Department of Computer Science & Engineering

Object Oriented Programming through JAVA (AK20)

ANNAMACHARYA INTITUTE OF TECHNOLOGY & SCIENCES:: TIRUPATHI

(Autonomous Institution - UGC, Govt. of India)
(Recognized under 2(f) and 12 (B) of UGC ACT 1956)

(Affiliated to JNTUA, Anantapuramu, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade, Accredited by Institute of Engineers, Kolkata, A-Grade Awarded by AP Knowledge Mission)
Tirupathi, Andhrapradesh - 517 520

Course Material

for

Object Oriented Programming through JAVA (AK20)

Course Code	:	CSE(20APC0512) / CIC(20APC3609)
Regulations	:	AK20
Class	:	II Year II Semester
Branch	:	CSE/CIC

Department of Computer Science & Engineering

ANNAMACHARYA INTITUTE OF TECHNOLOGY & SCIENCES:: TIRUPATHI

(Autonomous Institution - UGC, Govt. of India)

(Recognized under 2(f) and 12 (B) of UGC ACT 1956)

(Affiliated to JNTUA, Anantapuramu, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade, Accredited by Institute of Engineers, Kolkata, A-Grade Awarded by AP Knowledge Mission)

Tirupathi, Andhrapradesh - 517 520

TABLE OF CONTENTS

INSTITUTE VISION & MISSION

Vision

“To Promote Excellence in Technical and Management Education.”

Mission

- **Strengthen the Learning-Teaching Process for Holistic Development.**
- **Upgrade Physical Infrastructure to meet the Curriculum needs.**
- **Enhance Industry-Institute Interactions to acquire Professional Competency.**
- **Promote Innovation and Research to address Challenges of Society.**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

➢ To achieve excellence in the field of Computer Science and Engineering with professional competency.

Mission

M1: Provide quality education to achieve excellence.

M2: Upgrade infrastructure and Technologies to meet the learner’s needs.

M3: Establish linkages with Government and Industry to enhance technical skills, entrepreneurship and innovations.

M4: Support research to serve the needs of the society.

Program Educational Objectives

The department of CSE has developed and adopted Program Educational Objectives (PEO's) for guiding UG programs towards the mission and vision which reflects three aspects of student learning: Cognitive, Affective and Behavioral. PEOs are expected to attain by the students few years after their graduation.

PEO's (Program Educational Objectives)	
PEO1	Graduates will be able to become competent professionals rendering service to IT and ITES industry
PEO2	Graduates will be able to become lifelong learners by adapting new technologies to sustain in their career.
PEO3	Graduates will be able to become technocrats to serve the needs of the society with ethical values

Program Outcomes

PO's (Program Outcomes)	
PO No.	**Program Outcome Description**
PO 1	**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO 2	**Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO 3	**Design / Development of solution:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO 4	**Conduct investigation of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO 6	**The engineer & society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7	**Environment & sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO 8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO 9	**Individual & team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO 10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO 11	**Project management & finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to

	manage projects and in multidisciplinary environments.
PO 12	**Life long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes:

PSO's (Program Specific Outcomes)	
PSO1	Demonstrate the working principles of the hardware and software aspects of computer systems.
PSO2	Design products based on the professional engineering practices with effective strategies

Course Outcomes:

CO's (Course Outcomes)	
CO1	Understanding the Syntax, Semantics and features of Java Programming Language.
CO2	To gain knowledge on Object Oriented Programming concepts.
CO3	Design the method of creating Multi-threading programs and handle exceptions.
CO4	Understanding the concepts of java Collection Framework and Applets.
CO5	to create GUI applications & perform event handling.

List of CO's	**PO no. and keyword**	**Competency Indicator**	**Performance Indicator**
CO1	PO1: Understand Basics of concepts	1.4	1.4.1
CO2	PO1: Demonstrate competence in engineering fundamentals	1.3	1.3.1
CO3	PO3: Demonstrate an ability to generate a diverse set of alternative design solutions	3.2	3.2.1
CO4	PO1:Understand Basics of concepts	1.4	1.4.1
CO5	PO5: ability to identify / create modern engineering tools, techniques and resources	5.1	5.1.2

Course Code	Object Oriented Programming through Java		L	T	P	C
20APC0512			3	0	0	3
Pre-requisite	NIL	Semester	II-I			

Course Objectives:

At the end of the course, the students will be able to:

- To understand object oriented programming concepts, and apply them in solving Problems.
- To introduce the principles of inheritance and polymorphism and implementation of packages and interfaces.
- To learn java's exception handling mechanism, String Handling Methods.
- To introduce the concepts of multithreading and Collection Framework and internet programming using applets.
- To introduce the design of Graphical User Interface swing controls.

Course Outcomes (CO):

- Understanding the Syntax, Semantics and features of Java Programming Language.
- To gain knowledge on Object Oriented Programming concepts.
- Design the method of creating Multi-threading programs and handle exceptions.
- Understanding the concepts of java Collection Framework and Applets.
- Ability to create GUI applications & perform event handling.

UNIT - I		9Hrs

Object Oriented Thinking: History of Java, Java Buzzwords, Overview of OOP CLASSES AND Objects: Classes, Objects, Simple Java Program, Methods, Constructors, this Keyword, Garbage Collection, Data Types, Variables, Arrays, Operators, Control Statements Overloading of Methods and Constructors, Parameter Passing, Recursion, String Class and String handling methods.

UNIT - II		9 Hrs

Inheritance: Inheritance Basics, Using Super, Multilevel Hierarchy, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Using final with Inheritance, Object Class.

Packages: Packages, Access Protection, Importing Packages.

Interfaces: Defining an Interface, Implementing Interface, Applying Interface, Variables in Interfaces, Interfaces can be extended.

UNIT - III		8Hrs

Exception Handling: Exception Handling Fundamentals, Exception Types, Uncaught Exceptions, Using try and catch, Multiple catch Clauses, Nested try Statements, throw, throws, finally, Java's Built in Exceptions, Creating Own Exception Sub Classes.

Input and Output Operations: I/O basics, reading console input, writing console output, the PrintWriter class, reading and writing files, automatically closing a file.

Generic Programming : Generic classes, generic methods, Bounded Types, Restrictions and Limitations.

UNIT - IV		8 Hrs

Multithreading: Java Thread Model, The Main Thread, Thread Life Cycle, Creating Thread and Multiple Threads, isAlive() and join(), Thread Priorities, Synchronization, Inter thread Communication, Suspending, Resuming and Stopping Threads.

Collection Framework: Collection Overview, Collection Interfaces: The Collection Interface, the List Interface, the Queue Interface, Collection Classes: Array List Class, Linked List Class, String Tokenizer, Scanner.

UNIT - V		10Hrs

Applets: Applet Basics, Life Cycle of an Applet, Simple Applet Display Methods, The HTML APPLET tag, Passing Parameters to Applets.

Swing: Introduction to Swing Model-View, Controller design pattern button, layout management, Swing Components.

Textbooks:

1. Herbert Schildt, Java. The complete reference, TMH. 9thEdition, 2014
2. Cay. S. Horstmann and Gary Cornell Core Java 2, Vol 2, Advanced Features, Pearson Education, 7thEdition, 2004

Reference Books:

1. J.Nino and F.A. Hosch, An Introduction to programming and OO design using Java, John Wiley & sons.
2. Y. Daniel Liang, Introduction to Java programming, Pearson Education 6th Edition
3. R.A. Johnson- Thomson, An introduction to Java programming and object oriented application development.
4. P. Radha Krishna, Object Oriented Programming through Java, University Press.

Online Learning Resources:

www.javatpoint.com

	PO no. and keyword	**Competency Indicator**	**Performance Indicator**
CO1	PO1: Understand Basics of concepts	1.4	1.4.1
CO2	PO1: Demonstrate competence in engineering fundamentals	1.3	1.3.1
CO3	PO3: Demonstrate an ability to generate a diverse set of alternative design solutions	3.2	3.2.1
CO4	PO1:Understand Basics of concepts	1.4	1.4.1
CO5	PO5: ability to identify / create modern engineering tools, techniques and resources	5.1	5.1.2

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name :** Object Oriented Programming Through JAVA	**AY:** 2021-2022
Unit1 (Basics, Class, Objects, Methods, Strings)			

Syllabus : Unit-1
Object Oriented Thinking: History of Java, Java Buzzwords, **Overview of OOP Classes and Objects:** Classes, Objects, Simple Java Program, Methods, Constructors, this Keyword, Garbage Collection, Data Types, Variables, Arrays, Operators, Control Statements, Overloading of Methods and Constructors, Parameter Passing, Recursion, String Class and String handling methods.

The History and Evolution of Java:

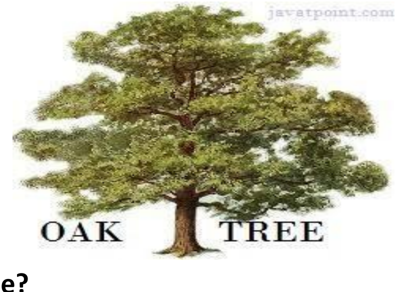
Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology
as incorporated by Netscape.

James Gosling

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1. **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
2. Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3. Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.
4. After that, it was called **Oak** and was developed as a part of the Green project.



Why Oak name for java language?

5. **Why Oak?** Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6. In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java name for java language?

7. **Why they choosed java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted

something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.

8. Java is an island of Indonesia where first coffee was produced (called java coffee).
9. Notice that Java is just a name not an acronym.
10. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
11. In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
12. JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

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

Java Lineage:

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.

Java is a high-level, third generation programming language, like C, FORTRAN, Smalltalk, Perl, and many others. You can use Java to write computer applications that play games, store data or do any of the thousands of other things computer software can do. Java is a programming language created by **James Gosling** from Sun Microsystems (Sun) in 1991.

The first publicly available version of Java (Java 1.0) was released in 1995. Java is most similar to C. However although Java shares much of C's syntax.The concepts are related to c++.

Birth of C language:

The C Programming Language by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted .The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs.

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed.

The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++.

.As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The Stage Is Set for Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible.

It is intended to let application developers **"write once, run anywhere" (WORA),** meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of computer architecture.

As of 2016, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers.

Java was originally developed by **James Gosling at Sun Microsystems** (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low- level facilities than either of them.

Following are the differences Between C and C++ :

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.

2. No virtual Functions are present in C	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used in C++. Polymorphism is the most Important Feature of OOPS.
4. Operator overloading is not possible in C.	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global varioables are not allowed.
8. In C • scanf() Function used for Input. • printf() Function used for output.	8. In C++ • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other Functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	13.In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception Handling is not present.	15. In C++, Exception Handling is done with Try and Catch block.

Difference between c++& java

C++	Java
C++ is not a purely object-oriented programming, since it is possible to write C++ programs without using a class or an object.	Java is purely an object-oriented programming language, since it is not possible to write a Java program without using at least one class.
Pointers are available in C++.	We cannot create and use pointers in Java.
Allocating memory and de-allocating memory is the responsibility of the programmer.	Allocation and de-allocation of memory will be taken care of by JVM.
C++ has **goto** statement.	Java does not have **goto** statement.
Automatic casting is available in C++.	In some cases, implicit casting is available. But it is advisable that the programmer should use casting wherever possible.
Operator overloading is available in C++.	It is not available in Java.
#define, typedef and header files are available in C++.	#define, typedef and header are not available in Java, but there are means to achieve them.
There are 3 access specifiers in C++: private, public and protected.	Java supports 4 access specifiers: private, public, protected, and default.
There are constructors and destructors in C++.	Only constructors are there in Java. No destructors are available in this language.

The Creation of Java:

James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan conceived Java 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 Payane, 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.

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 of 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.

- Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (plat form - independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems.
- Java derives much of its character from C and C++. This is by intent. The java designer 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.

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 also 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.

Computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted java was the need **for platform-independent programs** destined for distribution on the **Internet**. However, java also embodies changes in the way that people approach the writing of programs. Thus, java is not a language that exists in isolation. Rather, it is the current instance of an ongoing process begun many years ago. This fact alone is enough to ensure java a place in computer language history. Java is to Internet programming what C was to systems programming: a revolutionary force that will change the world.

The c# connection:

Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore .Perhaps the most important example of Java's influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java.

What's Similar Between C# and Java?

C# and Java are actually quite similar, from an application developer's perspective. The major similarities of these languages are

1. All Objects are References
2. Garbage Collection
3. Both C# and Java are Type-Safe Languages
4. Both C# and Java Are "Pure" Object-Oriented Languages
5. Single Inheritance
6. Built-in Thread and Synchronization Support
7. Formal Exception Handling

8. Built-in Unicode Support

What's Different Between C# and Java?

While C# and Java are similar in many ways, they also have some differences.

1. Formal Exception Handling
2. Java Will Run on "Any" Operating System
3. C# and Java Language Interoperability
4. C# Is a More Complex Language than Java
5. C# and Java Keyword Comparison
6. Operator
7. Delegate
8. Synchronized
9. Threading and synchronization

<u>How java changed the internet</u>

The Internet helped JAVA to the forefront of programming and JAVA in turn had a great impact on the Internet. While simplifying the web programming 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 main issues associated with the Internet like portability and security.

1.<u>Java Applets</u>

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 <u>web browser</u>. 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 <u>client computer</u>, yet it is initiated by the server. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems.

<u>2,Security :</u> As we know when we download a normal program we are taking a risk, because the code that we are downloading might contain a virus of any other harmful code. At the core of the problem is the fact that malicious code can cause its damage as it has gained unauthorized access to system resources. 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 its access to other parts of the computer.

3.Portability : It is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. Some means of generating portable executable code was needed.

Java's magic: The byte code:

The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a set of instructions designed to be executed by the Java run-time system known as *Java Virtual Machine* (JVM). That is, in its standard form, the JVM is an *interpreter for bytecode.*
Translating a Java program into bytecode helps makes it easier to run a program in a wide variety of environments.

The reason is straightforward: 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. Although the details of JVM will differ from platform to platform, all interpret the same Java bytecode.

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

Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release.

When JIT compiler is part of the JVM, it compiles bytecode into the executable code in real time, on a piece-by-piece, demand basis. It is not possible to compile an entire Java program into executable code all at once, because Java performs various run time checks that can be done only at run time. Instead, the JIT compiles code as it is needed, during execution.

Servlets: java on the server side

Java would also be useful on the server side. The result was the *servlet*. 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.
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.

