}
```

Output



JCheckbox

JCheckBox is a part of Java Swing package . JCheckBox can be selected or deselected . It displays it state to the user . JCheckBox is an implementation to checkbox . JCheckBox inherits JToggleButton class.

Constructor of the class are :

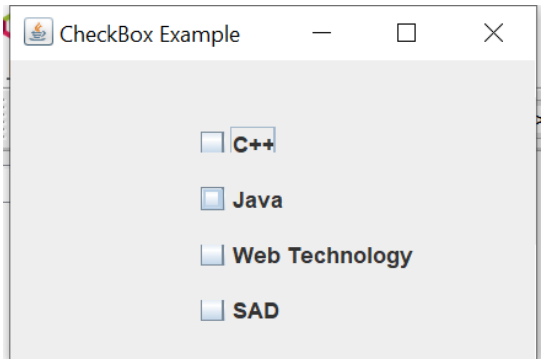
1. **JCheckBox()** : creates a new checkbox with no text or icon
2. **JCheckBox(Icon i)** : creates a new checkbox with the icon specified
3. **JCheckBox(Icon icon, boolean s)** : creates a new checkbox with the icon specified and the boolean value specifies whether it is selected or not.
4. **JCheckBox(String t)** :creates a new checkbox with the string specified
5. **JCheckBox(String text, boolean selected)** :creates a new checkbox with the string specified and the boolean value specifies whether it is selected or not.
6. **JCheckBox(String text, Icon icon)** :creates a new checkbox with the string and the icon specified.
7. **JCheckBox(String text, Icon icon, boolean selected)**: creates a new checkbox with the string and the icon specified and the boolean value specifies whether it is selected or not.

Example

```
import javax.swing.*;
public class JCheckBoxExample
{
     JCheckBoxExample(){
        JFrame f= new JFrame("CheckBox Example");
        JCheckBox checkBox1 = new JCheckBox("C++");
        checkBox1.setBounds(100,50, 200,50);
        JCheckBox checkBox2 = new JCheckBox("Java");
        checkBox2.setBounds(100,100, 200,50);
        JCheckBox checkBox3 = new JCheckBox("Web Technology");
        checkBox3.setBounds(100,150, 200,50);
        JCheckBox checkBox4 = new JCheckBox("SAD");
        checkBox4.setBounds(100,200, 200,50);
        f.add(checkBox1);
        f.add(checkBox2);
        f.add(checkBox3);
```

```
        f.add(checkBox4);
        f.setSize(400,400);
        f.setLayout(null);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
     }
public static void main(String args[])
    {
    new JCheckBoxExample();
    }
}
```

Output



JRadioButton

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

Commonly used constructors are:

Constructor	Description
JRadioButton()	Creates an unselected radio button with no text.
JRadioButton(String s)	Creates an unselected radio button with specified text.
JRadioButton(String s, boolean selected)	Creates a radio button with the specified text and selected status.

Commonly used methods are

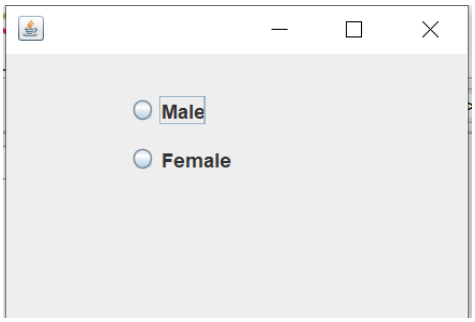
Methods	Description
void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Example

```
import javax.swing.*;
public class JRadioButtonExample {
JFrame f;
JRadioButtonExample(){
f=new JFrame();
JRadioButton r1=new JRadioButton("Male");
JRadioButton r2=new JRadioButton("Female");
r1.setBounds(75,50,100,30);
r2.setBounds(75,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(r1);
bg.add(r2);
f.add(r1);
f.add(r2);
f.setSize(300,300);
f.setLayout(null);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
}
public static void main(String[] args) {
    new JRadioButtonExample();
```

```
}
}
```

Output



JComboBox

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

Commonly used constructors are:

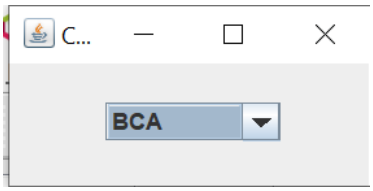
Constructor	Description
JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object[] items)	Creates a JComboBox that contains the elements in the specified array
JComboBox(Vector<?> items)	Creates a JComboBox that contains the elements in the specified Vector

Example

```
import javax.swing.*;
public class JComboBoxExample {
    JFrame f;
    JComboBoxExample() {
        f = new JFrame("ComboBox Example");
        String p[] = {"BCA", "BIT", "BSCCSIT", "BE Computer",
"BIM",};
        JComboBox cb = new JComboBox(p);
```

```
        cb.setBounds(50, 50, 90, 20);
        f.add(cb);
        f.setLayout(null);
        f.setSize(400, 500);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new JComboBoxExample();
    }
}
```

Output



JList

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits JComponent class.

Commonly used constructors

Commonly used methods

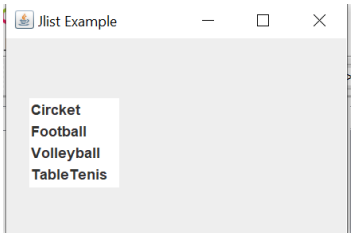
Constructor	Description
JList()	Creates a JList with an empty, read-only, model.
JList(ary[] listData)	Creates a JList that displays the elements in the specified array.
JList(ListModel<ary> dataModel)	Creates a JList that displays elements from the specified, non-null, model.

Example

```
import javax.swing.*;
public class JListExample
{
     JListExample(){
        JFrame f= new JFrame("JList Example");
        DefaultListModel<String> l1 = new DefaultListModel<>();
          l1.addElement("Circket");
          l1.addElement("Football");
          l1.addElement("Volleyball");
          l1.addElement("TableTenis");
```

```
            JList<String> list = new JList<>(l1);
            list.setBounds(100,100,75,75);
            f.add(list);
            f.setSize(400,400);
            f.setLayout(null);
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            f.setVisible(true);
      }
public static void main(String args[])
     {
    new JListExample();
     }
}
```

Output



JMenuBar

The JMenuBar class is used to display menubar on the window or frame. It may have several menus.

The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

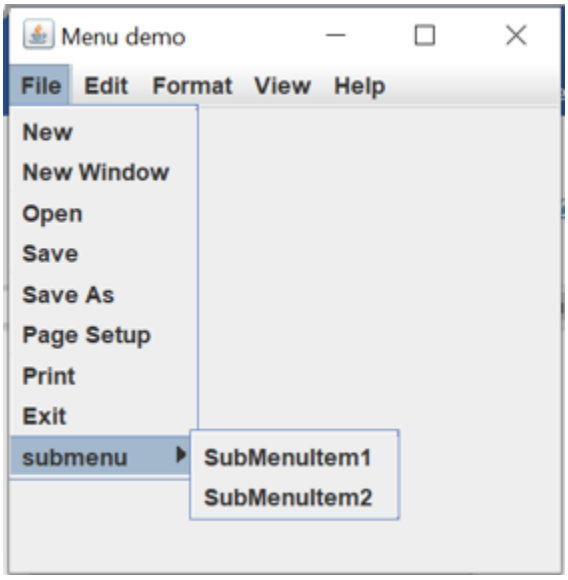
The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.

Example

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JMenubarExample extends JFrame
{
 static JMenuBar mb;
 static JMenu a,b,c,d,e,x1;
 static JMenuItem a1,a2,a3,a4,a5,a6,a7,a8,s1,s2;
 static JFrame f;
 public static void main(String[] args)
 {
```

```
        JMenubarExample m=new JMenubarExample();
        f=new JFrame("Menu demo");
        mb=new JMenuBar();
        a=new JMenu("File");
        b=new JMenu("Edit");
        c=new JMenu("Format");
        d=new JMenu("View");
        e=new JMenu("Help");
        x1=new JMenu("submenu");
        a1=new JMenuItem("New");
        a2=new JMenuItem("New Window");
        a3=new JMenuItem("Open");
        a4=new JMenuItem("Save");
        a5=new JMenuItem("Save As");
        a6=new JMenuItem("Page Setup");
        a7=new JMenuItem("Print");
        a8=new JMenuItem("Exit");
        s1=new JMenuItem("SubMenuItem1");
        s2=new JMenuItem("SubMenuItem2");
        a.add(a1);
        a.add(a2);
        a.add(a3);
        a.add(a4);
        a.add(a5);
        a.add(a6);
        a.add(a7);
        a.add(a8);
        x1.add(s1);
        x1.add(s2);
        a.add(x1);
        mb.add(a);
        mb.add(b);
        mb.add(c);
        mb.add(d);
        mb.add(e);
        f.setJMenuBar(mb);
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```

Output



JTable

The JTable class is a part of Java Swing Package and is generally used to display or edit two-dimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

Constructors in JTable:

1. **JTable():** A table is created with empty cells.
2. **JTable(int rows, int cols):** Creates a table of size rows * cols.
3. **JTable(Object[][] data, Object []Column):** A table is created with the specified name where []Column defines the column names.

Functions in JTable:

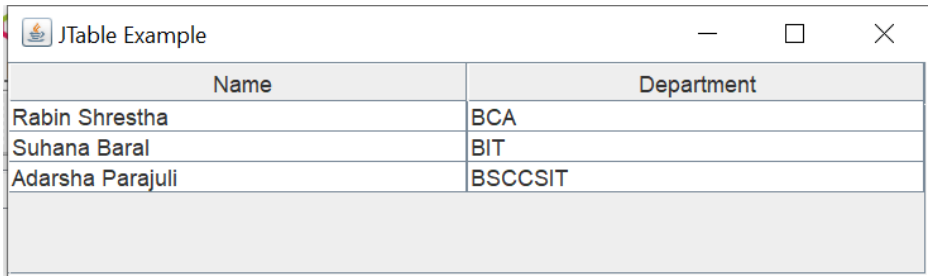
1. **addColumn(TableColumn []column) :** adds a column at the end of the JTable.
2. **clearSelection() :** Selects all the selected rows and columns.
3. **editCellAt(int row, int col) :** edits the intersecting cell of the column number col and row number row programmatically, if the given indices are valid and the corresponding cell is editable.
4. **setValueAt(Object value, int row, int col) :** Sets the cell value as 'value' for the position row, col in the JTable.

Example

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class JTableExample {
    JFrame f;
    JTable j;
    JTableExample()
    {
```

```
        f = new JFrame();
        f.setTitle("JTable Example");
        String[][] data = {
            { "Rabin Shrestha", "BCA" },
            { "Suhana Baral","BIT" },
            {"Adarsha Parajuli","BSCCSIT"}
        };
        String[] columnNames = { "Name", "Department" };
        j = new JTable(data, columnNames);
        j.setBounds(30, 40, 200, 300);
        JScrollPane sp = new JScrollPane(j);
        f.add(sp);
        f.setSize(500, 200);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new JTableExample();
    }
}
```

Output



Layout Management

The LayoutManagers are used to arrange components in a particular manner. The **Java LayoutManagers** facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers.