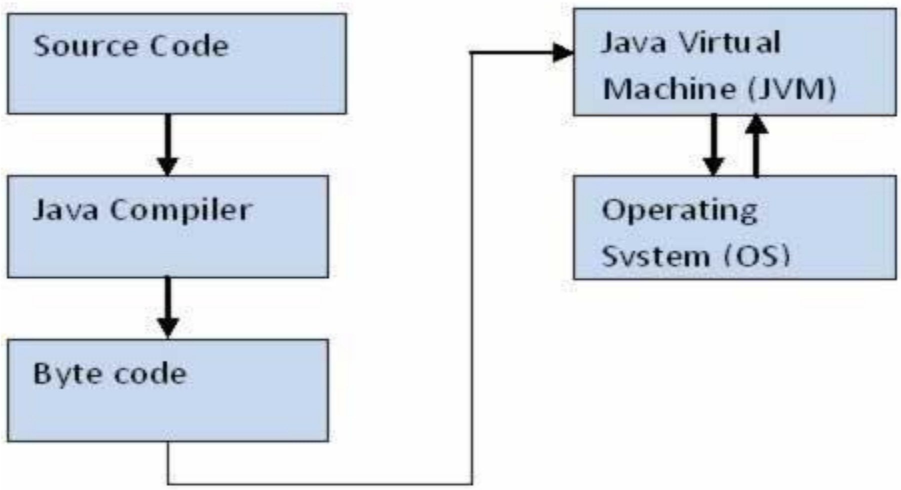
Q. Explain about java buzz words

java Buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

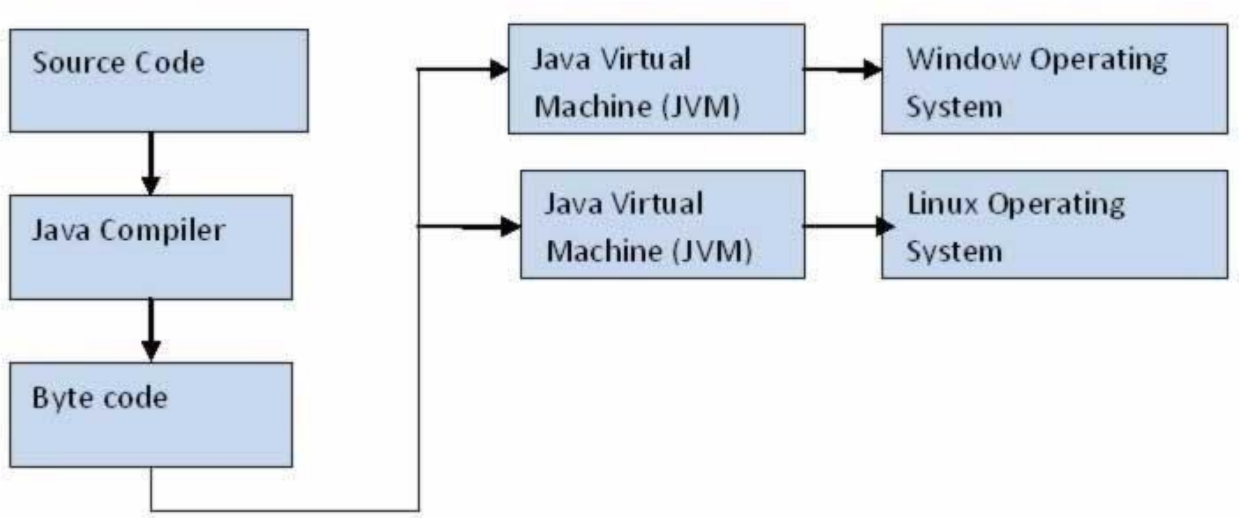
1. **Compiled and Interpreter:- Java** has both Compiled and Interpreter Feature Program of java is First Compiled and Then it is must to Interpret it .First of all The Program of java is Compiled then after Compilation it creates Bytes Codes rather than Machine Language.

Then After Bytes Codes are Converted into the Machine Language is Converted into the Machine Language with the help of the Interpreter So For Executing the java Program First of all it is necessary to Compile it then it must be Interpreter .



2).Platform Independent:- Java Language is Platform Independent means program of java is Easily transferable because after Compilation of java program bytes code will be created then we have to just transfer the Code of Byte Code to another Computer.

This is not necessary for computers having same Operating System in which the code of the java is Created and Executed After Compilation of the Java Program We easily Convert the Program of the java top the another Computer for Execution.



3. Object-Oriented:- We Know that is purely OOP Language that is all the Code of the java Language is Written into the classes and Objects So For This feature java is Most Popular Language because it also Supports Code Reusability, Maintainability etc.

4. Robust and Secure:- The Code of java is Robust andMeans ot first checks the reliability of the code before Execution When We trying to Convert the Higher data type into the Lower Then it Checks the Demotion of the Code the It Will Warns a User to Not to do this So it is called as Robust
Secure : When We convert the Code from One Machine to Another the First Check the Code either it is Effected by the Virus or not or it Checks the Safety of the Code if code contains the Virus then it will never Executed that code on to the Machine.

5. Distributed:- Java is Distributed Language Means because the program of java is compiled onto one machine can be easily transferred to machine and Executes them on another machine because facility of Bytes Codes So java is Specially designed For Internet Users which uses the Remote Computers For Executing their Programs on local machine after transferring the Programs from Remote Computers or either from the internet.

6. Simple Small and Familiar:- is a simple Language Because it contains many features of other Languages like c and C++ and Java Removes Complexity because it doesn't use pointers, Storage Classes and Go to Statements and java Doesn't support Multiple Inheritance

7. Multithreaded and Interactive:- Java uses Multithreaded Techniques For Execution Means Like in other in Structure Languages Code is Divided into the Small Parts Like These Code of java is divided into the Smaller parts those are Executed by java in Sequence and Timing Manner this is Called as

Multithreaded In this Program of java is divided into the Small parts those are Executed by Compiler of java itself Java is Called as Interactive because Code of java Supports Also CUI and Also GUI Programs

8. Dynamic and Extensible Code:- Java has Dynamic and Extensible Code Means With the Help of OOPS java Provides Inheritance and With the Help of Inheritance we Reuse the Code that is Pre-defined and Also uses all the built in Functions of java and Classes

9. Secure: Java was designed with security in mind. As Java is intended to be used in networked/distributor environments so it implements several security mechanisms to protect you against malicious code that might try to invade your file system.

For example: The absence of pointers in Java makes it impossible for applications to gain access to memory locations without proper authorization as memory allocation and referencing model is completely opaque to the programmer and controlled entirely by the underlying run- time platform .

10. Architectural Neutral: One of the key feature of Java that makes it different from other programming languages is architectural neutral (or platform independent). This means that the programs written on one platform can run on any other platform without having to rewrite or recompile them. In other words, it follows 'Write-once-run-anywhere' approach.

Java programs are compiled into byte-code format which does not depend on any machine architecture but can be easily translated into a specific machine by a Java Virtual Machine (JVM) for that machine. This is a significant advantage when developing applets or applications that are downloaded from the Internet and are needed to run on different systems.

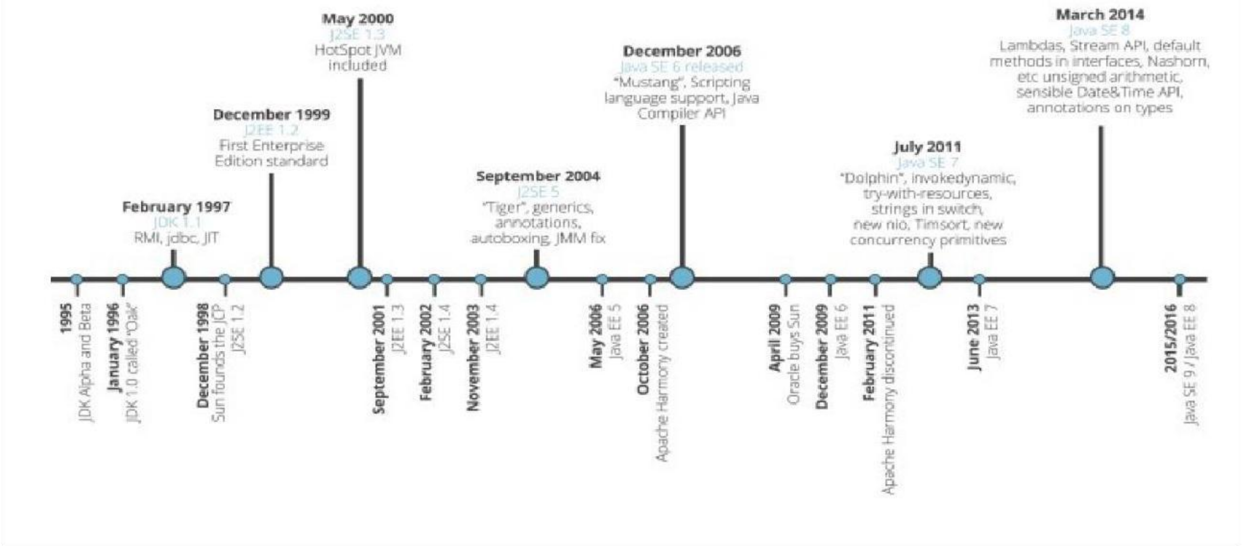
11. Portable : The portability actually comes from architecture-neutrality. In C/C++, source code may run slightly differently on different hardware platforms because of how these platforms implement arithmetic operations. In Java, it has been simplified.

Unlike C/C++, in Java the size of the primitive data types are machine independent. For example, an int in Java is always a 32-bit integer, and float is always a 32-bit IEEE 754 floating point number. These consistencies make Java programs portable among different platforms such as Windows, Unix and Mac .

12. Interpreted : Unlike most of the programming languages which are either complied or interpreted, Java is both complied and interpreted The Java compiler translates a java source file to bytecodes and the Java interpreter executes the translated byte codes directly on the system that implements the Java Virtual Machine. These two steps of compilation and interpretation allow extensive code checking and improved security

13. High performance: Java programs are complied to portable intermediate form know as bytecodes, rather than to native machine level instructions and JVM executes Java bytecode on. Any machine on which it is installed. This architecture means that Java programs are faster than program or scripts written in purely interpreted languages but slower than C and C++ programsthat compiled to native machine languages.

1.2.4 The Evolution of Java:



Introduction:

Introduction to Object Oriented Programming

Languages like Pascal, C, FORTRAN, and COBOL are called procedure oriented Programming languages since in these languages, a programmer uses procedures or functions to perform a task When the programmer wants to write a program, he will first divide the task into separate sub tasks, each of which is expressed as functions/ procedures This approach is called procedure oriented approach.

The languages like C++ and Java use classes and object in their programs and are called Object Oriented Programming languages the main task is divided into several modules and these are represented as classes. Each class can perform some tasks for which several methods are written in a class This approach is called Object Oriented approach.

Q. Object Oriented Concepts:/The Key Attributes Of Object Oriented Programming :

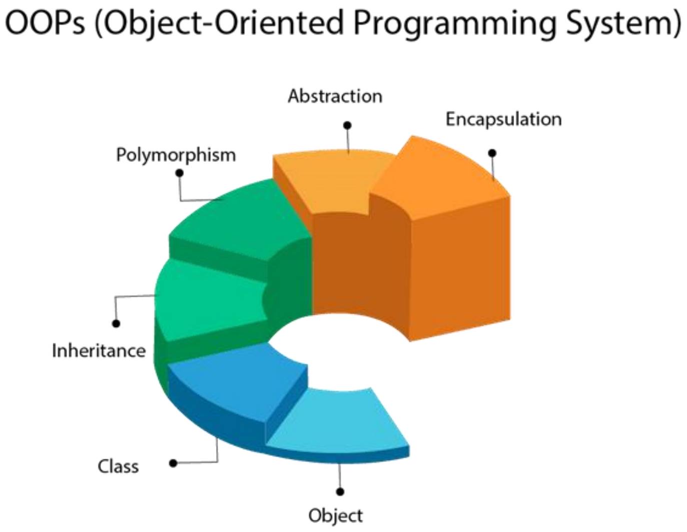
Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. Class

2. **Object**
3. **Abstraction**
4. **Polymorphism**
5. **Inheritance**
6. **Encapsulation**
7. **Dynamic Binding**
8. **Message passing**

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

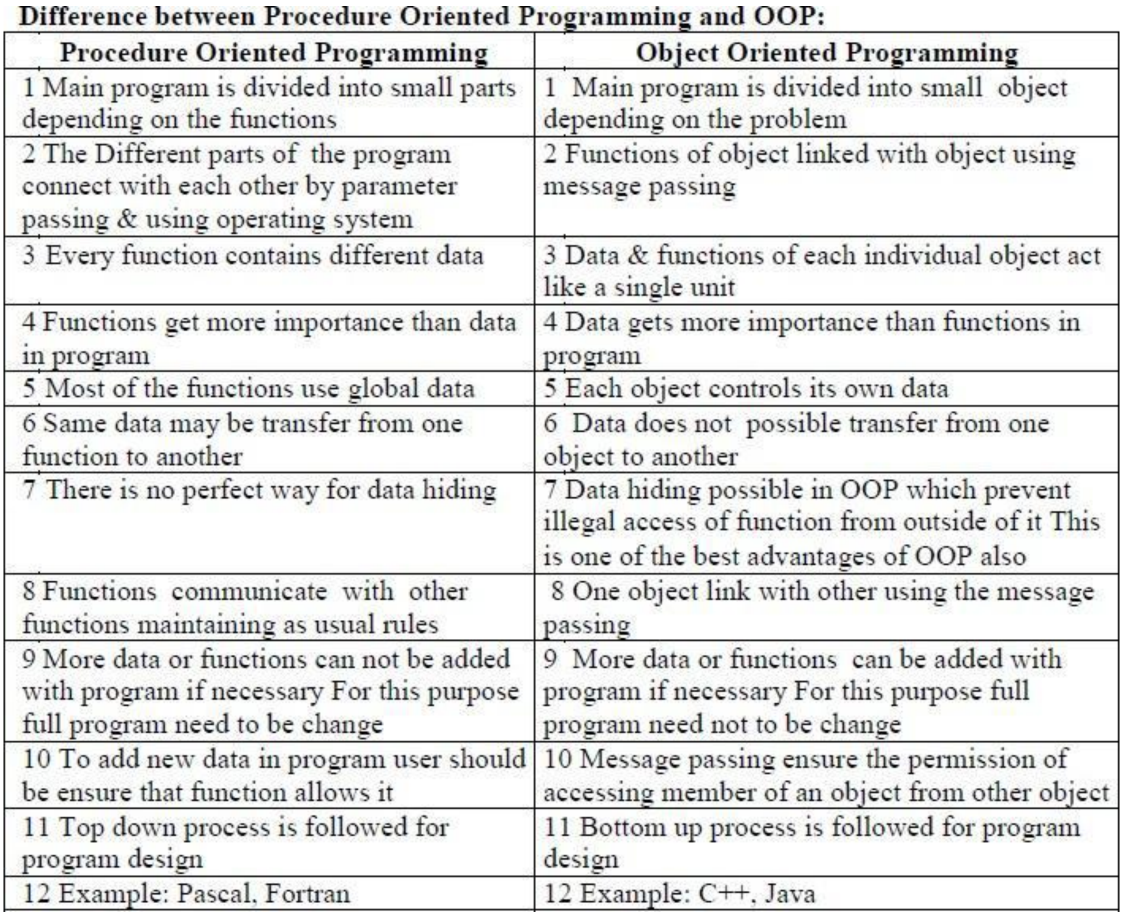
- Coupling
- Cohesion
- Association
- Aggregation
- Composition



1. Object :

1. Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

An Object is a real time entity An object is an instance of a class Instance means physically happening An object will have some properties and it can perform some actions.



Difference between Procedure Oriented Programming and OOP:

Procedure Oriented Programming	Object Oriented Programming
1 Main program is divided into small parts depending on the functions	1 Main program is divided into small object depending on the problem
2 The Different parts of the program connect with each other by parameter passing & using operating system	2 Functions of object linked with object using message passing
3 Every function contains different data	3 Data & functions of each individual object act like a single unit
4 Functions get more importance than data in program	4 Data gets more importance than functions in program
5 Most of the functions use global data	5 Each object controls its own data
6 Same data may be transfer from one function to another	6 Data does not possible transfer from one object to another
7 There is no perfect way for data hiding	7 Data hiding possible in OOP which prevent illegal access of function from outside of it This is one of the best advantages of OOP also
8 Functions communicate with other functions maintaining as usual rules	8 One object link with other using the message passing
9 More data or functions can not be added with program if necessary For this purpose full program need to be change	9 More data or functions can be added with program if necessary For this purpose full program need not to be change
10 To add new data in program user should be ensure that function allows it	10 Message passing ensure the permission of accessing member of an object from other object
11 Top down process is followed for program design	11 Bottom up process is followed for program design
12 Example: Pascal, Fortran	12 Example: C++, Java

We must acquire an actual, physical copy of the object and assign it to that variable We can do this using **new** operator The new operator dynamically allocates memory for an object and returns a reference to it This reference is, more or less, the address in memory of the object allocated by new This reference is then stored in the variable Thus, in Java, all class objects must be dynamically allocated.

2. Class:

In object-oriented programming, a class is a programming language constructs that is used as a blueprint to create objects This blueprint includes attributes and methods that the created objects all share Usually, a class represents a person, place, or thing - it is an abstraction of a concept within a computer program Fundamentally, it encapsulates the state and behavior of that which it conceptually represents It encapsulates state through data placeholders called member variables; it encapsulates behavior through reusable code called methods.

(Or)

Collection of objects is called class. It is a logical entity.

3. Inheritance:

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the IS-A relationship, also known as parent-child relationship.

1.For Method Overriding (so runtime polymorphism can be achieved).
2. For Code Reusability.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical .In java programming, multiple and hybrid inheritance is supported through **interface** only.

Single inheritance is easy to understand. When a class extends another one class only then we call it a single inheritance.

1. Multiple Inheritance

Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

1. Multilevel Inheritance:

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.

2. Hierarchical Inheritance:

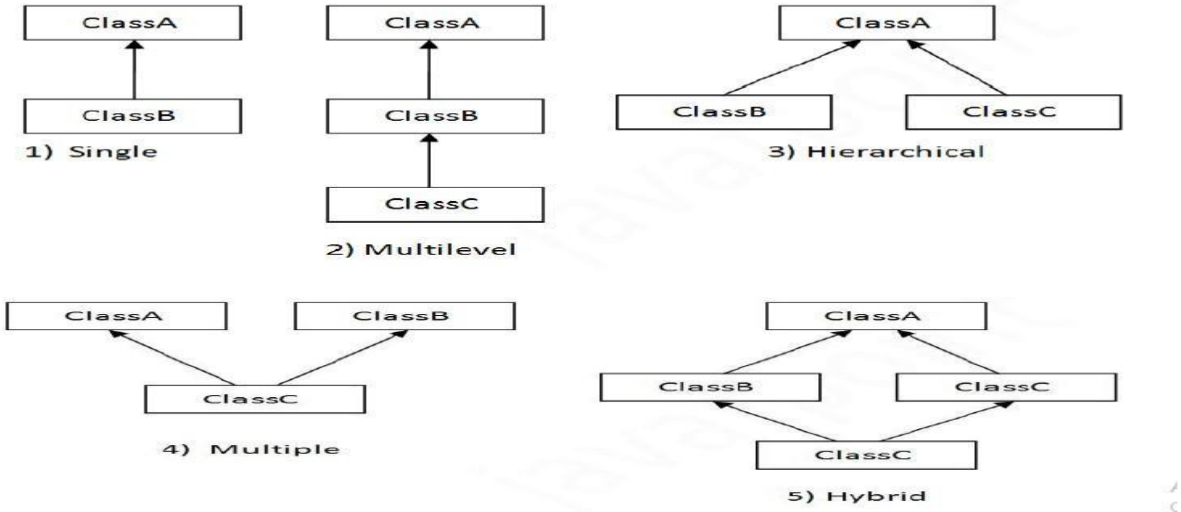
Such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.

Example:

3.Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces.

yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



4. Polymorphism:

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

1. **If you overload static method in java, it is the example of compile time polymorphism. Runtime Polymorphism in Java**

2. **Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.**

4. **Abstraction: It is** a process of hiding the implementation details and showing only functionality to the user.
1. Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
2. Abstraction lets you focus on what the object does instead of how it does it.
3. An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclass.

6. **Encapsulation:**

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

- improves maintainability and flexibility and re-usability

- □ The fields can be made read-only or write-only
- □ User would not be knowing what is going on behind the scene.

7. Dynamic Binding:

Association of method definition to the method call is known as binding. There are two types of binding: Static binding and dynamic binding.

Static Binding

The binding which can be resolved at compile time by compiler is known as static or early binding. All the static, private and final methods have always been bonded at compile-time. It resolves the problems related method overloading.

Dynamic Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method. Thus while calling the overridden method, the compiler gets confused between parent and child class method(since both the methods have same name).

8. Message passing:

Message passing is a method by which an object sends data to another object or requests other object to invoke method. This is also known as interfacing.It acts like a messenger from one object to other object to convey specific instructions. Message Passing involves specifying the name of objects, the name of the function, and the information to be sent.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

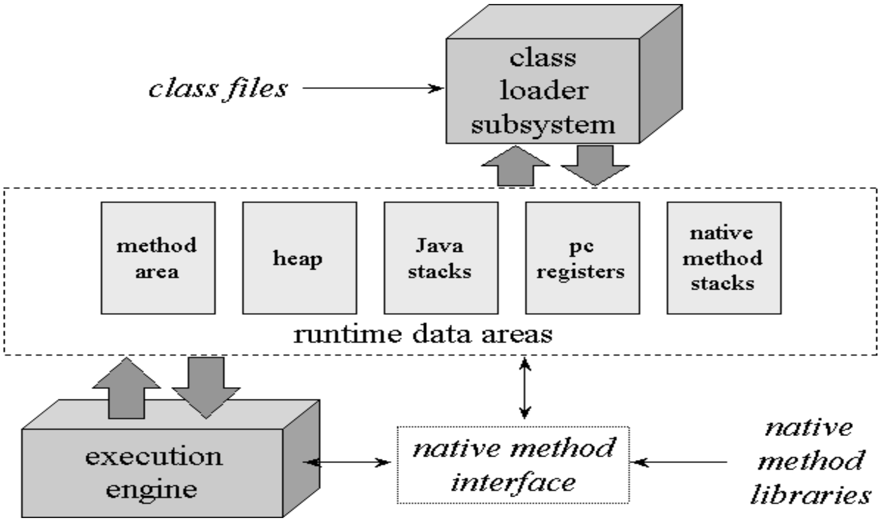
Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

JVM ARCHITECTURE



1. **Class loader sub system**: JVM's class loader sub system performs 3 tasks
   a. It loads .class file into memory.
   b. It verifies byte code instructions.
   c. It allots memory required for the program.

2. **Run time data area**: This is the memory resource used by JVM and it is divided into 5 parts
   a. Method area: Method area stores class code and method code.
   b. Heap: Objects are created on heap.
   c. Java stacks: Java stacks are the places where the Java methods are executed. A Java

stack contains frames. On each frame, a separate method is executed.

d. Program counter registers: The program counter registers store memory address of the instruction to be executed by the micro processor.

e. Native method stacks: The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

3. Native method interface: Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

4. Native method library: holds the native libraries information.

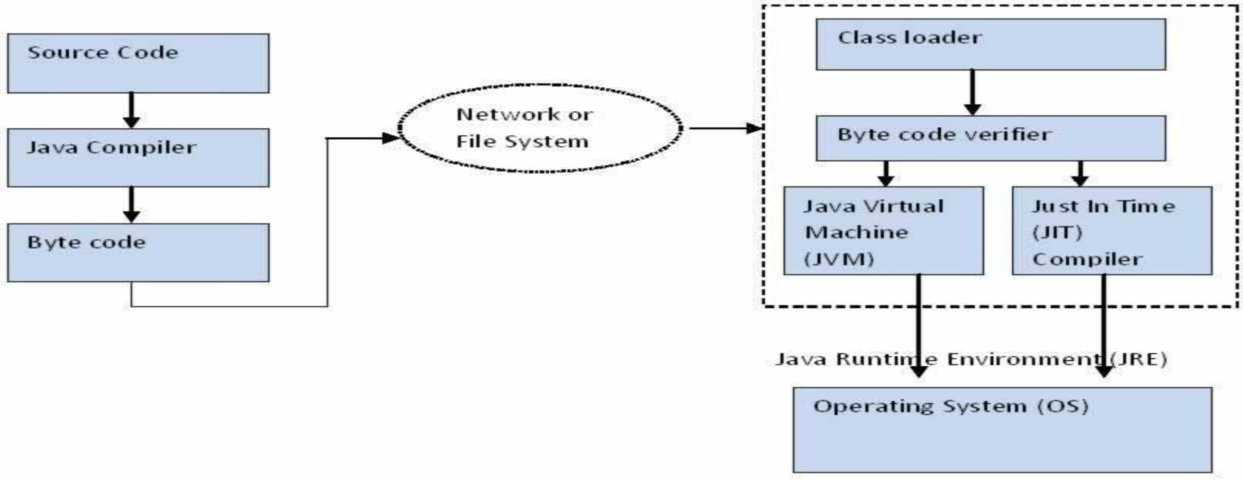
1. **Execution engine:** Execution engine contains interpreter and JIT compiler, which covert byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler.

Java Runtime Environment (JRE)

Java Runtime Environment contains JVM, class libraries and other supporting components. As you know the Java source code is compiled into bytecode by Java compiler. This bytecode will be stored in class files. During runtime, this bytecode will be loaded, verified and JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

1. Loads the class **::**This is done by the class loader
2. Verifies the bytecode **::**This is done by bytecode verifier.
3. Interprets the bytecode **::**This is done by the JVM



Class loader

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the bytecode is loaded successfully, then next step is bytecode verification by bytecode verifier.

Byte code verifier

The bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

1. The code follows JVM specifications.
2. There is no unauthorized access to memory.
3. The code does not cause any stack overflows.
4. There are no illegal data conversions in the code such as float to object references.

Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs.

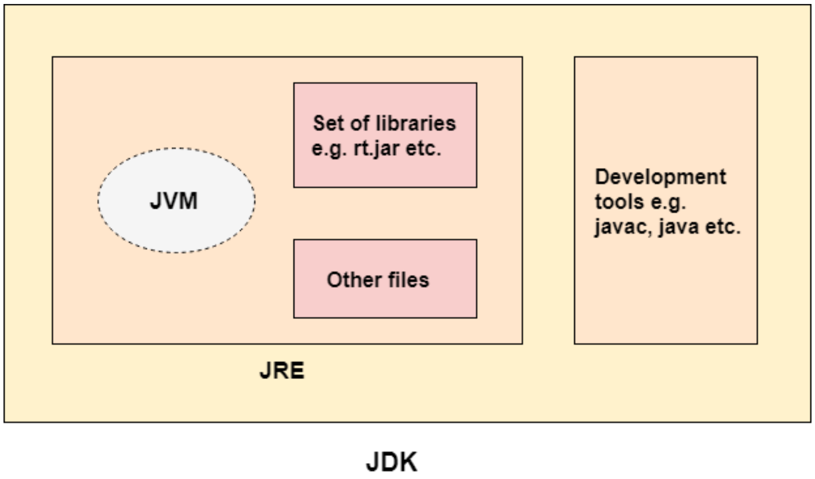
Just in Time Compiler

You might have noticed the component “Just in Time” (JIT) compiler in Figure 3. This is a component which helps the program execution to happen faster.

As we discussed earlier when the Java program is executed, the byte code is interpreted by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster.

If the JIT Compiler library exists, when a particular bytecode is executed first time, JIT complier compiles it into native machine code which can be directly executed by the machine in which the Java program runs.

Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser. This compilation happens when the byte code is about to be executed and hence the name “Just in Time”.



Simple program

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

Requirement for Java Program

- install the JDK if you don't have installed it,
- set path of the jdk/bin directory.