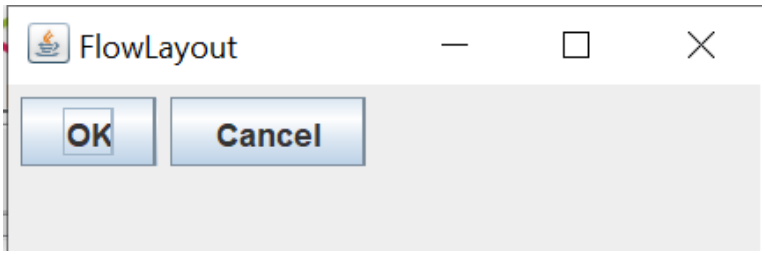
FlowLayout:

It arranges the components in a container like the words on a page. It fills the top line from **left to right and top to bottom**. The components are arranged in the order as they are added i.e. first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be **left, center or right aligned.**

Example

```
import java.awt.*;
import javax.swing.*;

public class MyFlowLayout
{
JFrame f;
MyFlowLayout()
{
    f=new JFrame("FlowLayout");
    JButton b1=new JButton("OK");
    JButton b2=new JButton("Cancel");
    f.add(b1);
    f.add(b2);
    f.setLayout(new FlowLayout(FlowLayout.LEFT));
    f.setSize(300,300);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
}
public static void main(String[] args)
{
    new MyFlowLayout();
}
}
```

Output



GridLayout

It arranges all the components in a grid of **equally sized cells**, adding them from the **left to right** and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.

Example

```
import java.awt.*;
import javax.swing.*;
public class MyGridLayout{
```

```
JFrame f;
MyGridLayout(){
    f=new JFrame("GridLayout");
    JButton b1=new JButton("Button 1");
    JButton b2=new JButton("Button 2");
    JButton b3=new JButton("Button 3");
    JButton b4=new JButton("Button 4");
    JButton b5=new JButton("Button 5");
    JButton b6=new JButton("Button 6");
    JButton b7=new JButton("Button 7");
    JButton b8=new JButton("Buttton 8");
    JButton b9=new JButton("Button 9");
    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.add(b6);f.add(b7);f.add(b8);f.add(b9);
    f.setLayout(new GridLayout(3,3));
    f.setSize(300,300);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
}
public static void main(String[] args) {
    new MyGridLayout();
}
}
```

Output

BorderLayout

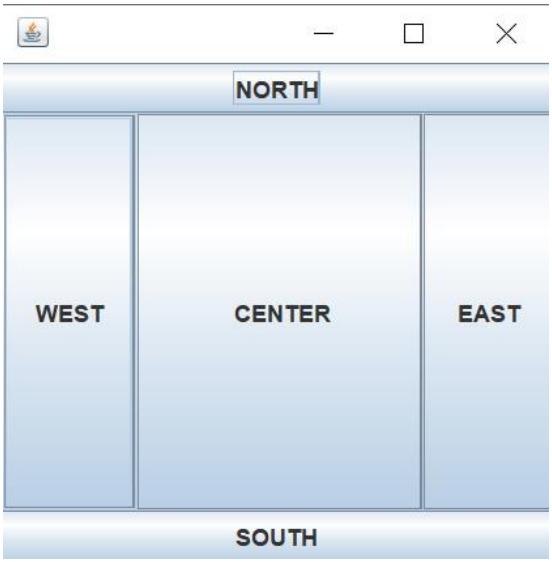
It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The

components added to the center gets neither its preferred height or width. It covers the remaining area of the container.

Example

```
import java.awt.*;
import javax.swing.*;
public class BorderLayoutExample {
JFrame f;
BorderLayoutExample(){
    f=new JFrame();
    JButton b1=new JButton("NORTH");
    JButton b2=new JButton("SOUTH");
    JButton b3=new JButton("EAST");
    JButton b4=new JButton("WEST");
    JButton b5=new JButton("CENTER");
    f.add(b1,BorderLayout.NORTH);
    f.add(b2,BorderLayout.SOUTH);
    f.add(b3,BorderLayout.EAST);
    f.add(b4,BorderLayout.WEST);
    f.add(b5,BorderLayout.CENTER);
    f.setSize(300,300);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
}
public static void main(String[] args) {
    new BorderLayoutExample();
}
}
```

Output



Event Handling

An event is a signal to the program that something has happened. It can be triggered by typing in a text field, selecting an item from the menu etc. The action is initiated outside the scope of the program and it is handled by a piece of code inside the program. Events may also be triggered when timer expires, hardware or software failure occurs, operation completes, counter is increased or decreased by a value etc.

Event handler: The code that performs a task in response to an event is called event handler.

Event handling: It is the process of responding to events that can occur at any time during execution of a program.

Event source: It is an object that generates the events. Usually the event source is a button or the other component that the user can click but any Swing component can be an event source. The job of the event source is to accept registration, get events from the user and call the listener's event handling method.

Event Listener: It is an object that watch for (i.e. listen for) events and handles them when they occur. It is basically a consumer that receives events from the source. To sum up, the job of an event listener is to implement the interface, register with the source and provide the event handling.

Listener interface: It is an interface which contains methods that the listener must implement and the source of the event invokes when the event occurs.

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change in state of any object. **For Example** Pressing a button, Entering a character in Textbox, Clicking or Dragging a mouse, etc.

Some Event Classes and Interfaces

Event classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener

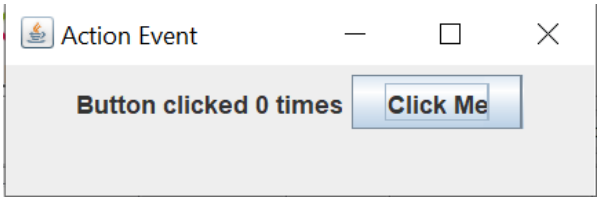
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

Action Event

Example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class EventExample extends JFrame implements ActionListener
{
    private int count=0;
    JLabel lblData;
    EventExample()
    {
        setLayout(new FlowLayout());
                {
                    lblData=new JLabel("Button clicked 0 times");
                    JButton btnClick=new JButton("Click Me");
                    btnClick.addActionListener(this);
                    add(lblData);
                    add(btnClick);
                }
    }
    public void actionPerformed(ActionEvent e)
    {
        count++;
        lblData.setText("Button Clicked "+count+" Times ");
    }
}
public class ActionEventExample {
    public static void main(String[] args)
    {
        EventExample frame=new EventExample();
        frame.setTitle("Action Event");
        frame.setSize(300,100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Output

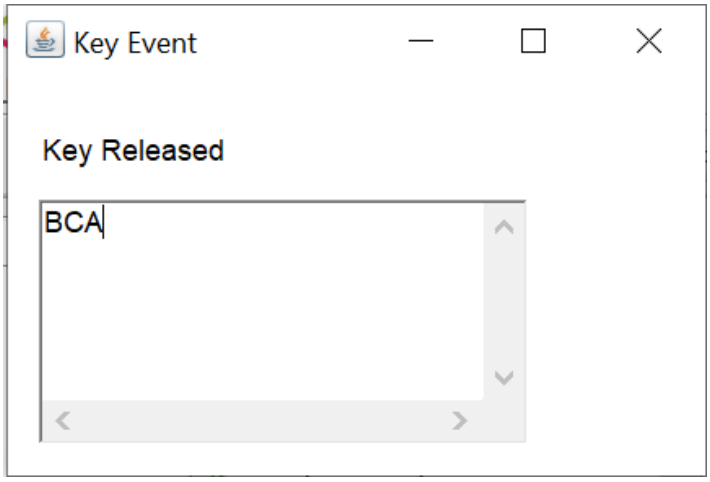


Key Event

Example

```
import java.awt.*;
import java.awt.event.*;
class KeyListenerExample extends Frame implements KeyListener {
    Label l;
    TextArea area;
    KeyListenerExample()
    {
        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300,300);
        area.addKeyListener(this);
        add(l);
        add(area);
       setTitle("Key Event");
        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }
    public void keyPressed(KeyEvent e)
    {
        l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e) {
        l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e)
    {
        l.setText("Key Typed");
    }
    public static void main(String[] args)
    {
        new KeyListenerExample();
    }
}
```

Output



Mouse Event

Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements
MouseListener{
    Label l;
    MouseListenerExample(){
        addMouseListener(this);

        l=new Label();
        l.setBounds(20,50,100,20);
        add(l);
        setTitle("Mouse Event");
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        l.setText("Mouse Clicked");
    }
    public void mouseEntered(MouseEvent e) {
        l.setText("Mouse Entered");
    }
    public void mouseExited(MouseEvent e) {
        l.setText("Mouse Exited");
    }
    public void mousePressed(MouseEvent e) {
        l.setText("Mouse Pressed");
    }
```

```
    public void mouseReleased(MouseEvent e) {
        l.setText("Mouse Released");
    }
public static void main(String[] args) {
    new MouseListenerExample();
}
}
```

Output

MDI (Multiple Document Interface)

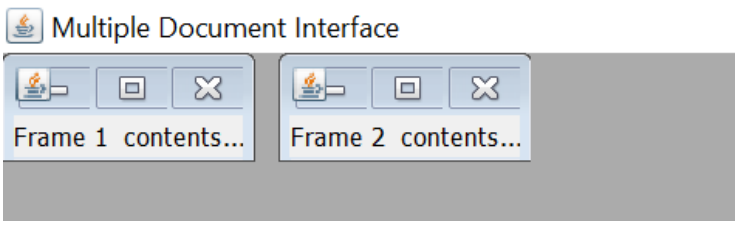
In an MDI application, one main window is opened and multiple child windows are open within the main window. In an MDI application, we can open multiple frames that will be instances of the JInternalFrame class. We can organize multiple internal frames in many ways. For example we can maximize and minimize them, we can view them side by side in a tiled fashion, or we can view them in a cascaded form.

Example

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JInternalFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;

public class MDIExample extends JFrame {
  private final JDesktopPane desktopPane = new JDesktopPane();

  public MDIExample() {
    JInternalFrame frame1 = new JInternalFrame("Frame 1", true,
true, true,
        true);

    JInternalFrame frame2 = new JInternalFrame("Frame 2", true,
true, true,
        true);
```

```
    frame1.getContentPane().add(new        JLabel("Frame        1
contents..."));
    frame1.pack();
    frame1.setVisible(true);

    frame2.getContentPane().add(new        JLabel("Frame        2
contents..."));
    frame2.pack();
    frame2.setVisible(true);

    int x2 = frame1.getX() + frame1.getWidth() + 10;
    int y2 = frame1.getY();
    frame2.setLocation(x2, y2);

    desktopPane.add(frame1);
    desktopPane.add(frame2);

    this.add(desktopPane, BorderLayout.CENTER);

    this.setMinimumSize(new Dimension(300, 300));
  }
  public static void main(String[] args) {
    try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName
());
    } catch (Exception e) {
      e.printStackTrace();
    }
    SwingUtilities.invokeLater(() -> {
      MDIExample frame = new MDIExample();
      frame.pack();
      frame.setVisible(true);
      frame.setExtendedState(frame.MAXIMIZED_BOTH);
    });
  }
}
```

Output



Adaptor class

Java adapter classes provide the default implementation of listener interfaces. If we inherit the adapter class, we will not be forced to provide the implements of all the methods of listener interfaces. So it saves code.