Eg: computer->properties->advanced->environmentvariables->new -> variable name:path variablevalue: C:\Program Files\Java\jdk1.6.0_26\bin below->new->
variable name:classpath

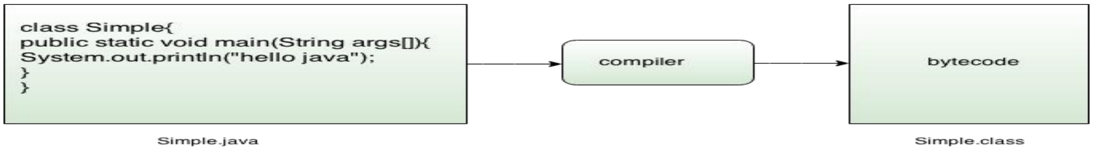
variablevalue: C:\Program Files\Java\jdk1.6.0_26\bin; and then press “ok”
create the java program in **notepad** and save **as “classname.java”**
eg:-Hai.java

compile and run the java program **To compile java program** C:\>javac Hai.java

To run program C:\>java Hai **Program:**

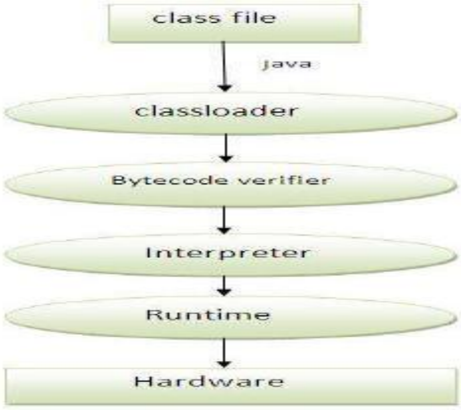
```
class Simple{

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

}
```

Compile Time:

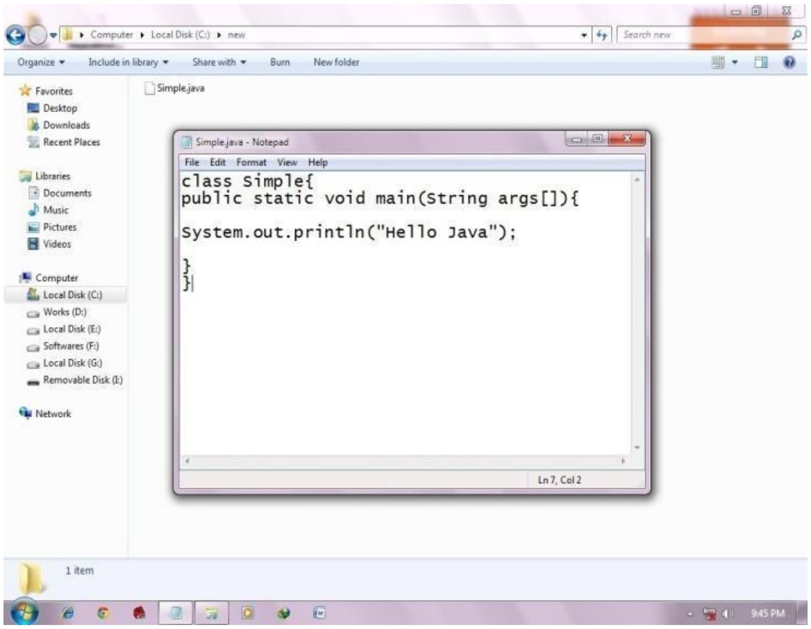


```
class Simple{
public static void main(String args[]){
System.out.println("hello java");
}
}
```

Run Time:



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



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

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

}
}
```

```
C:\
Mic
Cop
C:\
C:\
C:\
C:\
Hel
C:\
```

A Second Short Program

```
/*
Here is another short
example. Call this file
"Example2.java".
*/
class Example2 {
public static void main(String args [] ) {
int num; // this declares a variable called num
num = 100; // this assigns num the value 100
System.out.println("This is num: " + num);
num = num * 2;
System.out.print("The value of num * 2 is ");
System.out.println(num);
}
}
```

When you run this program, you will see the following output:

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

Lexical Issues: Java programs are a collection of whitespace, identifiers, literals,comments, operators, separators, and keywords.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules.

For instance, the **Example** program could have been written all on one
line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Some examples of valid identifiers are

1.AvgTemp 2. count 3.a4 4. $test 5.this_is_ok

Invalid identifier names include these:

1. 2count 2. high-temp 3. Not/ok

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

1.100 2.98.6 3.'X' 4."This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

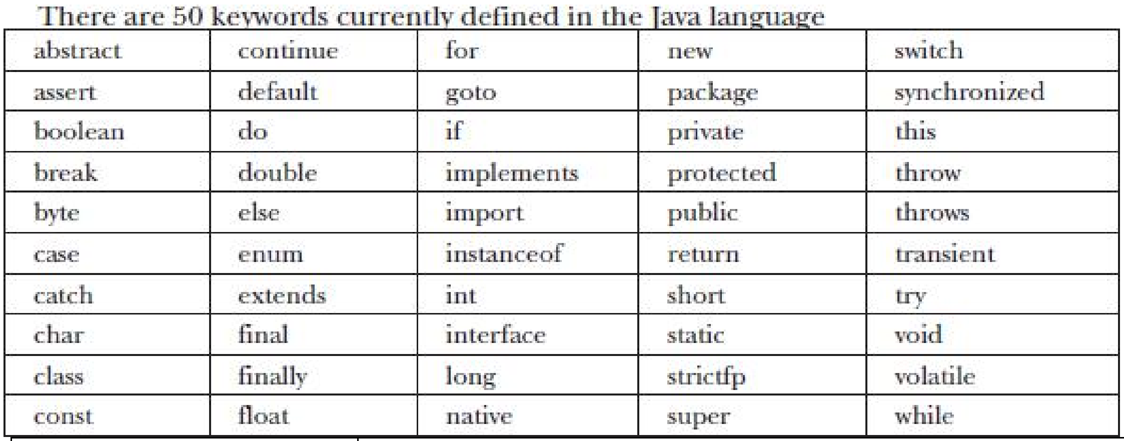
As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type
of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in

Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table.

Java Keywords



There are 50 keywords currently defined in the Java language

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

Access modifiers:	private, protected, public, default
Class, method, variable modifiers:	abstract, class, extends, final, implements, interface, native, new, static, strictfp, synchronized, transient, volatile
Flow control:	break, case, continue, default, do, else, for, if, instance of, return,switch, while
Package control:	import, package
Primitive types:	boolean, char, byte, short, int, long, float, double
Error handling:	assert, catch, finally, throw, throws, try
Enumeration:	Enum
Others:	super, this, void
Unused:	const, goto

Java class libraries:

Two of Java's built-in methods:

println( ) and print( ). As mentioned, these methods are available through System.out. System is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI).

List of Library Classes in Java:

Library classes	Purpose of the class
Java.io	Use for input and output functions.
Java.lang	Use for character and string operation.

Java.awt	Use for windows interface.
Java.util	Use for develop utility programming.
Java.applet	Use for applet.
Java.net	Used for network communication.
Java.math	Used for various mathematical calculations like power, square root etc.

We will come across different library classes, which basically deal with input/output operation.

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

- Single Line Comment
- Multi Line Comment
- Documentation Comment

Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

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

Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

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

Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

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

Introducing Classes

The General Form of a Class

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

A Simple Class

```
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
 class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Declaring Objects

The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let’s look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

Box mybox = new Box();

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object.

Assigning Object Reference Variables

Box b1 = new Box();
Box b2 = b1;

You might think that **b2** is being assigned a reference to a copy of the object referred to by**b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects.

Data Types, Arrays and Variables:

Primitive types: boolean, byte, char, double, float, int, long, short **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.

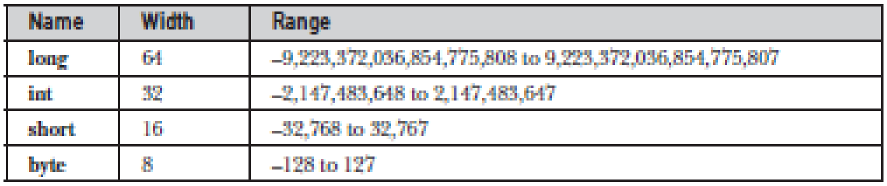
Floating-point numbers This group includes float and double, which represent numbers with fractional precision.

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.

Integers

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.



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.

Ex:

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.

Ex:-int lightspeed;

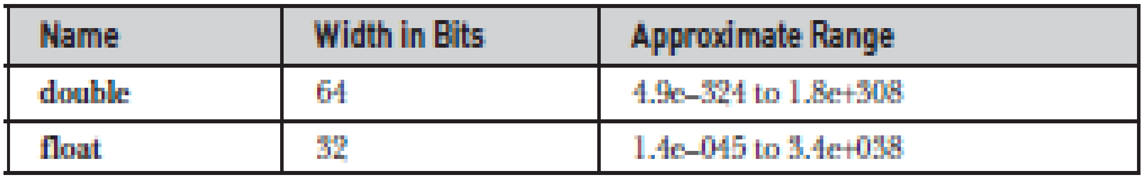
long

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.

Ex:- long distance;

Floating Point:

There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–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.

Ex:-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

ex:-double pi, r, a;

Characters

In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is *not* the case in Java. Instead, 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. 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.

<u>**Ex:-**</u> char ch1, ch2;

Boolean:

<u>**Booleans**</u>

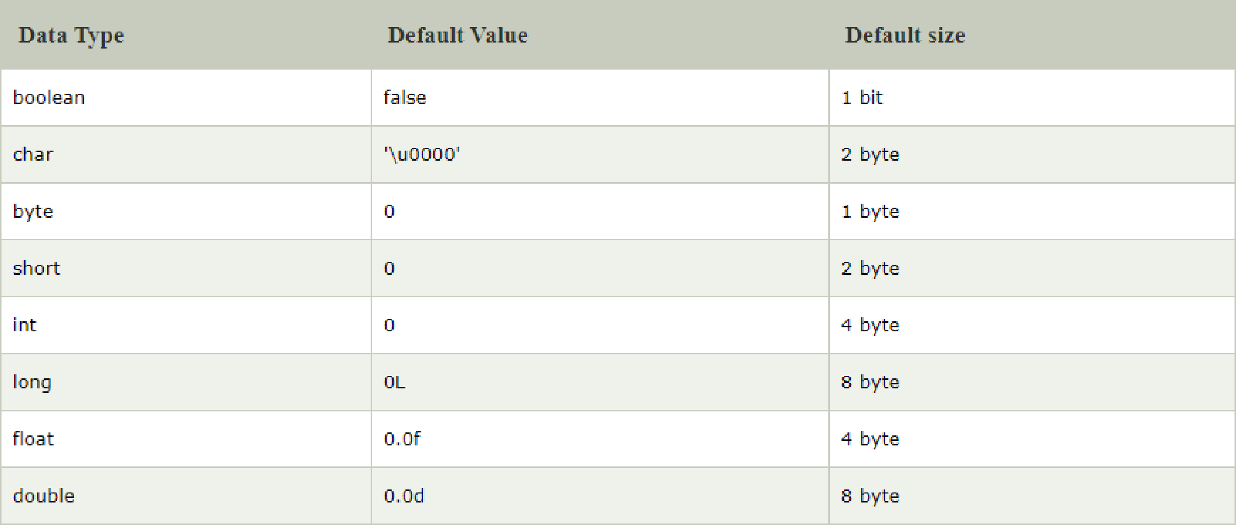
Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**.

Ex-boolean b;

Example:

```
public class PrimitiveDemo {
public static void main(String[] args) {
byte b =100;
short s =123;
int v = 123543;
int calc = -9876345;
long amountVal = 1234567891;
float intrestRate = 12.25f;
double sineVal = 12345.234d;
 boolean flag = true;
boolean val = false;
char ch1 = 88; // code for X
char ch2 = 'Y';
System.out.println("byte Value = "+ b);
System.out.println("short Value = "+ s);
System.out.println("int Value = "+ v);
System.out.println("int second Value = "+ calc);
System.out.println("long Value = "+ amountVal);
System.out.println("float Value = "+ intrestRate);
System.out.println("double Value = "+ sineVal);
System.out.println("boolean Value = "+ flag);
System.out.println("boolean Value = "+ val);
System.out.println("char Value = "+ ch1);
System.out.println("char Value = "+ ch2);
}
}
```

Summary of Data Types:



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initialize.

Declaring a Variable

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

type is one of Java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value.

int a, b, c; // declares three ints, a, b, and c.

```
int d = 3, e, f = 5; // declares three more ints, initializing d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization

class DynInit {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

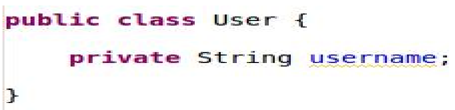
Q. Explain scope of variable

Scope of the Variable:

Scope and Lifetime of **Variables**. The **scope** of a **variable** defines the section of the code in which the **variable** is visible. As a general rule, **variables** that are defined within a block are not accessible outside that block. The lifetime of a **variable** refers to how long the **variable** exists before it is destroyed.

Class Level Scope

In Java, there are some variables that you want to be able to access from anywhere within a Java class. The scope of these variables will need to be at the class level, and there is only one way to create variables at that level – just inside the class but outside of any methods. Let's take a look at an example:

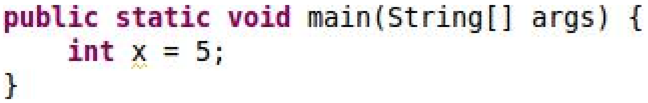


```
public class User {
    private String username;
}
```

You will notice that the variables have been defined at the top of the class, before any methods. This is simply a convention; you can define your class-level variables anywhere in the class (so long as it's outside of any methods in the class). However, it is good practice to put these variables at the top so it's easier to see where they all are.

Method Scope

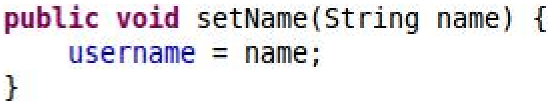
Some variables you might want to make temporary and preferably they are used for only one method. This would be an example of method scope. Here's a pretty typical example of a variable in method scope using an example of Java's main method:



```
public static void main(String[] args) {
    int x = 5;
}
```

The variable x in the example is created inside the method. When the method ends, the variable reference goes away and there is no way to access that variable any longer. You cannot use that same variable x in another method because it only exists in the main method's scope and that is it.

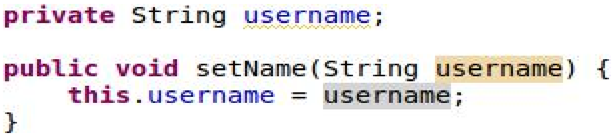
Here's another example of method scope, except this time the variable got passed in as a



```
public void setName(String name) {
    username = name;
}
```

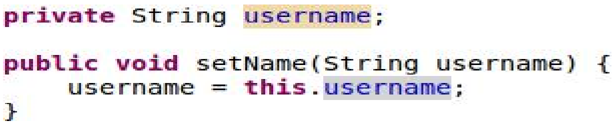
parameter to the method:

The above example is the typical example of a setter method. The purpose of a setter method as you might recall is to set a class variable to a particular value from somewhere outside of the class. The above example is a pretty clean example of this. Now let's look at the same example, except now we'll use a conflicting variable name.



```
private String username;

public void setName(String username) {
    this.username = username;
}
```

So, what's going on with the variable scope here? And what is "this"? The this keyword helps Java to differentiate between the local scope variable (the method scope variable) and the class variable. This. username tells Java that you are referencing the class variable. So, the setter above sets the method-scope variable called name to the class variable called name. The variable scope here is not in conflict because Java knows which variable to access because of the "this" keyword. You could also do this:

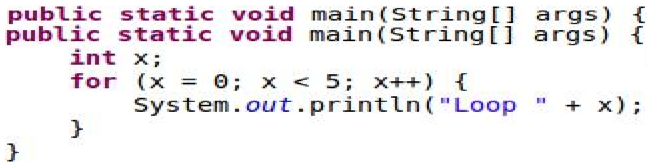


```
private String username;

public void setName(String username) {
    username = this.username;
}
```

However the above is something you probably won't be doing often (if at all!) because you're setting the local, temporary variable to the value of the more permanent one.

Loop Scope

Any variables created inside of a loop are **LOCAL TO THE LOOP**. This means that once you exit the loop, the variable can no longer be accessed! This includes any variables created in the loop signature. Take a look at the following two for loop examples:



```
public static void main(String[] args) {
public static void main(String[] args) {
    int x;
    for (x = 0; x < 5; x++) {
        System.out.println("Loop " + x);
    }
}
```

In the first example, x can ONLY be used inside of the for loop. In the second example, you are free to use x inside of the loop as well as outside of the loop because it was declared outside of the loop (it has been declared at method scope).

Q. Explain about Type Conversion or casting OR Explain about Widening and narrowing

Type Conversion and Casting

Assigning a value of one type to a variable of another type is known as **Type Casting**.

Java supports two types of castings – **primitive data type casting** and reference type casting. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting. Now let us go for data type casting.

Java data type casting comes with 3 flavors.

1. **Implicit casting**
2. **Explicit casting**
3. **Boolean casting.**

1. automatic type conversion.: Implicit casting (widening conversion)

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as **automatic type conversion**.

Examples:

```
int x = 10;          // occupies 4 bytes
double y = x;        // occupies 8 bytes
System.out.println(y);              // prints 10.0
```

1. Casting Incompatible Types: Explicit casting (narrowing conversion)

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires **explicit casting**; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x = 10.5;// 8 bytes
int y = x;           // 4 bytes ; raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x = 10.5;
int y = (int) x;
```

The double **x** is explicitly converted to int **y**. The thumb rule is, on both sides, the same data type should exist.