Event Adapters classes are abstract class that provides some methods used for avoiding the heavy coding. Adapter class is defined for the listener that has more than one abstract methods. A source generates an Event and sends it to one or more listeners registered with the source. Once event is received by the listener, they process the event and then return. Events are supported by a number of Java packages, like java.util, java.awt and java.awt.event.

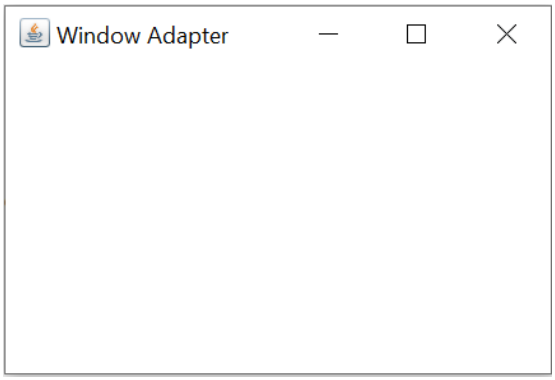
java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Window Adapter Example

```
import java.awt.*;
import java.awt.event.*;
public class WindowAdapterExample{
    Frame f;
    WindowAdapterExample(){
        f=new Frame("Window Adapter");
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                f.dispose();
            }
        });
        f.setSize(300,200);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String[] args) {
    new WindowAdapterExample();
}
}
```

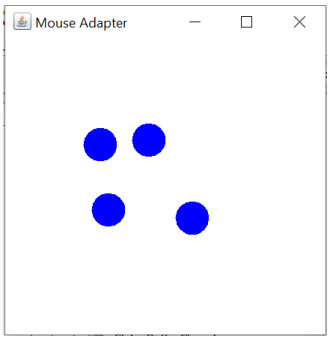
Output



MouseAdapter Example

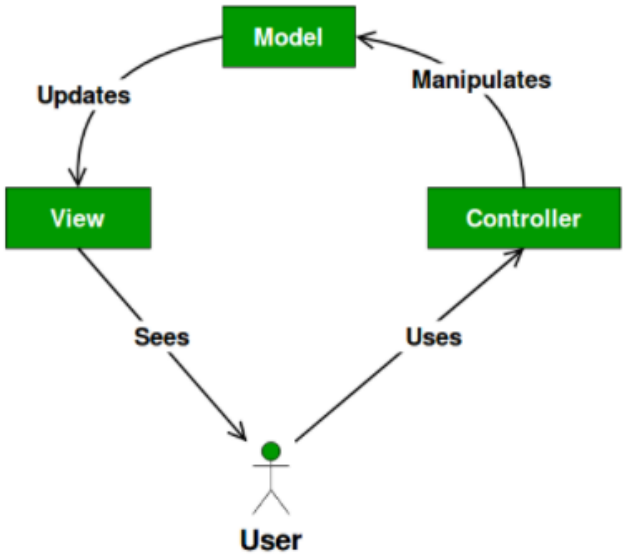
```
import java.awt.*;
import java.awt.event.*;
public class MouseAdapterExample extends MouseAdapter{
    Frame f;
    MouseAdapterExample(){
        f=new Frame("Mouse Adapter");
        f.addMouseListener(this);

        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        Graphics g=f.getGraphics();
        g.setColor(Color.BLUE);
        g.fillOval(e.getX(),e.getY(),30,30);
    }

public static void main(String[] args) {
    new MouseAdapterExample();
}
}
```

Output



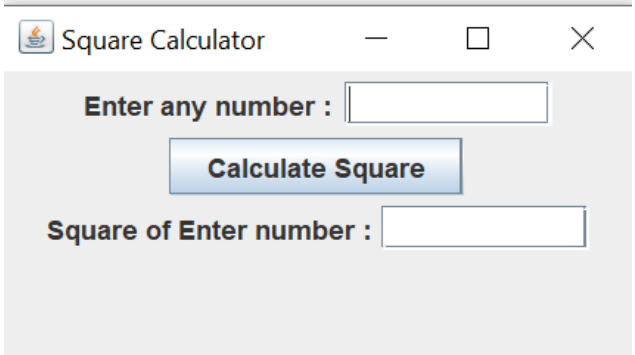
MVC (Model Viewer Controller) design pattern

The **Model View Controller** (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.
MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application. You're still going to need business logic layer, maybe some service layer and data access layer.



- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

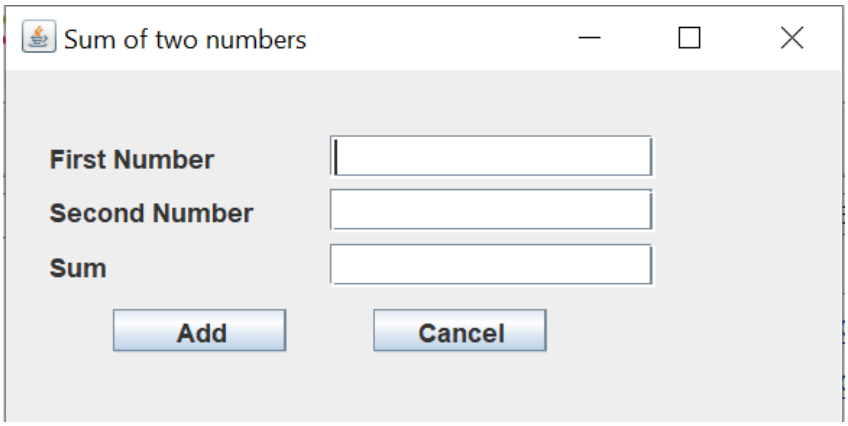
Write a java GUI program to calculate square of entered number.



```
import java.awt.event.*;
import javax.swing.*;
public class SquareNumber
{
    SquareNumber()
    {
        JFrame jframe=new JFrame("Square Calculator");
        jframe.setLayout(null);
        jframe.setSize(400,200);
        jframe.setVisible(true);

        JLabel lblFirstNumber=new JLabel("Enter any Number: ");
        lblFirstNumber.setBounds(20, 30, 150, 20);
        jframe.add(lblFirstNumber);
        JTextField txtFirstNumber=new JTextField();
        txtFirstNumber.setBounds(180,30,150,20);
        jframe.add(txtFirstNumber);

        JButton btnSquare=new JButton("Square");
        btnSquare.setBounds(40, 60, 100, 20);
        jframe.add(btnSquare);

        JLabel lblSquare=new JLabel("Square of Enter Number:
");
        lblSquare.setBounds(20, 90, 150, 20);
        jframe.add(lblSquare);
        JTextField txtSquare=new JTextField();
        txtSquare.setBounds(180, 90, 150, 20);
        jframe.add(txtSquare);

        btnSquare.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
             int a=Integer.parseInt(txtFirstNumber.getText());
             int square=a*a;
             txtSquare.setText(" "+square);
            }
        }
        );
    }
public static void main(String args[])
{
        SwingUtilities.invokeLater(new Runnable()
        {
```

```
                public void run()
                {
                    new SquareNumber();
                }
            }
            );
    }
    }
```

Write a java GUI program to calculate the sum of two numbers.

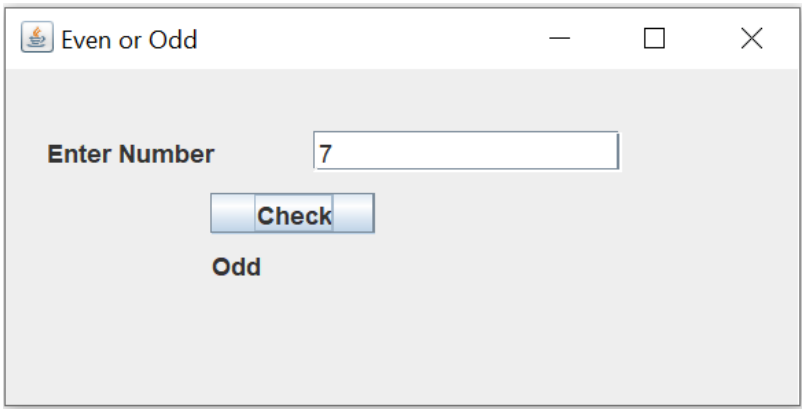



```
import java.awt.event.*;
import javax.swing.*;
public class Sum
{
    Sum()
    {
     JFrame jframe=new JFrame("Sum of two numbers");
     jframe.setLayout(null);
     jframe.setSize(400, 200);
     jframe.setVisible(true);

     JLabel lblFirstNumber=new JLabel("First Number");
     lblFirstNumber.setBounds(20, 30, 150, 20);
     jframe.add(lblFirstNumber);

     JTextField txtFirstNumber = new JTextField();
     txtFirstNumber.setBounds(150, 30, 150, 20);
     jframe.add(txtFirstNumber);

     JLabel lblSecondNumber=new JLabel("Second Number");
     lblSecondNumber.setBounds(20, 55, 150, 20);
     jframe.add(lblSecondNumber);
```

```
 JTextField txtSecondNumber=new JTextField();
 txtSecondNumber.setBounds(150, 55, 150, 20);
 jframe.add(txtSecondNumber);

 JLabel lblSum=new JLabel("Sum");
 lblSum.setBounds(20,80,150,20);
 jframe.add(lblSum);

 JTextField txtSum=new JTextField();
 txtSum.setBounds(150, 80, 150,20);
 jframe.add(txtSum);

 JButton btnAdd=new JButton("Add");
 btnAdd.setBounds(50, 110, 80, 20);
 jframe.add(btnAdd);

 JButton btnCancel=new JButton("Cancel");
 btnCancel.setBounds(170, 110, 80, 20);
 jframe.add(btnCancel);

 btnAdd.addActionListener(new ActionListener()
 {
   public void actionPerformed(ActionEvent ae)
     {
         int a=Integer.parseInt(txtFirstNumber.getText());
         int b=Integer.parseInt(txtSecondNumber.getText());
         int c=a+b;
         txtSum.setText(" "+c);
     }
 }
 );

 btnCancel.addActionListener(new ActionListener()
 {
     public void actionPerformed(ActionEvent ae)
     {
         txtFirstNumber.setText(null);
         txtSecondNumber.setText(null);
         txtSum.setText(null);
     }
 }
 );
}
public static void main(String args[])
{
    SwingUtilities.invokeLater(new Runnable()
    {
```

```
            public void run()
            {
                new Sum();
            }
        }
        );
    }
}
```

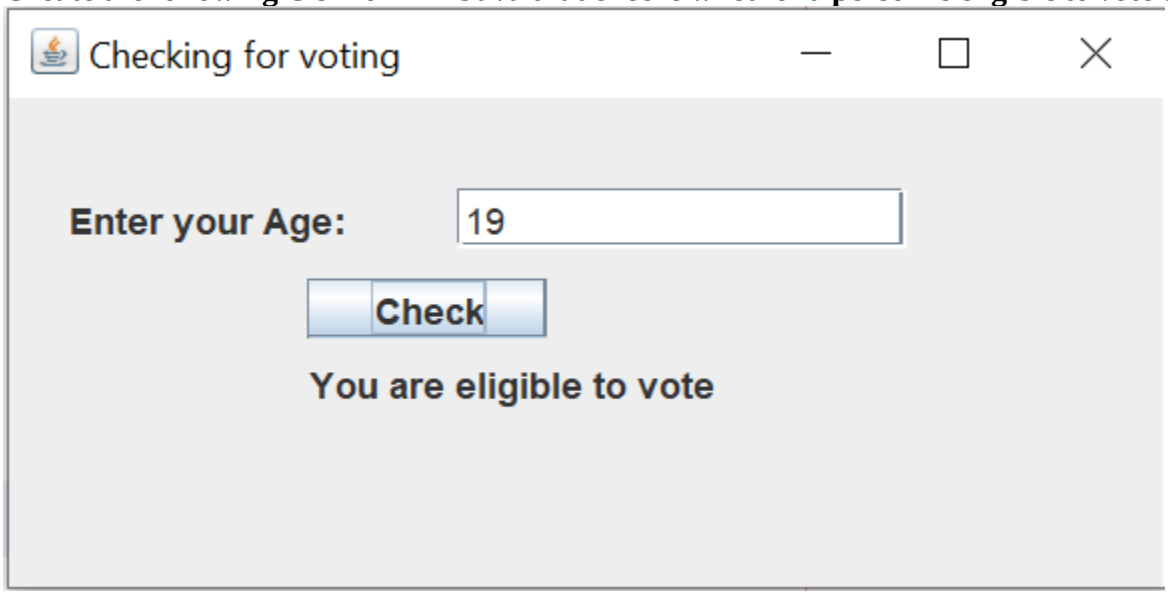
Create the following GUI form in Java that checks a number entered in the text box is even or odd.




```
import java.awt.event.*;
import javax.swing.*;
public class EvenOdd
{
    EvenOdd()
    {
     JFrame jframe=new JFrame("Even or Odd");
     jframe.setLayout(null);
     jframe.setSize(400, 200);
     jframe.setVisible(true);

     JLabel lblNumber=new JLabel("Enter Number");
     lblNumber.setBounds(20, 30, 150, 20);
     jframe.add(lblNumber);

     JTextField txtNumber = new JTextField();
     txtNumber.setBounds(150, 30, 150, 20);
     jframe.add(txtNumber);

     JButton btnCheck=new JButton("Check");
     btnCheck.setBounds(100, 60, 80, 20);
     jframe.add(btnCheck);
```

```
        JLabel lblResult=new JLabel("");
        lblResult.setBounds(100, 85, 150, 20);
        jframe.add(lblResult);
        btnCheck.addActionListener(new ActionListener()
        {
          public void actionPerformed(ActionEvent ae)
            {
                int a=Integer.parseInt(txtNumber.getText());
                if(a%2==0)
                   lblResult.setText("Even");
                else
                    lblResult.setText("Odd");
            }
        }
        );
    }
    public static void main(String args[])
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                new EvenOdd();
            }
        }
        );
    }
}
```

Create the following GUI form in Java that checks whether a person is eligible to vote or not.



```
import java.awt.event.*;
import javax.swing.*;
public class Vote
{
    Vote()
    {
     JFrame jframe=new JFrame("Checking for voting");
     jframe.setLayout(null);
     jframe.setSize(400, 200);
     jframe.setVisible(true);

     JLabel lblAge=new JLabel("Enter your Age:");
     lblAge.setBounds(20, 30, 150, 20);
     jframe.add(lblAge);

     JTextField txtAge = new JTextField();
     txtAge.setBounds(150, 30, 150, 20);
     jframe.add(txtAge);

     JButton btnCheck=new JButton("Check");
     btnCheck.setBounds(100, 60, 80, 20);
     jframe.add(btnCheck);

     JLabel lblResult=new JLabel("");
     lblResult.setBounds(100, 85, 150, 20);
     jframe.add(lblResult);
     btnCheck.addActionListener(new ActionListener()
     {
       public void actionPerformed(ActionEvent ae)
         {
             int a=Integer.parseInt(txtAge.getText());
             if(a>=18)
                lblResult.setText("You are eligible to vote");
             else
                 lblResult.setText("Yor   are   not   eligible   to
vote");
         }
     }
     );
    }
    public static void main(String args[])
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                new Vote();
```

```
                }
            }
            );
        }
    }
```

Unit-12

Introduction to Java Applets

Introduction

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

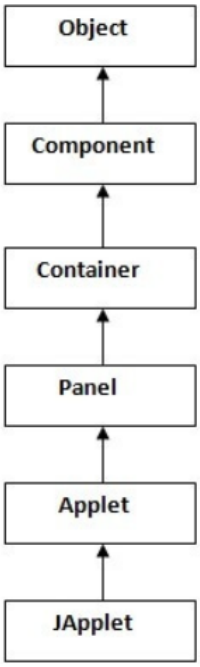
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet



Java Application vs. Java Applet

Java Application	Java Applet
1. Applications are just like a Java programs that can be execute independently without using the web browser.	1. Applets are small Java programs that are designed to be included with the HTML web document. They require a Java-enabled web browser for execution.
2. Application program requires a main function for its execution.	2. Applet does not require a main function for its execution.
3. Java application programs have the full access to the local file system and network.	3. Applets don't have local disk and network access.
4. Applications can access all kinds of resources available on the system.	4. Applets can only access the browser specific services. They don't have access to the local system.
5. Applications can executes the programs from the local system.	5. Applets cannot execute programs from the local machine.
6. An application program is needed to perform some task directly for the user.	6. An applet program is needed to perform small tasks or the part of it.

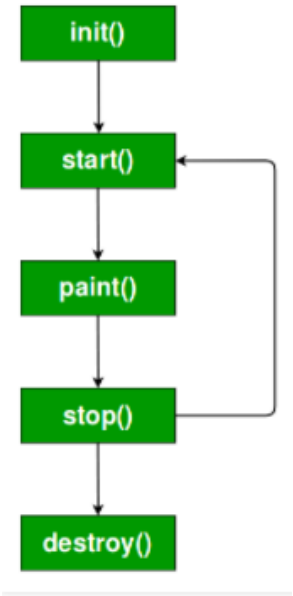
AppletExample(AppletExample.java)

```
import java.applet.Applet;
import java.awt.Graphics;
public class AppletExample extends Applet {
    public void init() {

    }
public void main (Graphics g)
{
 g.drawString("Welcome", 150, 150);
}
}
```

(AppletExample.html)

```
<html>
<body>
<applet code="AppletExample.class" width="300" height="300">
</applet>
</body>
</html>
```

Applet life cycle



public void init(): This method is used to perform any initialization that needed for the applet. It is called automatically when the applet is first loaded into the browser and is called only once during the life cycle of the applet. In this method, we generally perform startup activities such as adding GUI components; loading resources such as images, audio, font; creating threads; and getting string parameter values from the APPLET tag in the HTML page.

public void start(): After the applet is loaded and initialized, the Java environment automatically calls the start() method. It also called when the user returns to the HTML page that contains the applet after browsing through other webpages. This method is used to start the processing for the applet. For example, Action performed here might include starting an animation or starting other threads of execution. Unlike the init() method, the start() method may be called more than once during the life cycle of an applet.

public void stop(): This method is called automatically by the browser when the user moves off the HTML page containing the applet. It is generally used to suspend the applet's execution so that it does not take up system resources when the user is not viewing the HTML page or has quit the browser. For example Processor intensive activities such as animation, playing audio files or performing calculations in a thread can be stopped which not visible to the user until the user returns to the page.

public void destroy(): This method called after the stop() method when the user exits the web browser normally. This method is used to clean up resources allocated to the applet that is managed by the local operating system. Like the init() method, it is called only once in the lifetime of the applet.

Applet Graphics

AppletGraphics (AppletGraphics.java)

```
import java.applet.Applet;
import java.awt.*;
public class AppletGraphics extends Applet {
    public void init()
    {

    }

    public void paint(Graphics g)
{
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);
g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
}
}
```

(AppletGraphics.html)

```
<html>
<body>
<applet code="AppletGraphics.class" width="300" height="300">
</applet>
</body>
</html>
```

Applet Animation

```
import java.applet.Applet;
import java.awt.*;
public class AnimationExample extends Applet {
public void paint(Graphics g)
{
```

```
    int x=150,y=150,a=10,b=10;
    for(int i=0;i<15;i++)
    {
     try
     {
         Thread.sleep(1000);
     }
     catch(InterruptedException e)
     {

     }
     g.drawOval(x,y,a,b);
     x=x-10;
     y=y-10;
     a=a+8;
     b=b+8;
    }
}
    public void init() {
        // TODO start asynchronous download of heavy resources
    }

    // TODO overwrite start(), stop() and destroy() methods
}
```

(AnimationExample.html)

```
<html>
<body>
<applet code="AnimationExample.class" width="300" height="300">
</applet>
</body>
</html>
```

Unit-13

Database Programming using JDBC

The design of JDBC

Java Database Connectivity (JDBC) is an Application Programming Interface (API), from Sun microsystem that is used by the Java application to communicate with the relational databases from different vendors. JDBC and database drivers work in tandem to access spreadsheets and databases. **Design of JDBC** defines the components of JDBC, which is used for connecting to the database.