Boolean casting

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is **incompatible** for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.

```
boolean x = true;
int y = x;            // error
```

```
boolean x = true;
int y = (int) x;             // error
```

byte –> short –> int –> long –> float –> double

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, **byte** can be assigned to **short** implicitly but **short** to **byte** requires explicit casting.

Operators in java Language:

Arithmetic operators

Operator	Use	Description
+	x + y	Adds x and y
-	x - y	Subtracts y from x
	-x	Arithmetically negates x
*	x * y	Multiplies x by y
/	x / y	Divides x by y
%	x % y	Computes the remainder of dividing x by y

Operator	Use	Description
++	x++	y = x++; is the same as y = x; x = x + 1;
	++x	y = ++x; is the same as x = x + 1; y = x;
--	x--	y = x--; is the same as y = x; x = x - 1;
	--x	y = --x; is the same as x = x - 1; y = x;

Java Example:

```
public class BasicArithmeticDemo
{
public static void main(String[] args)
{
int number1 = 10; int number2 = 5;

//calculating number1 + number2;
int sum = number1 + number2;

//calculating number1 - number2;
int difference = number1 - number2;

//calculating number1 * number2;
 int product = number1 * number2;

//calculating number1 / number2;
int quot = number1 / number2;

//calculating number1 % number2;
int rem = number1 % number2;

//Displaying the values
System.out.println("number1 : "+number1);
System.out.println("number2 : "+number2);
System.out.println("sum : "+sum);
System.out.println("difference : "+difference);
System.out.println("product : "+product);
System.out.println("quot : "+quot);
System.out.println("rem : "+rem);
}
```

Relational Operators

Operator	Use	Description
>	x > y	x is greater than y
>=	x >= y	x is greater than or equal to y
<	x < y	x is less than y
<=	x <= y	x is less than or equal to y
==	x == y	x is equal to y
!=	x != y	x is not equal to y

```
class ComparisonDemo {
public static void main(String[] args){
 int value1 = 1;
int value2 = 2;
if(value1 == value2)
System.out.println("value1 == value2");
if(value1 != value2)
System.out.println("value1 != value2");
if(value1 > value2)
System.out.println("value1 > value2");
if(value1 < value2)
System.out.println("value1 < value2");
if(value1 <= value2)
System.out.println("value1 <= value2");
}
}
```

Bitwise Operators

Operator	Use	Evaluates to true if
~	~x	Bitwise complement of x
&	x & y	AND all bits of x and y
|	x | y	OR all bits of x and y
^	x ^ y	XOR all bits of x and y
>>	x >> y	Shift x right by y bits, with sign extension
>>>	x >>> y	Shift x right by y bits, with 0 fill
<<	x << y	Shift x left by y bits

```
public class BitwiseLogicalOpDemo {
public static void main(String[] args) {
//Integer bitwise logical operator

int a = 65; // binary representation 1000001
 int b = 33; // binary representation 0100001
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);
        }}
```

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:
00101010
becomes 11010101
after the NOT operator is applied.

The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
00101010 42
&00001111 15
____________
00001010 10
```

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
00101010 42
|
00001111 15
___________

00101111 47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```
00101010 42
^
00001111 15
___________

00100101 37
```

The Left Shift

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form: **value << num**

Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand,bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>> (Unsigned right shift Operator)

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

Output:

```
5
5
-5
1073741819
```

Logical Boolean Operators

Operator	Use	Evaluates to true if
&&	x && y	Both x and y are true
||	x || y	Either x or y are true
!	!x	x is not true

Eg:-

```
public class BitwiseLogicalOpDemo {
 public static void main(String[] args) {
//Integer bitwise logical operator
int a = 65; // binary representation 1000001
int b = 33; // binary representation 0100001
System.out.println("a && b= " + (a && b));
System.out.println("a | |b= " + (a | |b));
System.out.println("!a= " + !a);

}}
```

Assignment Operators

Operator	Use	Shortcut for
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y

%=	x %= y	x = x % y
&=	x &= y	x = x & y (also works for boolean values)
|=	x |= y	x = x | y (also works for boolean values)
^=	x ^= y	x = x ^ y (also works for boolean values)
>>=	x >>= y	x = x >> y
>>>=	x >>>= y	x = x >>> y
<<=	x <<= y	x = x << y

```
class AssignOptrDemo
{
        public static void main(String[] args)
        {
int a = 10, b = 15, c = 15;
System.out.println("Assignment and shortcut assignment operators");
System.out.println(" a = " + (a = 15));
System.out.println(" Addition = " + (a += b));
System.out.println(" Subtraction = " + (c -= b));
System.out.println(" Division = " + (a /= 2));
System.out.println(" Multiplication = " + (a *= 2));

}
}
```

Other Operators

Operator	**Use**	**Description**
()	(x + y) * z	Require operator precedence
?:	z = b ? x : y	Equivalent to: if (b) { z = x; } else { z = y; }
[]	array[0]	Access array element
.	str.length()	Access object method or field
(*type*)	int x = (int) 1.2;	Cast from one type to another
new	d = new Date();	Create a new object
instanceof	o instanceof String	Check for object type, returning boolean

Short-Circuit Logical Operators :

Java provides two interesting Boolean operators not found in some other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators.

if (denom != 0 && num / denom > 10)

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero

The ? Operator

The value of a variable often depends on whether a particular boolean expression is or is not true and on nothing else. For instance one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write

```
if (a > b) { max = a;
}
else { max = b;
}
```

Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it, the conditional operator, ?:. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

(a > b) ? a : b; is an expression which returns one of two values, a or b. The condition, (a > b), is tested. If it is true the first value, a, is returned. If it is false, the second value, b, is returned.
Whichever value is returned is dependent on the conditional test, a > b. The condition can be any expression which returns a Boolean value.

```
class ByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2); System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
} }
```

The output generated by this program is shown here:

```
Original value of a: 64 i and b: 256 0
```

Operator Precedence

Highest
++ (postfix) – – (postfix)
++ (prefix) – – (prefix) ~ ! + (unary) – (unary) (*type-cast*)
* / %
+ –
>> >>> <<
> >= < <= instanceof
== !=
&
^
|

<table>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?:</td></tr>
<tr><td>–></td></tr>
<tr><td>= op=</td></tr>
<tr><td><b>Lowest</b></td></tr>
</table>

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement
   - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. **Simple if statement**
2. **if-else statement**
3. **if-else-if ladder**
4. **Nested if-statement**

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java

- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
  case value1:
   statement1;
   break;
  .
  .
  .
  case valueN:
   statementN;
   break;
  default:
   default statement;
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

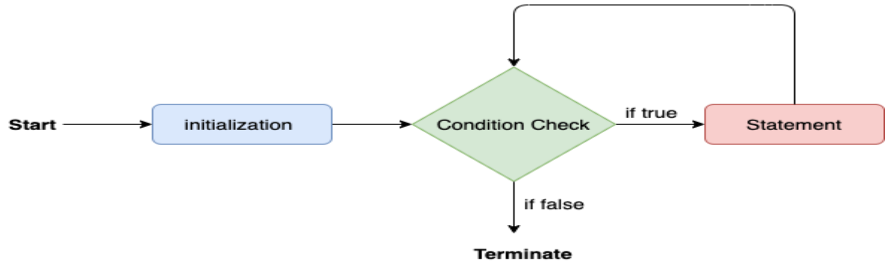
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization;condition; increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

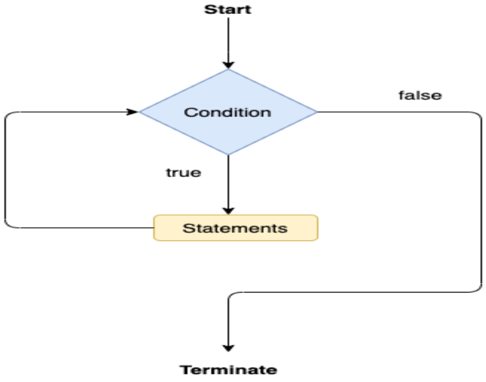
It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){
```

```
//looping statements
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

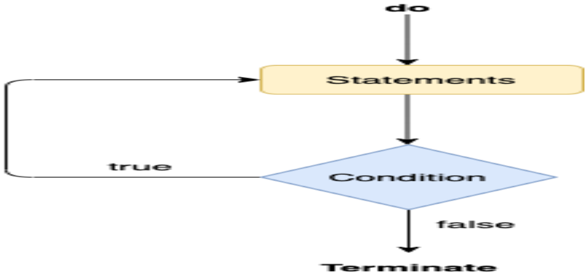
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do
{
//statements
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

Calculation.java

```
public class Calculation {

public static void main(String[] args) {
// TODO Auto-generated method stub
a:
for(int i = 0; i<= 10; i++) {
b:
```

```
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}
}
}

}
}


}
```

Output:

```
0
1
2
3
4
5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {

public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
continue;
}
System.out.println(j);
}
}
}

}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

return

The last control statement is **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.

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

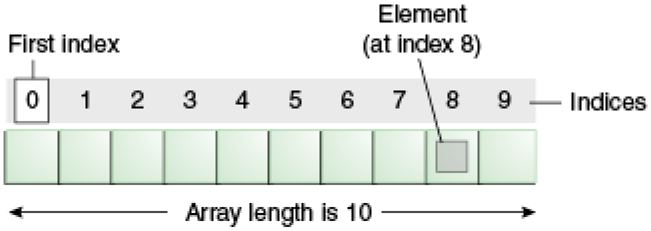
The output from this program is shown here:
Before the return.

Arrays

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

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java.



Advantage of Java Array

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

Disadvantage of Java Array

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

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

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

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example

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

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

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

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Array Length

To find out how many elements an array has, use the **length** property:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);// Outputs 4
```

For-each Loop for Java Array

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Passing Array to a Method in Java

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

```
//Java Program to demonstrate the way of passing an array
//to method.
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
```

```
  min=arr[i];

System.out.println(min);
}
 public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
//Java Program to demonstrate the way of passing an anonymous array
//to method.
public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}

public static void main(String args[]){
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}}
```

Returning Array from the Method

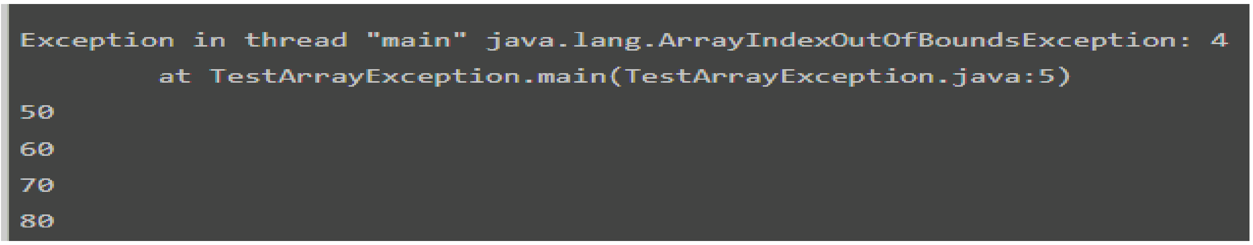
```
//Java Program to return an array from the method
class TestReturnArray{
//creating method which returns an array
static int[] get(){
return new int[]{10,30,50,90,60};
}

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

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.
public class TestArrayException{
public static void main(String args[]){
```

```
int arr[]={50,60,70,80};
for(int i=0;i<=arr.length;i++){
System.out.println(arr[i]);
}
```



```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at TestArrayException.main(TestArrayException.java:5)
50
60
70
80
```

```
}}
```

Multidimensional Array in Java

Arrays can have more than one dimension, these arrays-of-arrays are called *multidimensional arrays*. They are very similar to standard arrays with the exception that they have multiple sets of square brackets after the array identifier. A two dimensional array can be though of as a grid of rows and columns.

Syntax to Declare Multidimensional Array in Java

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

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

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

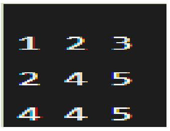
Example of Multidimensional Java Array

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

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
```

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

Output:



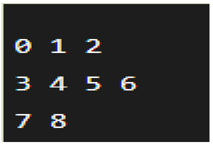
```
1 2 3
2 4 5
4 4 5
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;

        //printing the data of a jagged array
        for (int i=0; i<arr.length; i++){
            for (int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}
```

Output:



```
0 1 2
3 4 5 6
7 8
```

What is the class name of Java array?

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

```
//Java Program to get the class name of array in Java
class Testarray4{
public static void main(String args[]){
//declaration and initialization of array
int arr[]={4,4,5};
//getting the class name of Java array
Class c=arr.getClass();
String name=c.getName();
//printing the class name of Java array
System.out.println(name);

}}
```

Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

Syntax of arraycopy method

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

Example of Copying an Array in Java

```
//Java Program to copy a source array into a destination array in Java
class TestArrayCopyDemo {
   public static void main(String[] args) {
      //declaring a source array
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  'i', 'n', 'a', 't', 'e', 'd' };
      //declaring a destination array
      char[] copyTo = new char[7];
      //copying array using System.arraycopy() method
      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      //printing the destination array
      System.out.println(String.valueOf(copyTo));
   }
}
```

Output:

14. caffein

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, **it creates the shallow copy of the Java array which means it copies the references.**

```
//Java Program to clone the array
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
System.out.println("Printing original array:");
for(int i:arr)
System.out.println(i);
 System.out.println("Printing clone of the array:");
int carr[]=arr.clone();
for(int i:carr)
System.out.println(i);
System.out.println("Are both equal?");
System.out.println(arr==carr);
}}
```

Output:

```
Printing original array:
33
3
4
5
Printing clone of the array:
33
3
4
5
Are both equal?
false
```

Q. Explain about toString() and deepToString() method

To *print Java array in a meaningful way*, you don't need to look further because your very own Collection framework provides lots of array utility methods in java.util.Arrays class. Here we have toString() and deepToString() method to print array in Java.