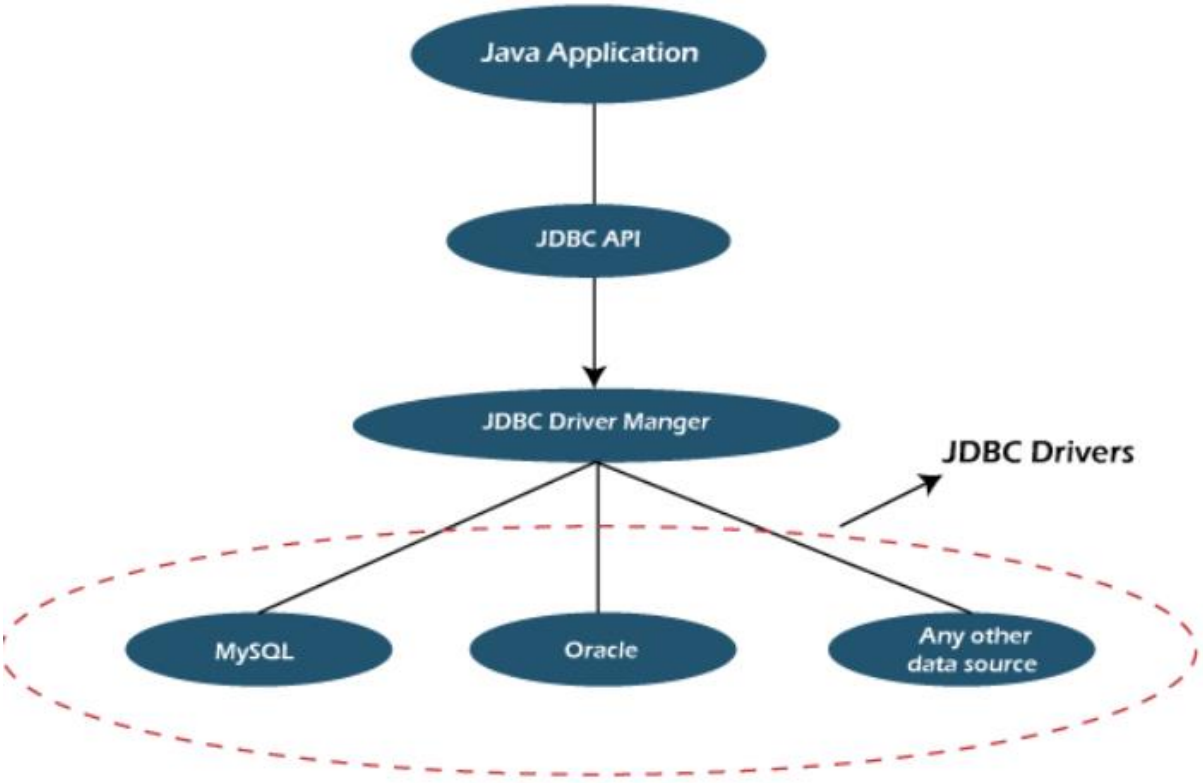


Figure: Components of JDBC

JDBC has four major components that are used for the interaction with the database.

1. JDBC API
2. JDBC Test Suite
3. JDBC Driver Manger
4. JDBC ODBC Bridge Driver

1) JDBC API: JDBC API provides various interfaces and methods to establish easy connection with different databases.

```
javax.sql.*;
java.sql.*;
```

2) JDBC Test suite: JDBC Test suite facilitates the programmer to test the various operations such as deletion, updation, insertion that are being executed by the JDBC Drivers.

3) JDBC Driver manager: JDBC Driver manager loads the database-specific driver into an application in order to establish the connection with the database. The JDBC Driver manager is also used to make the database-specific call to the database in order to do the processing of a user request.

4) JDBC-ODBC Bridge Drivers: JDBC-ODBC Bridge Drivers are used to connect the database drivers to the database. The bridge does the translation of the JDBC method calls into the ODBC method call. It makes the usage of the sun.jdbc.odbc package that encompasses the native library in order to access the ODBC (Open Database Connectivity) characteristics.

JDBC Driver Types

JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

1. **JDBC-ODBC Bridge Driver**
2. **Native Driver**
3. **Network Protocol Driver**
4. **Thin Driver**

Type 1 - JDBC-ODBC Bridge Driver

Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases. The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

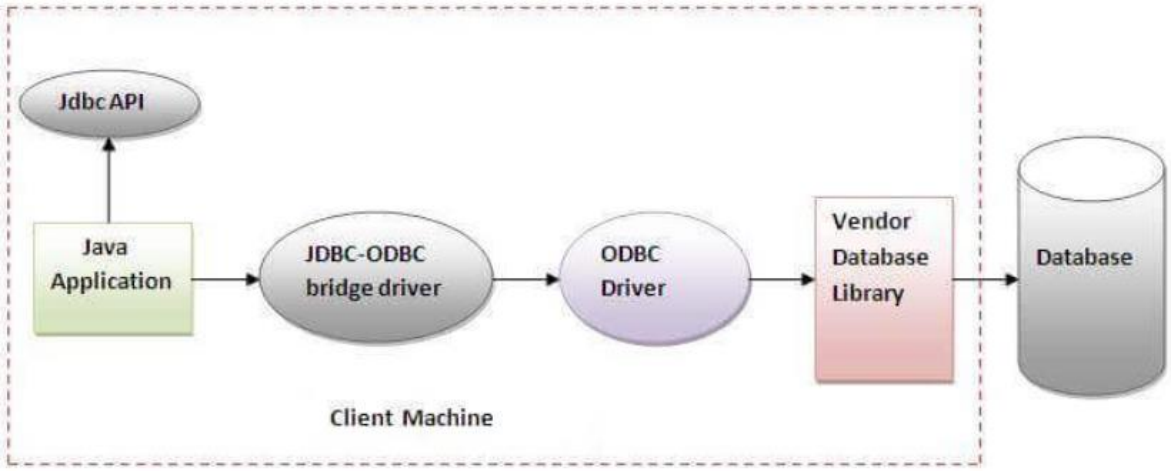


Figure- JDBC-ODBC Bridge Driver

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.
- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

From Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Type 2 – Native Driver

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver. Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

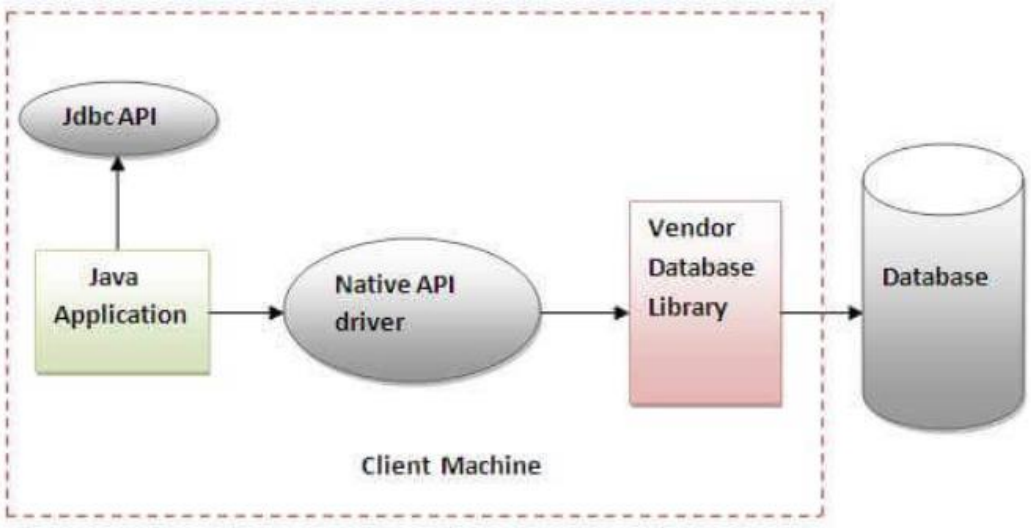


Figure- Native API Driver

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver
- It is a database dependent driver.

Type 3 – Network Protocol Driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation. If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

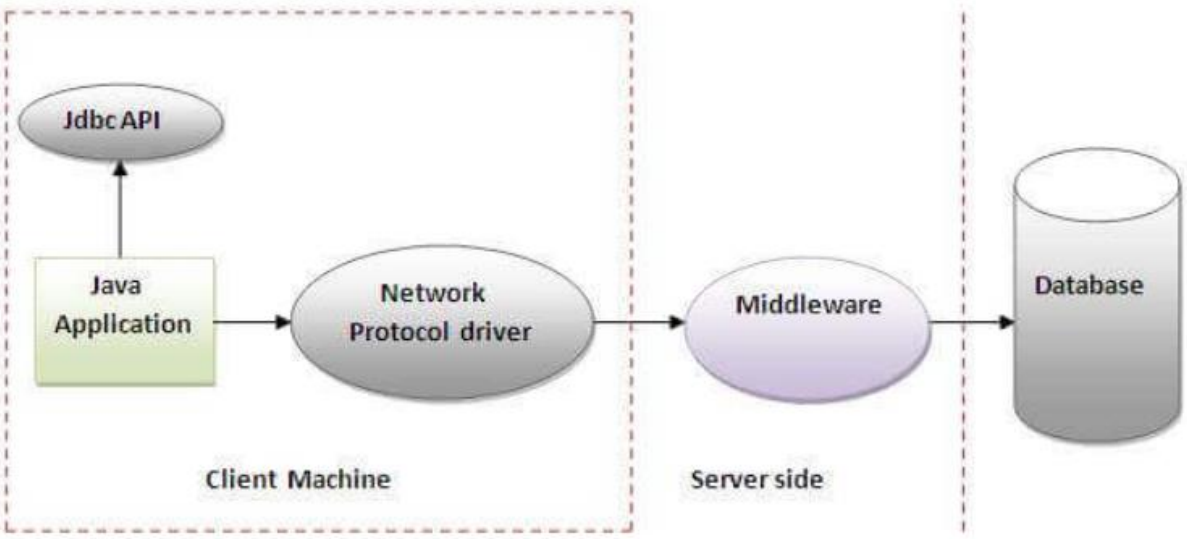


Figure- Network Protocol Driver

- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.
- Switch facility to switch over from one database to another database.