```
import java.util.Arrays;
 /** * Java Program to print arrays in Java.
 * We will learn how to print String, int,
* byte and two dimensional arrays in Java by using toString() and
* deepToString() method of Arrays class.
*/
 public class PrintArrayInJava{ public static void main(String args[]) {
 // Example 1 : print int array in Java
 int[] primes = {5, 7, 11, 17, 19, 23, 29, 31, 37};
 System.out.println("Prime numbers : " + primes); // Not OK
 System.out.println("Real prime numbers : " + Arrays.toString(primes)); //Ok
 // Example 2 : print String array in Java
 String[] buzzwords = {"Java", "Android", "iOS", "Scala", "Python"};
 System.out.println("Buzzing .." + buzzwords);
 System.out.println("Not buzzing? try again : " + Arrays.toString(buzzwords));
 // Example 3 : print two dimensional array in Java
 String[][] phones = {{"Apple", "iPhone"}, {"Samsung", "Galaxy"}, {"Sony", "Xperia"}};
 System.out.println("Hot phones .. " + phones);
 System.out.println("Not hot? See again.." + Arrays.deepToString(phones));
 // Example 4 : print byte array in Java
 String random = "In Java programming langue, array is object";
 byte[] bytes = random.getBytes();
 System.out.println("What is inside bytes : " + bytes);
 System.out.println("Not visible, check closely .." + Arrays.toString(bytes));
   }
 }
```

Output:

```
Prime numbers : [I@5eb1404f
Real prime numbers : [5, 7, 11, 17, 19, 23, 29, 31, 37]
Buzzing ..[Ljava.lang.String;@46f5331a
Not buzzing? try again : [Java, Android, iOS, Scala, Python]
Hot phones .. [[Ljava.lang.String;@57398044
Not hot? See again..[[Apple, iPhone], [Samsung, Galaxy], [Sony, Xperia]]
What is inside bytes : [B@31602bbc
Not visible, check closely ..[73, 110, 32, 74, 97, 118, 97, 32, 112,
114, 111, 103, 114, 97, 109, 109, 105,
110, 103, 32, 108, 97, 110, 97, 103, 117, 101, 44, 32, 97, 114, 114, 97, 121, 32, 105,
115, 32, 111, 98, 106, 101, 99, 116]
```

Naming Conventions of the Different Identifiers

The following table shows the popular conventions used for the different identifiers.

Identifiers Type	Naming Rules	Examples

Class	It should start with the uppercase letter. It should be a noun such as Color, Button, System, Thread, etc. Use appropriate words, instead of acronyms.	public class **Employee**
		{
		//code snippet
		}

Interface	It should start with the uppercase letter.
	It should be an adjective such as Runnable, Remote, ActionListener.
	Use appropriate words`, instead of acronyms.
		interface **Printable**
		{
		//code snippet
		}

Method	It should start with lowercase letter.
	It should be a verb such as main(), print(), println().
	If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
		class Employee
		{
		// method
		void **draw()**
		{
		//code snippet
		}
		}

Variable	It should start with a lowercase letter such as id, name.
	It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
	Avoid using one-character variables such as x, y, z.
		class Employee
		{
		// variable
		int **id**;
		//code snippet
		}

Package	It should be a lowercase letter such as java, lang.
	If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
		//package
		package **com.javatpoint;**
		class Employee
		{
		//code snippet
		}

Constant	It should be in uppercase letters such as RED, YELLOW.
	If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
	It may contain digits but not as the first letter.
		class Employee
		{
		//constant
		static final int **MIN_AGE** = 18;
		//code snippet
		}

CamelCase in Java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Q. Write a short note on Class and Objects

Objects and Classes in Java

In object-oriented programming technique, we design a program using objects and classes.
An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Syntax to declare a class:

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

Instance variable in Java

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.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization
- new keyword in Java
- The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.
File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101 Sonoo
```

```
102 Amit
```

2) Object and Class Example: Initialization through method

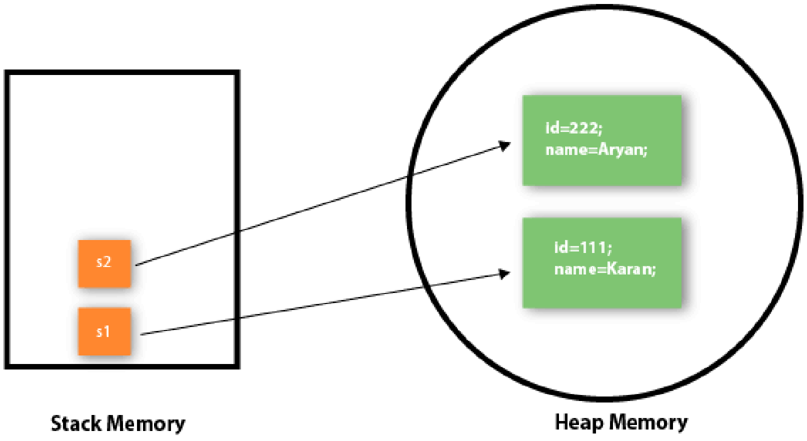
In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

```
111 Karan
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
class Employee{
  int id;
  String name;
  float salary;
  void insert(int i, String n, float s) {
    id=i;
    name=n;
    salary=s;
  }
  void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
  Employee e1=new Employee();
  Employee e2=new Employee();
  Employee e3=new Employee();
  e1.insert(101,"ajeet",45000);
  e2.insert(102,"irfan",25000);
  e3.insert(103,"nakul",55000);
  e1.display();
  e2.display();
  e3.display();
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

new Calculation();//anonymous object

Calling method through a reference:

```
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

int a=10, b=20;

Initialization of refernce variables:

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();//creating two objects

Let's see the example:

```
//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
 public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

```
55
```

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
class Account{
        int acc_no;
        String name;
        float amount;
        //Method to initialize object
        void insert(int a,String n,float amt){
                acc_no=a;
                name=n;
                amount=amt;
        }
        //deposit method
        void deposit(float amt){
                amount=amount+amt;
                System.out.println(amt+" deposited");
        }
        //withdraw method
        void withdraw(float amt){
                if(amount<amt){
                System.out.println("Insufficient Balance");
                }else{
                        amount=amount-amt;
                        System.out.println(amt+" withdrawn");
                }
        }
//method to check the balance of the account
        void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
        void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
        public static void main(String[] args){
                Account a1=new Account();
                a1.insert(832345,"Ankit",1000);
                a1.display();
                a1.checkBalance();
                a1.deposit(40000);
                a1.checkBalance();
                a1.withdraw(15000);
                a1.checkBalance();
        }
}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Q. Explain about Constructors in Java

Constructors in Java

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 whic h 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.

Rules for creating Java constructor

There are two 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 constructors

There are two types of constructors in Java:

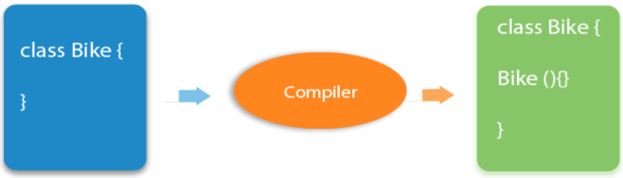
1. Default constructor
2. no-argument constructor
3. Parameterized constructor

Java Default Constructor

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

```
<class_name>(){}
```

Example of default constructor



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

```
//Java Program to create and call a default constructor
class Bike1{
int bikeNo;
float bikecost;
String bikeName;
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
System.out.println("bikeNo=" + b.bikeno);
System.out.println("bikeCost=" + b.bikeCost);
System.out.println("bikeName=" + b.bikeName);

}
}
```

Output:

```
bikeNo=0
bikeCost=0.0f
bikeName=null
```

no-argument constructor

Similar to methods, a Java constructor may or may not have any parameters (arguments).
If a constructor does not accept any parameters, it is known as a no-argument constructor. For example

```
class Main {
  int i;
  // constructor with no parameter
  private Main() {
    i = 5;
    System.out.println("Constructor is called");
  }

  public static void main(String[] args) {

    // calling the constructor without any parameter
    Main obj = new Main();
    System.out.println("Value of i: " + obj.i);
  }
}
```

output:

```
Constructor is called
Value of i: 5
```

Java Parameterized Constructor

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

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

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

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

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

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

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student5(int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}

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

output:

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

Difference between constructor and method in Java

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

Java Constructor	Java Method

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

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

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

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

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

```
//Java program to initialize the values from one object to another object.
class Student6{
  int id;
  String name;
  //constructor to initialize integer and string
  Student6(int i,String n){
  id = i;
  name = n;
  }
  //constructor to initialize another object
  Student6(Student6 s){
  id = s.id;
  name =s.name;
  }
  void display(){System.out.println(id+" "+name);}

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

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{
  int id;
  String name;
  Student7(int i,String n){
  id = i;
  name = n;
  }
  Student7(){}
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student7 s1 = new Student7(111,"Karan");
  Student7 s2 = new Student7();
  s2.id=s1.id;
  s2.name=s1.name;
  s1.display();
  s2.display();
 }
}
```

Output:

```
111 Karan
111 Karan
```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

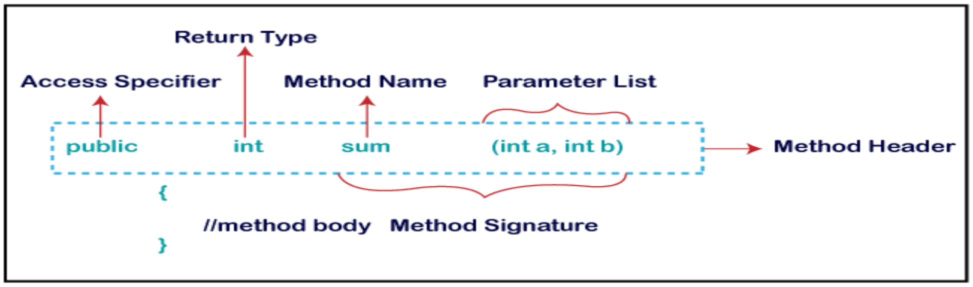
Method in Java

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.

Method Declaration



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

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. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** 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.

example:

Single-word method name: **sum(), area()**

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

Output:

```
The maximum number is: 9
```

In the above example, we have used three predefined methods **main(), print(),** and **max()**. We have used these methods directly without declaration because they are predefined. The print() method is a method of **PrintStream** class that prints the result on the console. The max() method is a method of the **Math** class that returns the greater of two numbers.

User-defined Method

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will

```
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

```
public class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
int c = add(a, b);   //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
public static int add(int n1, int n2)   //n1 and n2 are formal parameters
{
int s;
s=n1+n2;
return s; //returning the sum
}
}
```

Output:

```
The sum of a and b is= 24
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Display.java

```
public class Display
{
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Output:

```
It is an example of a static method.
```

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

InstanceMethodExample.java

```
public class InstanceMethodExample
{
public static void main(String [] args)
{
//Creating an object of the class
InstanceMethodExample obj = new InstanceMethodExample();
//invoking instance method
System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-defined method because we have not used static keyword
public int add(int a, int b)
{
s = a+b;
//returning the sum
return s;
}
}
```

Output:

```
The sum is: 25
```

There are two types of instance method:

- **Accessor Method**
- **Mutator Method**

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. It is also known as **getters**. It returns the value of the private field. It is used to get the value of the private field.

Example

```
public int getId()
{
return Id;
}
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word **set**. It is also known as **setters** or **modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
public void setRoll(int roll)
{
this.roll = roll;
}
```

Example of accessor and mutator method

Student.java

```
public class Student
{
private int roll;
private String name;
public int getRoll()    //accessor method
{
return roll;
}
public void setRoll(int roll) //mutator method
{
this.roll = roll;
}
public String getName()
{
```

```
return name;
}
public void setName(String name)
{
this.name = name;
}
public void display()
{
System.out.println("Roll no.: "+roll);
System.out.println("Student name: "+name);
}
}
```

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

Syntax

abstract void method_name();

Example of abstract method

Demo.java

```
abstract class Demo //abstract class
{
//abstract method declaration
abstract void display();
}
public class MyClass extends Demo
{
//method impelmentation
void display()
{
System.out.println("Abstract method?");
}
public static void main(String args[])
{
//creating object of abstract class
Demo obj = new MyClass();
//invoking abstract method
obj.display();
}
}
```

Output:

```
Abstract method...
```

Factory method

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

Advantage of Factory Design Pattern

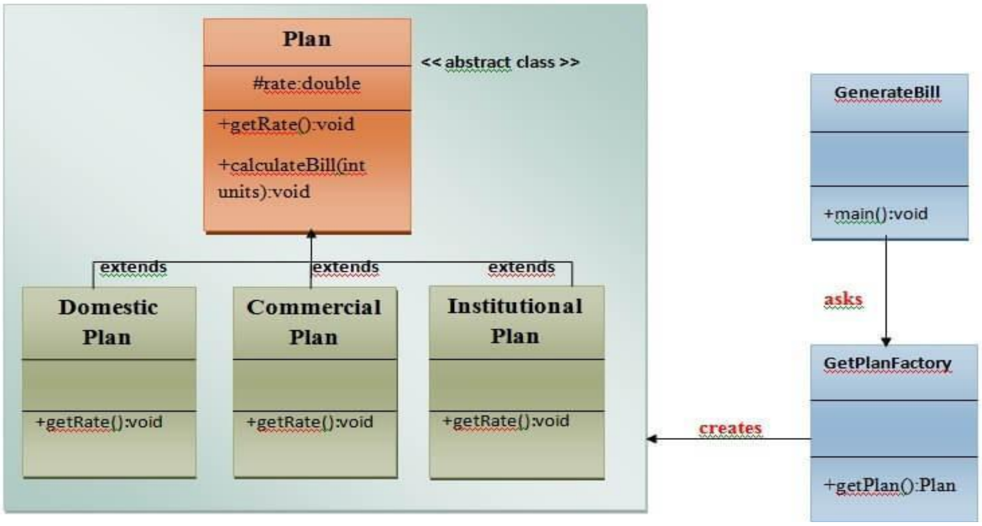
- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

UML for Factory Method Pattern

- We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
- GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.