Type 4 – Thin Driver

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver. If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.

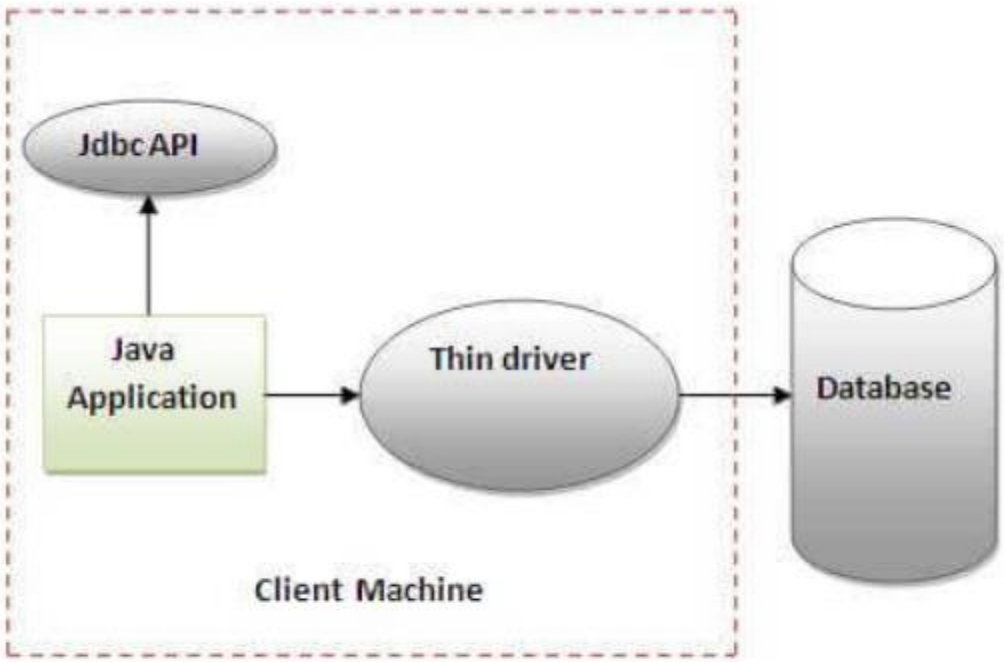


Figure- Thin Driver

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server. In this model, a JDBC driver is deployed on the client.

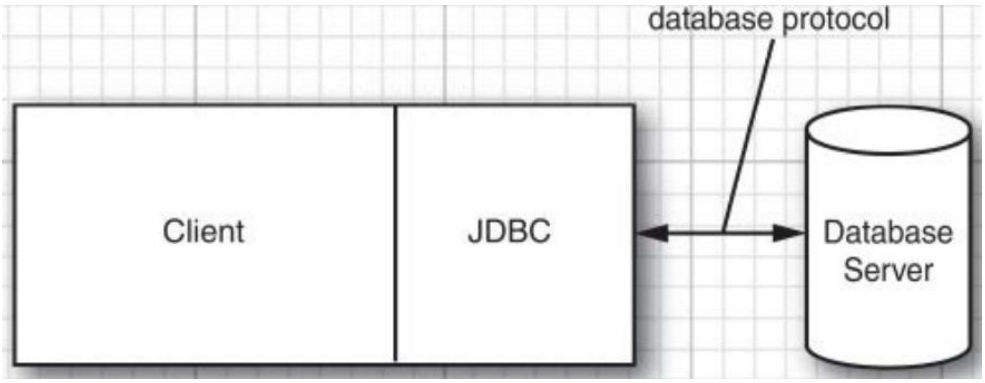


Figure: A traditional Client/Server Application

However, nowadays it is far more common to have a three-tier model where the client application does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java desktop application, a web browser, or a mobile app. Communication between the client and the middle tier typically occurs through HTTP. JDBC manages the communication between the middle tier and the back-end database.

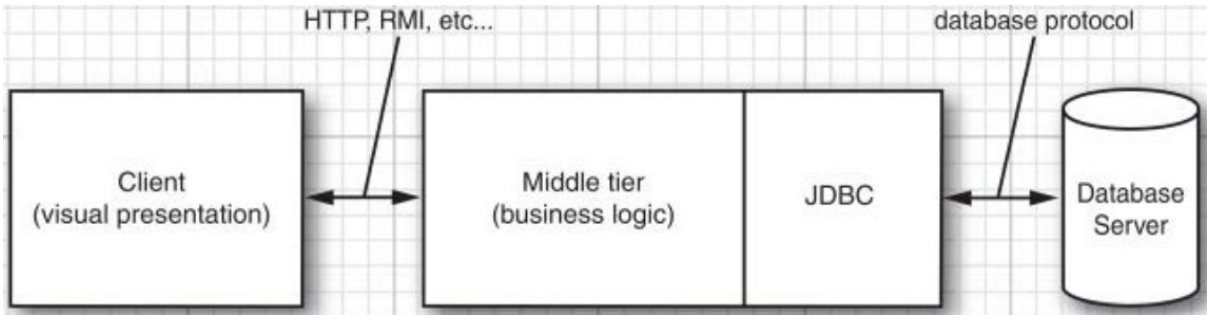


Figure: A three Tier application

JDBC Configuration

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. Load a JDBC Driver class
2. Establish a Connection
3. Create a Statement
4. Execute Sql queries
5. Process the ResultSet
6. Close Connection

Step-1:

Load a JDBC Driver : To load a class at runtime into JVM, we need to call a static method called **forName()** of java.lang.Class. By calling the forName() method we can load the JDBC Driver class into the JVM.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Step-2:

Establish a Connection : By using the DriverManager class, we can establish the connection to the database. By calling the **getConnection(String url, String user,String password)** static factory method in DriverManager class, we can obtain a Connection object. To get the connection object we need to proved the connection url as a first parameter and database username and password as second and third parameters.

```
Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3307/test"
,"root","");
```

Step-3:

Create a Statement : In order to send the SQL commands to database from our java program, we need Statement object. We can get the Statement object by calling the **createStatement()** method on **connection.**

```
Statement stmt = con.createStatement();
```

Step-4:

Execute Sql queries : Inorder to execute the SQL commands on database, Statement interface provides provides three different methods:

- executeUpdate()
- executeQuery()
- execute()

When we want to run the **non-select** operations then we can choose **executeUpdate()**

```
int count = stmt.executeUpdate("non-select command");
```

When we want to execute **select** operations then we can choose **executeQuery()**

```
ResultSet rs = stmt.executeQuery("select command");
```

When we want to run both select and non-select operations, then we can use **execute()**

```
boolean isTrue = stmt.executeQuery("select / non-select
command");
```

Step-5:

Process the ResultSet : For the select operations, we use **executeQuery()** method. The executed query returns the data in the form of ResultSet object. To process the data we need to go through the ResultSet.

```
ResultSet rs = stmt.executeQuery("select * from student");

while(rs.next()){
      System.out.println(rs.getInt(1));
}
```

Step-6:

Close Connection : Last but not least, finally we need to close the connection object. It is very important step to close the connection. Else we may get JDBCConnectionException exception.

```
conn.close();
```

To connect java application with the mysql database, **mysqlconnector.jar** file is required to be loaded.

download the jar file mysql-connector.jar

Example

```
import java.sql.*;
import javax.swing.*;
public class JDBCProgram {
   public static void main(String args[]) {
    try
    {
       //step-1 Load a JDBC Driver class
       Class.forName("com.mysql.cj.jdbc.Driver");
       //step-2 Establish a Connection
       Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3307/test"
);
       //step-3 Create a Statement
       Statement stmt=conn.createStatement();
       String sql="select * from student";
       //step-4 Execute Sql queries
       ResultSet rs=stmt.executeQuery(sql);
       //step-5 Process the ResultSet
       while(rs.next())
       {
           System.out.println(rs.getInt(2)+"     "+rs.getString(3)+"
"+rs.getString(4));
       }
       //step-6 Close Connection
```

```
        conn.close();
    }
    catch(Exception e)
    {
        JOptionPane.showMessageDialog(null,e);
    }
}
}
```

Output

```
run:
101 Ram Chitwan
102 Mahesh Butwal
105 Nabin Tandi
103 Kalpana Pokhara
BUILD SUCCESSFUL (total time: 0 seconds)
```

Working with JDBC Statements

How to use the JDBC Statement to execute SQL statements, obtain results, and deal with errors.

Executing SQL Statements

To execute a SQL statement, you first create a **Statement** object. To create statement objects, use the **Connection** object that you obtained from the call to **DriverManager.getConnection**.

```
Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example

```
String command = "UPDATE Books"
+ " SET Price = Price - 5.00"
+ " WHERE Title NOT LIKE '%Introduction%'";
```

Then call the **executeUpdate** method of the Statement interface:

```
stat.executeUpdate(command);
```

The **executeUpdate** method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count.

When you execute a query, you are interested in the result. The **executeQuery** object returns an object of type **ResultSet** that you can use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

The basic loop for analyzing a result set looks like this:

```
while (rs.next())
{
look at a row of the result set
```

```
}
```

A large number of accessor methods give you this information.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

Managing Connections, Statements and Result Sets

Every **Connection** object can create one or more Statement objects. We can use the same **Statement** object for multiple unrelated commands and queries. However, a statement has at most one open result set. When we are done using a **ResultSet, Statement, or Connection,** we should call the **close** method immediately. To make absolutely sure that a connection object cannot possibly remain open, use a try with resources statement:

```
try (Connection conn = . . .)
{
        Statement stat = conn.createStatement();
        ResultSet result = stat.executeQuery(queryString);
        process query result
}
```

Query Execution

Prepared Statement

In this program, we use one new feature, prepared statements. Consider the query for all books by a particular publisher, independent of the author. The SQL query is:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = the name from the list box
```

Each host variable in a prepared query is indicated with a ?. If there is more than one variable, we must keep track of the positions of the ? when setting the values. For example, our prepared query becomes

```
String publisherQuery =
"SELECT Books.Price, Books.Title" +
```

```
" FROM Books, Publishers" +
" WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name =
?";

PreparedStatement stat = conn.prepareStatement(publisherQuery);
```

Before executing the prepared statement, We must bind the host variables to actual values with a set method. Here, we want to set a string to a publisher name.

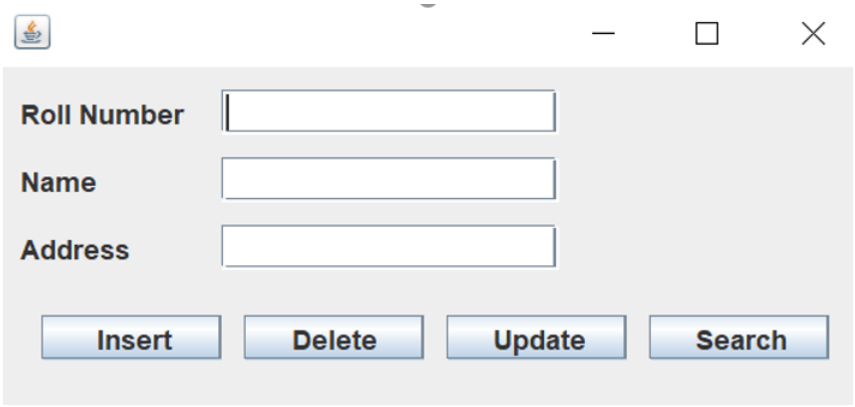
```
stat.setString(1, publisher);
```

The first argument is the position number of the host variable that we want to set. The position 1 denotes the first ?. The second argument is the value that we want to assign to the host variable.

Once all variables have been bound to values, you can execute the prepared statement:

```
ResultSet rs = stat.executeQuery();
int r = stat.executeUpdate();
System.out.println(r + " rows updated");

UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

Demo




```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.sql.*;
public class AllOp
{
 AllOp()
 {
      JFrame f=new JFrame();

      JLabel lblRollno=new JLabel("Roll Number");
      lblRollno.setBounds(10, 10, 150, 20);
      f.add(lblRollno);

      JTextField txtRollno=new JTextField();
      txtRollno.setBounds(100, 10, 150, 20);
      f.add(txtRollno);

      JLabel lblName=new JLabel("Name");
      lblName.setBounds(10, 40, 150, 20);
      f.add(lblName);

      JTextField txtName=new JTextField();
      txtName.setBounds(100, 40, 150, 20);
      f.add(txtName);
```

```
        JLabel lblAddress=new JLabel("Address");
        lblAddress.setBounds(10, 70, 150, 20);
        f.add(lblAddress);

        JTextField txtAddress=new JTextField();
        txtAddress.setBounds(100, 70, 150, 20);
        f.add(txtAddress);

        JButton btnInsert=new JButton("Insert");
        btnInsert.setBounds(20, 110, 80, 20);
        f.add(btnInsert);

        JButton btnDelete=new JButton("Delete");
        btnDelete.setBounds(110, 110, 80, 20);
        f.add(btnDelete);

        JButton btnUpdate=new JButton("Update");
        btnUpdate.setBounds(200, 110, 80, 20);
        f.add(btnUpdate);

        JButton btnSearch=new JButton("Search");
        btnSearch.setBounds(290, 110, 80, 20);
        f.add(btnSearch);

        btnInsert.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                try
                {
                Class.forName("com.mysql.cj.jdbc.Driver");
                Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3307/test"
,"root","");
                String sql="insert into tblStd
(ID,RollNo,Name,Address) values(NULL,?,?,?)";
                PreparedStatement ptst=conn.prepareStatement(sql);
                ptst.setInt(1, Integer.parseInt(txtRollno.getText()));
                ptst.setString(2, txtName.getText());
                ptst.setString(3, txtAddress.getText());
                ptst.executeUpdate();
                if(ptst.getUpdateCount()>0)
                JOptionPane.showMessageDialog(null,"Data inserted
Successfully");
                conn.close();
                }
                catch(Exception e)
                {
```

```
                JOptionPane.showMessageDialog(null, ae);
            }
        }
    }
    );

    btnDelete.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            try
            {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3307/test"
,"root","");
             String sql="delete from tblstd where RollNo=?";
            PreparedStatement ptst=conn.prepareStatement(sql);
            ptst.setInt(1, Integer.parseInt(txtRollno.getText()));
             ptst.executeUpdate();
             if(ptst.getUpdateCount()>0)
             JOptionPane.showMessageDialog(null,"Data deleted
Successfully");
            else
            JOptionPane.showMessageDialog(null,"Data not Found");
            conn.close();
            }
            catch(Exception e)
            {
                JOptionPane.showMessageDialog(null, ae);
            }
        }
    }
    );

    btnUpdate.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            try
            {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3307/test"
,"root","");
            String sql="update tblstd set
RollNo=?,Name=?,Address=? where RollNo=?";
            PreparedStatement ptst=conn.prepareStatement(sql);
            ptst.setInt(1, Integer.parseInt(txtRollno.getText()));
```

```
                ptst.setString(2, txtName.getText());
                ptst.setString(3, txtAddress.getText());
                ptst.setInt(4, Integer.parseInt(txtRollno.getText()));
                ptst.executeUpdate();
                if(ptst.getUpdateCount()>0)
                JOptionPane.showMessageDialog(null,"Data Updated
Successfully");
                else
                 JOptionPane.showMessageDialog(null,"Data not found");
                conn.close();
                }
                catch(Exception e)
                {
                    JOptionPane.showMessageDialog(null, ae);
                }
            }
        }
        );

        btnSearch.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                try
                {
                Class.forName("com.mysql.cj.jdbc.Driver");
                Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3307/test"
,"root","");
                String sql="select * from tblstd where RollNo=?";
                PreparedStatement ptst=conn.prepareStatement(sql);
                ptst.setInt(1, Integer.parseInt(txtRollno.getText()));
                ResultSet rs=ptst.executeQuery();
                if(rs.next())
                {
                txtName.setText(rs.getString("Name"));
                txtAddress.setText(rs.getString("Address"));
                }
                else
                {
                    JOptionPane.showMessageDialog(null,"Data not
Found");
                }
                conn.close();
                }
                catch(Exception e)
                {
                    JOptionPane.showMessageDialog(null, ae);
                }
```

```
            }
        }
        );