Calculate Electricity Bill : A Real World Example of Factory Method

Step 1: Create a Plan abstract class.

```
import java.io.*;
abstract class Plan{
        protected double rate;
        abstract void getRate();

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}//end of Plan class.
```

Step 2: Create the concrete classes that extends Plan abstract class.

```
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
           rate=3.50;
       }
   }//end of DomesticPlan class.
class  CommercialPlan extends Plan{
   //@override
    public void getRate(){
       rate=7.50;
   }
/end of CommercialPlan class.
class  InstitutionalPlan extends Plan{
   //@override
    public void getRate(){
        rate=5.50;
    }
/end of InstitutionalPlan class.
```

Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information..

```
class GetPlanFactory{

   //use getPlan method to get object of type Plan
       public Plan getPlan(String planType){
            if(planType == null){
             return null;
            }
          if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
                 return new DomesticPlan();
               }
           else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
                return new CommercialPlan();
            }
          else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
                return new InstitutionalPlan();
          }
      return null;
   }
}//end of GetPlanFactory class.
```

Step 4: Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

```
import java.io.*;
class GenerateBill{
    public static void main(String args[])throws IOException{
      GetPlanFactory planFactory = new GetPlanFactory();

      System.out.print("Enter the name of plan for which the bill will be generated: ");
      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

      String planName=br.readLine();
      System.out.print("Enter the number of units for bill will be calculated: ");
      int units=Integer.parseInt(br.readLine());

      Plan p = planFactory.getPlan(planName);
      //call getRate() method and calculateBill()method of DomesticPaln.

       System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
           p.getRate();
           p.calculateBill(units);
```

```
        }
  }//end of GenerateBill class.
```

Call By Value

Call by Value means calling a method with a parameter as value. Through this, the argument value is passed to the parameter.

```
public class Tester{
  public static void main(String[] args){
    int a = 30;
    int b = 45;
    System.out.println("Before swapping, a = " + a + " and b = " + b);
    // Invoke the swap method
    swapFunction(a, b);
    System.out.println("\n**Now, Before and After swapping values will be same here**:");
    System.out.println("After swapping, a = " + a + " and b is " + b);
  }
  public static void swapFunction(int a, int b) {
    System.out.println("Before swapping(Inside), a = " + a + " b = " + b);
    // Swap n1 with n2
    int c = a;
    a = b;
    b = c;
    System.out.println("After swapping(Inside), a = " + a + " b = " + b);
  }
}
```

Output

This will produce the following result −

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30
**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Call By Reference

Java uses only call by value while passing reference variables as well. It creates a copy of references and passes them as valuable to the methods. As reference points to same address of object, creating a copy of reference is of no harm. But if new object is assigned to reference it will not be reflected.

```
public class JavaTester {
  public static void main(String[] args) {
    IntWrapper a = new IntWrapper(30);
    IntWrapper b = new IntWrapper(45);
    System.out.println("Before swapping, a = " + a.a + " and b = " + b.a);
    // Invoke the swap method
    swapFunction(a, b);
    System.out.println("\n**Now, Before and After swapping values will be different here**:");
    System.out.println("After swapping, a = " + a.a + " and b is " + b.a);
  }
  public static void swapFunction(IntWrapper a, IntWrapper b) {
    System.out.println("Before swapping(Inside), a = " + a.a + " b = " + b.a);
    // Swap n1 with n2
    IntWrapper c = new IntWrapper(a.a);
    a.a = b.a;
    b.a = c.a;
    System.out.println("After swapping(Inside), a = " + a.a + " b = " + b.a);
  }
}
class IntWrapper {
  public int a;
  public IntWrapper(int a){ this.a = a;}
}
```

This will produce the following result –

Output

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30
**Now, Before and After swapping values will be different here**:
After swapping, a = 45 and b is 30
```

Variable Arguments(var-args)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows −

typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example

```
public class VarargsDemo {

  public static void main(String args[]) {
    // Call method with variable args
            printMax(34, 3, 3, 2, 56.5);
    printMax(new double[]{1, 2, 3});
  }

  public static void printMax( double... numbers) {
    if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
    }

    double result = numbers[0];

    for (int i = 1; i <  numbers.length; i++)
    if (numbers[i] >  result)
    result = numbers[i];
    System.out.println("The max value is " + result);
  }
}
```

This will produce the following result −

Output

```
The max value is 56.5
The max value is 3.0
```

The finalize( ) Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

The finalize( ) method has this general form −

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

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.

This means that you cannot know when or even if finalize( ) will be executed. For example, if your program ends before garbage collection occurs, finalize( ) will not execute.

Java static keyword

The **static keyword** in Java is used for memory management mainly.

The static can be:

1. **Variable (also known as a class variable)**
2. **Method (also known as a class method)**
3. **Block**
4. **Nested class**

1) Java static variable

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

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

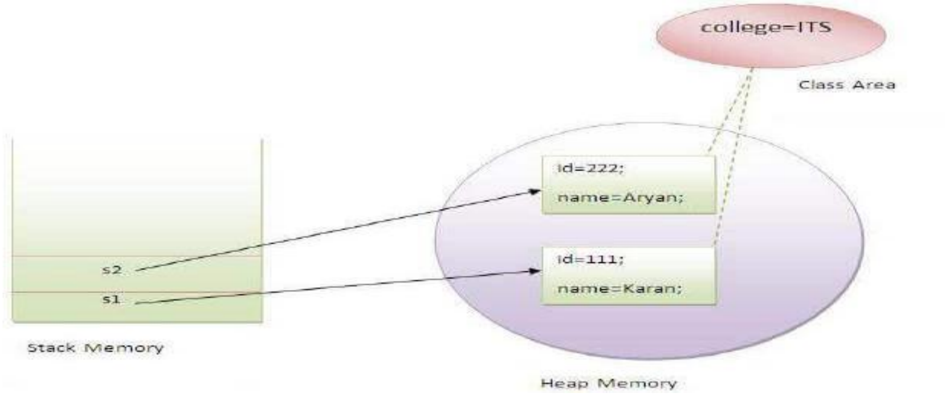
It makes your program **memory efficient** (i.e., it saves memory).

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
  String name;
  static String college ="AITS";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Output:

```
111 Karan AITS
222 Aryan AITS
```



Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.
class Counter{
int count=0;//will get memory each time when the instance is created

Counter(){
count++;//incrementing value
System.out.println(count);
}

public static void main(String args[]){
//Creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

Output:

```
1
1
1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2{
```

```
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Output:

```
1
2
3
```

2) Java static method

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

- A static method belongs to the class rather than 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.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "AITS";
    //static method to change the value of static variable
    static void change(){
    college = "AITS-TPT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   //calling display method
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Output:

```
  111 Karan AITS-TPT
   222 Aryan AITS-TPT
   333 Sonoo AITS-TPT
```

```
//Java Program to get the cube of a given number using the static method

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

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

Output:125

Restrictions for the static method

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

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

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

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

Output: Compile Time Error

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

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

Example of static block

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

Output:static block is invoked
Hello main

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method

```
.class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

Output:

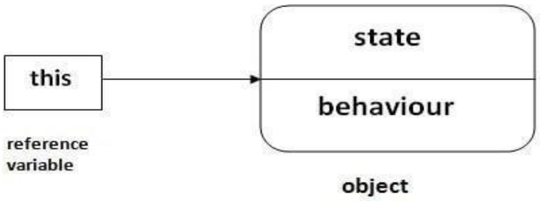
```
static block is invoked
```

Since JDK 1.7 and above, output would be:

```
Error: Main method not found in class A3, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

this keyword in Java

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



Usage of Java this keyword

Here is given the 6 usage of java this keyword.

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

1) this: to refer current class instance variable

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

Understanding the problem without this keyword

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

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

Output:

```
0 null 0.0
0 null 0.0
```

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

Solution of the above problem by this keyword

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

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

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

Program where this keyword is not required

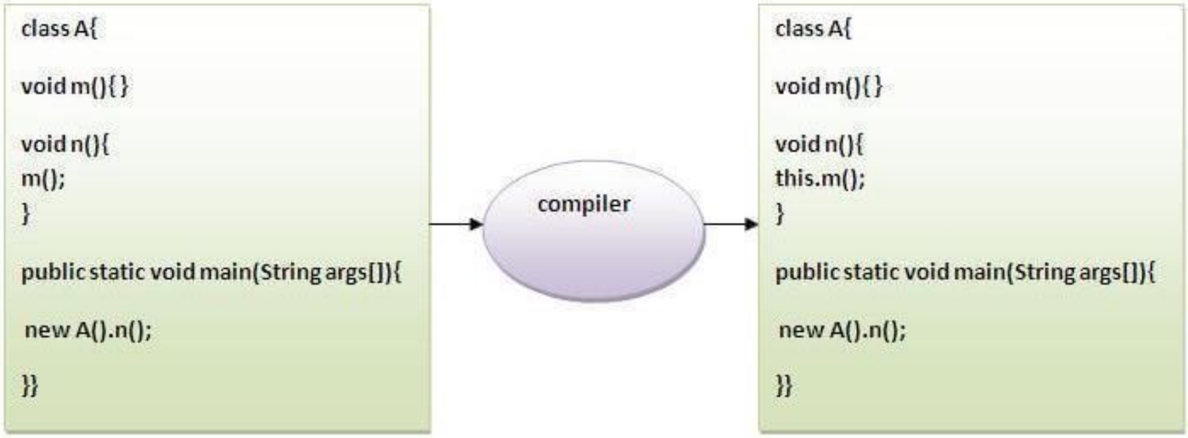
```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

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

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

2) this: to invoke current class method

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



```
class A{

void m(){}

void n(){
m();
}

public static void main(String args[]){

new A().n();

}}
```

```
class A{

void m(){}

void n(){
this.m();
}

public static void main(String args[]){

new A().n();

}}
```

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

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor

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

Calling default constructor from parameterized constructor:

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

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

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

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

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

Output:

```
111 ankit java 0.0
112 sumit java 6000.0
```

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

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);//C.T.Error
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis8{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

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

4) this: to pass as an argument in the method

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

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

Output:

```
method is invoked
```

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

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

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}

class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

```
Output:10
```

6) this keyword can be used to return current class instance

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

Syntax of this that can be returned as a statement

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

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

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

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

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

Output:

```
A5@22b3ea59
```

```
A5@22b3ea59
```

Java String

In Java , string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

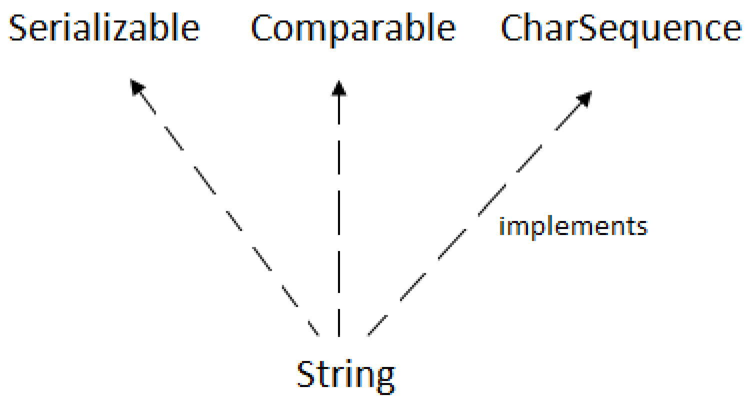
char[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=new String(ch);

is same as:

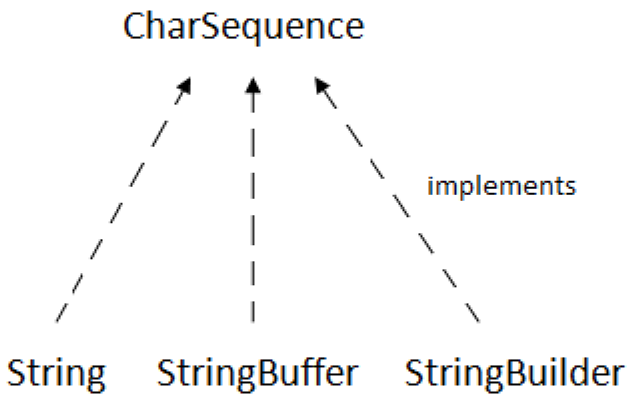
String s="javatpoint";

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

How to create a string object?

There are two ways to create String object:

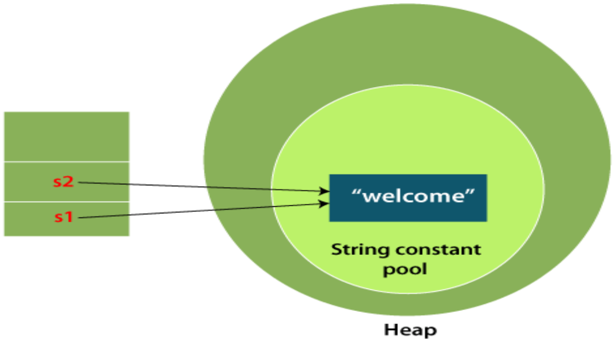
1. By string literal
2. By new keyword

1) 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. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```



2) By new keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM

will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

StringExample.java

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index

2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.

19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

```
public class StringMethodsDemo {
        public static void main(String[] args) {
                String targetString = "Java is fun to learn";
                String s1= "JAVA";
                String s2= "Java";
                String s3 = "  Hello Java  ";
                System.out.println("Char at index 2(third position): " + targetString.charAt(2));
                System.out.println("After Concat: "+ targetString.concat("-Enjoy-"));
                System.out.println("Checking equals ignoring case: " +s2.equalsIgnoreCase(s1));
                System.out.println("Checking equals with case: " +s2.equals(s1));
                System.out.println("Checking Length: "+ targetString.length());
                System.out.println("Replace function: "+ targetString.replace("fun", "easy"));
                System.out.println("SubString of targetString: "+ targetString.substring(8));
                System.out.println("SubString of targetString: "+ targetString.substring(8, 12));
                System.out.println("Converting to lower case: "+ targetString.toLowerCase());
                System.out.println("Converting to upper case: "+ targetString.toUpperCase());
                System.out.println("Triming string: " + s3.trim());
                System.out.println("searching s1 in targetString: " + targetString.contains(s1));
                System.out.println("searching s2 in targetString: " + targetString.contains(s2));
```

```
                    char [] charArray = s2.toCharArray();
                    System.out.println("Size of char array: " + charArray.length);
                    System.out.println("Printing last element of array: " + charArray[3]);
                    }

}
```

Output:

```
Char at index 2(third position): v
After Concat: Java is fun to learn-Enjoy-
Checking equals ignoring case: true
Checking equals with case: false
Checking Length: 20
Replace function: Java is easy to learn
SubString of targetString: fun to learn
SubString of targetString: fun
Converting to lower case: java is fun to learn
Converting to upper case: JAVA IS FUN TO LEARN
Triming string: Hello Java
searching s1 in targetString: false
searching s2 in targetString: true
Size of char array: 4
Printing last element of array: a
```

Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

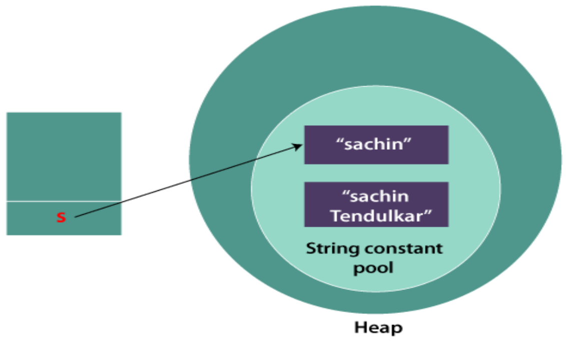
Once String object is created its data or state can't be changed but a new String object is created.