        f.setSize(400,500);
        f.setLayout(null);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        new AllOp();
    }
}
```

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2019

Bachelor in Computer Applications **Full marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time: 3 hours**
Semester: III

Centre: **Symbol No:**

Candidates are required to answer the questions in their own words as far as possible.

Group A

Attempt all the questions. **[10×1=10]**

1. Circle (◯) the correct answer in the following questions.

i. Which one of the following is not a valid java bitwise operator?
   a) >> b) <<
   c) >>> **d) <<<**

ii. Which one of the following keyword is used to declare an exception?
   a) throws b) throw
   c) try d) catch

iii. Which of these is an incorrect array declaration?
   a) int ary[]=new int[5]; b) int[]=new int[5];
   c) int ary=int[5] new; d) int ary[]; ary=new int[5];

iv. Which one of the following access specified is appropriate for members of super class to access only from subclass?
   a) private **b) protected**
   c) public d) default

v. Which one of the following is not a collection class defined in java?
   a) Linked list b) Hash set
   c) Tree set **d) Graph set**

vi. Which one of the following inheritance is being implanted using interface in java?
   a) Single inheritance b) Multi-level inheritance
   c) Multiple inheritance d) Hierarchical inheritance

vii. Which one of the following method is called only once during the run time of your applet?
   a) stop() b) paint)
   c) int() d) start()

viii. which of these method of class String is used to compare two string objects for their equality?
   a) equals() b) Equals()
   c) isEqual() d) IsEqual()

ix. What is the default value of priority variable MIN-PRIORITY and MAX-PRIORITY?
   a) 0 & 63 **b) 1 & 10**
   c) 0 & 1 d) 1 & 32

x. Which one of the following is not java swing container?
   a) panel b) Tabbed pane
   c) Scroll pane **d) Scroll bar**

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2019

Bachelor in Computer Applications **Full Marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time : 3 hours**
Semester: III

Candidates are required to answer the questions in their own words as far as possible.

Group B

Attempt any SIX questions. **[6×5=30]**

2. Define OOP. Explain features of Object Oriented Programming Language. [1+4]
3. Explain different types of control statements used in java [5]
4. Define Abstract Class. Explain different types of Access controls available in java. [1+4]
5. Define method overriding? Write any program to implement concept of multiple inheritance in java. [1+4]
6. Why it is important to handle exception in java? Write a program to illustrate the use of exception handling. [1+4]
7. Define the use of static keyword. Write any four String methods used in java with example. [1+4]
8. Define super, final and this keyword in java. Explain the concept of MVC in brief. [1+4]

Group C

Attempt any TWO questions. **[2×10= 20]**

9. a) Define multithreading. Write a java program to show the inter-thread communication. [1+4]
   b) Define Stream. Write a program in java to copy the content from one file to another. [1+4]

10. a) Define Collection Class. Explain different Wrapper classes and associated methods in java. [1+4]
    b) Define AWT. Explain different types of Layout Managers in java. [1+4]

11. a) List and explain any five swing controls with their uses. [5]
    b) Define JDBC. Write a program to display all records from a table of database. [1+4]

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2020

Bachelor in Computer Applications **Full Marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time : 3 hours**
Semester: III

Candidates are required to answer the questions in their own words as far as possible.

Group B

Attempt any SIX questions. **[6×5=30]**

2. Define OOP. Write the characteristics of OOP language. [1+4]
3. Explain the operators available in Java programming. [5]
4. Define loop. Write a java program to print first n prime numbers. [1+4]
5. Differentiate between abstract class and interface with suitable example. [5]
6. Define access modifier. Explain access modifiers in java with example. [1+4]
7. Define exception. Explain exception handling mechanism in java with example. [1+4]
8. Write short note on (Any Two): [2.5+2.5]
   a) final keyword b) Collection class c) JDBC

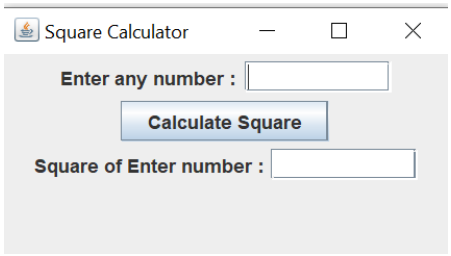
Group C

Attempt any TWO questions. **[2×10= 20]**

9. a) Write a program to create and use java package. [5]
   b) Define thread. Explain the life cycle of thread. [1+4]

10. a) Write a program to sort name of any five cities in ascending order. [5]
    b) Define polymorphism. How do we achieve polymorphism in java explain with example? [1+4]

11. a) Differentiate between java AWT and java Swing. Explain the different types of layout managers in java GUI programming. [2+3]
    b) Write a java GUI program to calculate square of entered number. [5]



Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2021

Bachelor in Computer Applications **Full marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time: 3 hours**
Semester: III

Symbol No : **Center:**

Candidates are required to answer the questions in their own words as far as possible.

Group A

Attempt all the questions. **[10×1=10]**

1. Circle (◯) the correct answer in the following questions.

i. Which one of the following is a ternary operator?
   a) >> b) >>>
   c) += **d) ?:**

ii. Which keyword is used to make a variable constant in Java?
   a) super b) constant
   c) global **d) final**

iii. Which of the following is not a keyword in Java?
   a) boolean b) default
   c) this **d) implicit**

iv. Which of these operators can skip evaluating right hand operand?
   a) ^ b) |
   c) & **d) &&**

v. Which of these keywords is not a part of exception handling?
   a) try b) finally
   c) threw d) catch

vi. The method that returns a string associated with a GUI Button is ……
   a) getSource() method b) setSource() method
   c) getActionCommand() method d) setActionCommand() method

vii. Which one of the following is the container that doesn't contain title bar and menu bar but it can have other GUI components such as buttons and textfields?
   a) Window b) Frame
   c) Panel d) Menu

viii. Which of the following classes is defined in javax.swing package?
   a) Button **b) JButton**
   c) Applet d) Frame

ix. Which method is automatically called after the browser calls the init method?
   a) start b) stop
   c) destroy d) paint

x. In which of the following package Integer class exist?
   a) java.util b) java.file
   c) java.io **d) java.lang**

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2021

Bachelor in Computer Applications **Full Marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time : 3 hours**
Semester: III

Candidates are required to answer the questions in their own words as far as possible.

Group B

Attempt any SIX questions. **[6×5=30]**

2. Explain features of Object oriented concept in detail. [5]
3. What is a continue statement? Write a program in java that displays the sum of odd integers of an array. [1+4]
4. What is parameterized constructor? Explain with an example. [1+4]
5. Explain method overloading in java with proper example. [5]
6. What is the use of 'this' and 'super' keyword? Explain with examples. [2.5+2.5]
7. Defined exception. How exceptions are handled in java? Explain with a proper example. [1+4]
8. Write short notes on (Any Two): [2.5+2.5]
   a) Interface b) Access modifiers c) JDBC

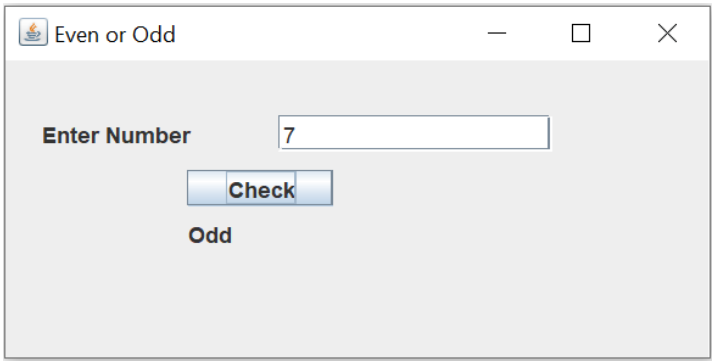
Group C

Attempt any TWO questions. **[2×10= 20]**

9. a) Define inheritance. Explain the use of "extends" keyword in java. [1+4]
   b) What is an abstract class? Explain its use with an example. [1+4]

10. a) List and explain any five String methods used in Java with examples. [5]
    b) Differentiate between byte stream and character stream. Write a program to copy the content of a file to another file using streams. [1+4]

11. a) Create the following GUI form in Java that checks a number entered in the text box is even or odd. [5]



b) Differentiate AWT with swing. Explain any two layout management in Java. [1+4]

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2022

Bachelor in Computer Applications **Full marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time: 3 hours**
Semester: III

Symbol No : **Center:**

Candidates are required to answer the questions in their own words as far as possible.

Group A

Attempt all the questions. **[10×1=10]**

2. Circle (◯) the correct answer in the following questions.

i. Which one of the following is not feature of java
a) Simple b) Secure **c) Pointer** d) Robust

ii. Which function is used to get single character from the scanner?
a) next() **b) next().charAt(0)** c) next().charAt(0) d) nextLine()

iii. Which of the following is not jumping statement in java?
a) goto b) break c) return d) continue

iv. What is process by which we can control which part of program can access the member of class?
a) Polymorphism b) Abstraction **c) Encapsulation** d) Inheritance

v. What is return type of constructor in java?
a) int b) String c) void **d) none**

vi. Which of the following keyword used to refer base class constructor to subclass constructor in java?
a) this b) final **c) super** d) static

vii. Which one of the following keyword is not part of exception handling?
a) thrown b) catch c) try d) finally

viii. Which of the following is not method of Wrapper class?
a) valueOf() **b) indexOf()** c) parseDouble() d) equals()

ix. Which of the following is not adapter class used in event handling?
a) ActionAdapter b) KeyAdapter c) MouseAdapter d) WindowAdapter

x. Which of the following method is used to execute select query in JDBC API?
a) executeQuery() b) executeUpdate()
c) executeNonQuery() d) executeReader()

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2022

Bachelor in Computer Applications **Full Marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time : 3 hours**
Semester: III

Candidates are required to answer the questions in their own words as far as possible.

Group B

Attempt any SIX questions. **[6×5=30]**

2. What is jagged array? Write a java program to initialize and display jagged array element with sum of each row. [1+4]
3. What are uses of this keyword? Write a java program to implement method overloading concept. [2+3]
4. Why inheritance is needed in java? How will you implement multiple inheritance using interface? Explain with program. [1+4]
5. What is multithreading? Explain with program how you will create thread by implementing runnable interface and extending Thread class [1+2+2]
6. What are exception handling keyword in java? Write java program to illustrate the concept of NullPointerException and NumberFormatException [1+2+2]
7. What is serialization? Write a java program to write three student information (Roll, Name, Address, College) into file student.txt and display the student information whose address is Jhapa. [1+4]
8. What is Wrapper class? Explain with program, how will you use iterator and comparator in collection objects? [1+2+2]

Group C

Attempt any TWO questions. **[2×10= 20]**

9. What is difference between class and interface in java? Create a java class **Time** with three attribute **hours, minutes and seconds**. Use appropriate constructors to initialize instance variable. Use methods to display time in **HH:MM:SS** format, add and subtract two time object. Implement the class to add, subtract and display time object. [3+7]
10. How AWT is differing from Swing? Write a GUI program using components to find factorial and cube of number. Use TextField for giving input and Label for output. The program should display factorial if user press mouse on result button and cube if user release mouse from result button. [6+4]
11. What is JDBC? Write a java program to connect database **Company** and insert 5 employee record (EID, Ename, Salary, Department) in **Employee** table and display the employee record who's Department is "Sales". [2+4+4]

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2023

Bachelor in Computer Applications **Full marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time: 3 hours**
Semester: III

Symbol No : **Center:**

Candidates are required to answer the questions in their own words as far as possible.

Group A

Attempt all the questions. **[10×1=10]**

1. Circle (◯) the correct answer in the following questions.

i. Which of the following is not primitive data type defined in Java?
a) Byte b) Long c) Boolean **d) String**

ii. Which of the following is not valid declaration of for loop?
a) for(int a=7;a<=77;a+=7) **b) for(int a=99;a>=0;a/9)**
c) for(int a=2;a<=20;a=2*a) d) for(int a=20;a>=2;--a)

iii. What is process by which one object acquires all the properties and behaviors of another object?
a) Inheritance b) Polymorphism c) Abstraction d) Encapsulation

iv. Which of the following is unchecked exception in java?
a) IOException b) SQLException
c) NullPointerException d) FileNotFoundException

v. Which of the following keyword is used to derive child interface from parent interface?
a) package b) import c) extends **d) implements**

vi. What is process when two or more threads need to access share resource, they need some way to ensure that resource will used by only one thread at a time?
a) Deadlock **b) Synchronization**
c) Interthread communication d) Starvation

vii. Which of the following class is used to read data from file in character Stream?
a) FileInputStream b) FileOutputStream
c) FileReader d) FileWriter

viii. Which of the following is not method of Iterator?
a) hasNext() b) next() c) remove() **d) empty()**

ix. What is name of swing control that is normally display one entry and act as dropdown list?
a) JList **b) JComboBox** c) JRadioButton d) JCheckBox

x. Which of the following JDBC component is used to handle communication with database server?
a) DriverManager b) Driver
c) Connection d) SQLException

Tribhuvan University
Faculty of Humanities and Social Sciences
OFFICE OF THE DEAN
2023

Bachelor in Computer Applications **Full Marks: 60**
Course Title: OOP in Java **Pass Marks: 24**
Code No: CACS 204 **Time : 3 hours**
Semester: III

Candidates are required to answer the questions in their own words as far as possible.

Group B

Attempt any SIX questions. **[6×5=30]**

2. What is java Buzzwords? Write a java program to find simple interest. Use command line argument to take input. [1+4]
3. What are uses of super keyword? Write a java program to create base class **Fruit** which has **name, taste** and **size** as its attributes and method called **eat()** which describe name and its taste. Inherit the same in two other class **Apple** and **Orange** and override the **eat()** method to represent each fruit taste. [1+4]
4. What is difference between String and StringBuffer class? Write a java program to identify the input string is palindrome or not? [2+3]
5. Why we need file handling in java? Write java program to read file into a variable and then write a variable's content into another file. [1+4]
6. What is the purpose of **valueof()** method in Wrapper classes? Write a java program to generate random integer, double and bytes. [1+4]
7. What is internal frames? Write a java program that display two internal frame within some parent frame. [1+4]
8. What is JDBC? Write java program to connect database **College** and display all student information (**Roll, Name, Address** and **Program**) from **Student** table. [1+4]

Group C

Attempt any TWO questions. **[2×10= 20]**

9. What is Constructor overloading? Write a class **Distance** containing private variable **feet** of type int and **inches** of type int, suitable constructors and three methods **addDistance, subtractDistance** and **displayDistance** for adding, subtracting and displaying objects. Write a separate class **MyDistance** containing main method to create, add, subtract and display distance objects. [2+8]
10. a) What is difference between checked and unchecked exception? Write a java program that will read college name from keyboard and display it on screen. The program should throw an exception when length of college name is more than 50. [1+4]
    a) What are methods used for inter-thread communication? Write a java program to create two threads so that one thread prints even number and other thread prints odd number between 100 and 200. [1+4]
11. What is MVC design pattern? Write a GUI program using swing components to calculate sum and difference of two numbers. Use two text fields for input and pre-built dialog box for output. Your program should display sum if Add button and difference if subtract button is clicked. [2+8]