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Output:

```
Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but ***s*** reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

Testimmutablestring1.java

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

Output:

```
Sachin Tendulkar
```

In such a case, s points to the "Sachin Tendulkar". Please notice that still Sachin object is not modified.

Why String objects are immutable in Java?

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

1. ClassLoader:

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, String is immutable.

2. Thread Safe:
As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

3. Security:
As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

4. Heap Space:
The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

Why String class is Final in Java?
The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer Class

Constructor	**Description**
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public	insert(int offset, String	It is used to insert the specified string with this string at

synchronized StringBuffer	s)	the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

```
package com.basics;
import java.lang.*;

class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java

sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello

sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo

sb.delete(1,3);
System.out.println(sb);//prints Hlo

sb.reverse();
System.out.println(sb);//prints olleH

StringBuffer sb1=new StringBuffer();
System.out.println(sb1.capacity());//default 16
sb1.append("Hello");
System.out.println(sb1.capacity());//now 16
sb1.append("java is my favourite language");
System.out.println(sb1.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2

sb1.ensureCapacity(10);//now no change
System.out.println(sb1.capacity());//now 34
sb1.ensureCapacity(50);//now (34*2)+2
System.out.println(sb1.capacity());//now 70

}
}
```

Output:

```
Hello Java
HJavaello Java
HJavavaello Java
Hvavaello Java
avaJ olleavavH
16
16
34
34
70
```

Java StringBuilder Class

Important Constructors of StringBuilder class

Constructor	Description

StringBuilder()	It creates an empty String Builder with the initial capacity of 16.
StringBuilder(String str)	It creates a String Builder with the specified string.
StringBuilder(int length)	It creates an empty String Builder with the specified capacity as length.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public StringBuilder insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	It is used to reverse the string.
public int capacity()	It is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	It is used to return the character at the specified position.
public int length()	It is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

```
package com.basics;
import java.lang.*;

class StringBuilderExample{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java

sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello

sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo

sb.delete(1,3);
System.out.println(sb);//prints Hlo

sb.reverse();
System.out.println(sb);//prints olleH

StringBuilder sb1=new StringBuilder();
System.out.println(sb1.capacity());//default 16
sb1.append("Hello");
System.out.println(sb1.capacity());//now 16
sb1.append("Java is my favourite language");
System.out.println(sb1.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2

StringBuilder sb2=new StringBuilder();
System.out.println(sb2.capacity());//default 16
sb2.append("Hello");
System.out.println(sb2.capacity());//now 16
sb2.append("Java is my favourite language");
System.out.println(sb2.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb2.ensureCapacity(10);//now no change
System.out.println(sb2.capacity());//now 34
sb2.ensureCapacity(50);//now (34*2)+2
System.out.println(sb2.capacity());//now 70
}
}
```

Output:

```
Hello Java
HJavaello Java
HJavavaello Java
Hvavaello Java
```

```
avaJ olleavavH
16
16
34
16
16
34
34
70
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

Performance Test of String and StringBuffer

ConcatTest.java

```
public class ConcatTest{
  public static String concatWithString()    {
    String t = "Java";
    for (int i=0; i<10000; i++){
      t = t + "Tpoint";
    }
    return t;
  }
  public static String concatWithStringBuffer(){
    StringBuffer sb = new StringBuffer("Java");
    for (int i=0; i<10000; i++){
            sb.append("Tpoint");
        }
        return sb.toString();
    }
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        concatWithString();
        System.out.println("Time taken by Concating with String: "+(System.currentTimeMillis()-
startTime)+"ms");
        startTime = System.currentTimeMillis();
        concatWithStringBuffer();
        System.out.println("Time taken by Concating with  StringBuffer: "+(System.currentTimeMillis()-
startTime)+"ms");
    }
}
```

Output:

```
Time taken by Concating with String: 578ms
Time taken by Concating with  StringBuffer: 0ms
```

The above code, calculates the time required for concatenating a string using the String class and StringBuffer class.

Difference between StringBuffer and StringBuilder

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

Performance Test of StringBuffer and StringBuilder

Let's see the code to check the performance of StringBuffer and StringBuilder classes.

ConcatTest.java

```
//Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
public class ConcatTest{
   public static void main(String[] args){
      long startTime = System.currentTimeMillis();
      StringBuffer sb = new StringBuffer("Java");
      for (int i=0; i<10000; i++){
         sb.append("Tpoint");
      }
      System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) + "ms");
      startTime = System.currentTimeMillis();
      StringBuilder sb2 = new StringBuilder("Java");
      for (int i=0; i<10000; i++){
         sb2.append("Tpoint");
      }
      System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) + "ms");
   }
}
```

Output:

```
Time taken by StringBuffer: 16ms
Time taken by StringBuilder: 0ms
```

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name :** Object Oriented Programming Through JAVA	**AY:** 2021-2022
Unit 2 (Inheritance, Polymorphism, Interfaces, packages)			

Inheritance: **It** is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

use inheritance in java:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Important terminology:

Super Class: The class whose features are inherited is known as super class (or a base class or a parent class).

Sub Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the super class fields and methods.

Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is **extends**.

Syntax :

```
class derived-class extends base-class  {
   //methods and fields
}
```

Example: In the below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class that extends Bicycle class and class Test is a driver class to run program.

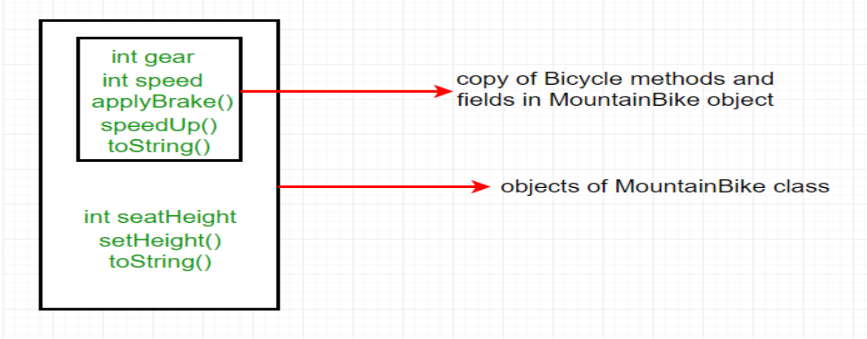
```
// Java program to illustrate the
// concept of inheritance

// base class
class Bicycle {
        // the Bicycle class has two fields
        public int gear;
        public int speed;

        // the Bicycle class has one constructor
        public Bicycle(int gear, int speed){
                this.gear = gear;
                this.speed = speed;
        }
        // the Bicycle class has three methods
        public void applyBrake(int decrement)   {
                speed -= decrement;
        }
        public void speedUp(int increment){
```

```
        speed += increment;      }

    // toString() method to print info of Bicycle
    public String toString()   {
        return ("No of gears are " + gear + "\n"
                        + "speed of bicycle is " + speed);
    }
}

// derived class
class MountainBike extends Bicycle {
    // the MountainBike subclass adds one more field
    public int seatHeight;
    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed, int startHeight){
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }
    // the MountainBike subclass adds one more method
    public void setHeight(int newValue){
        seatHeight = newValue;
    }
    // overriding toString() method
    // of Bicycle to print more info
    @Override public String toString()         {
        return (super.toString() + "\nseat height is "+ seatHeight);
    }
}

// driver class
public class Test {
    public static void main(String args[]){
        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

Output

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

Illustrative image of the program:


int gear
int speed
applyBrake()
speedUp()
toString()
copy of Bicycle methods and fields in MountainBike object
objects of MountainBike class
int seatHeight
setHeight()
toString()

Member Access and Inheritance

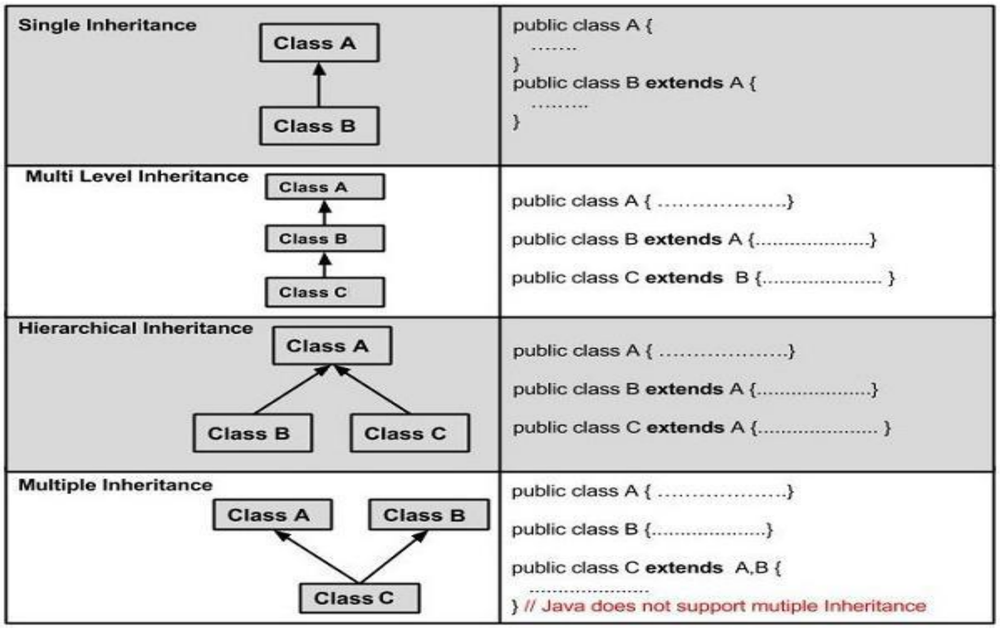
Although a subclass includes all of the members of its super class, it cannot access those members of the super class that have been declared as private.

For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class. This program contains an error and will not compile.
*/
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x; j = y;
}
}
// A's j is not accessible here.

class B extends A { int total;
void sum() {
total = i + j;        // ERROR, j is not accessible here
}
}

class Access {
public static void main(String args[]) { B subOb = new B();
subOb.setij(10, 12); subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

Types of Inheritance in Java

Below are the different types of inheritance which are supported by Java.



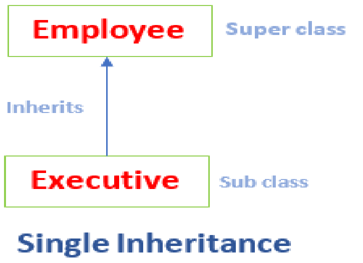
```
public class A {
   .......
}
public class B extends A {
   .........
}
```

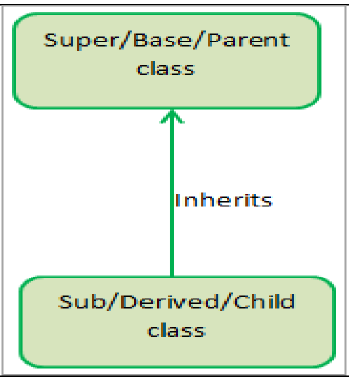
```
public class A { ....................}
public class B extends A {....................}
public class C extends  B {..................... }
```

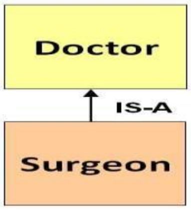
```
public class A { ...................}
public class B extends A {....................}
public class C extends A {..................... }
```

```
public class A { ...................}
public class B {....................}
public class C extends  A,B {
   ......................
} // Java does not support mutiple Inheritance
```

Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.





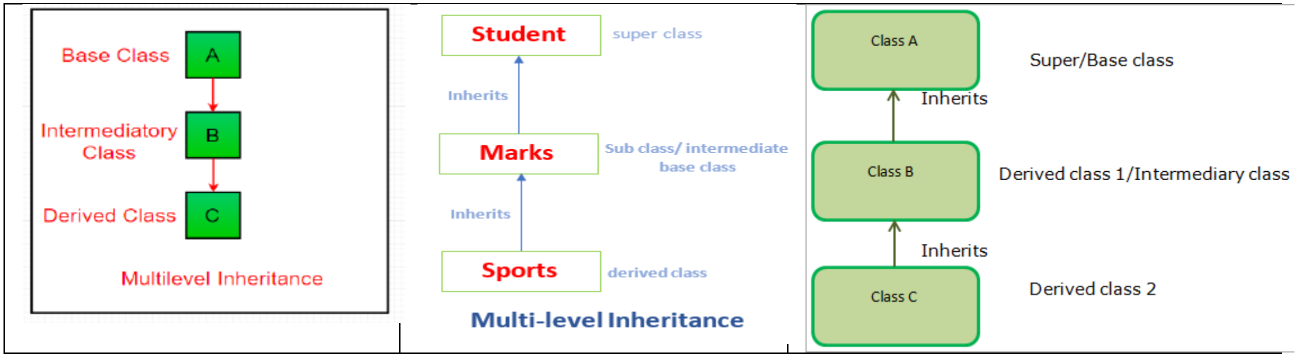


```
// Java program to illustrate the
// concept of single inheritance
class ParentClass{
        int a;
        void setData(int a) {
                this.a = a;
        }
}
class ChildClass extends ParentClass{
        void showData() {
                System.out.println("Value of a is " + a);
        }
}
```

```
public class SingleInheritance {
        public static void main(String[] args) {
                ChildClass obj = new ChildClass();
                obj.setData(100);
                obj.showData();
        }
}
```

Output
Value of a is 100

2. Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

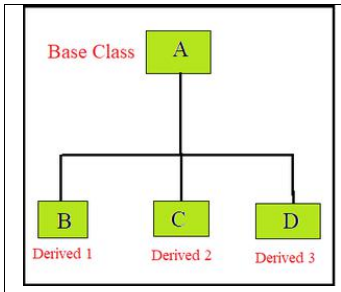


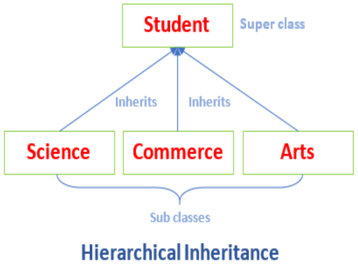
```
// Java program to illustrate the
// concept of Multilevel inheritance
class ParentClass{
        int a;
        void setData(int a) {
                this.a = a;
        }
}
class ChildClass extends ParentClass{
        void showData() {
                System.out.println("Value of a is " + a);
        }
}
class ChildChildClass extends ChildClass{
        void display() {
                System.out.println("Inside ChildChildClass!");
        }
}
public class MultipleInheritance {
        public static void main(String[] args) {
                ChildChildClass obj = new ChildChildClass();
                obj.setData(100);
                obj.showData();
                obj.display();
        }
}
```

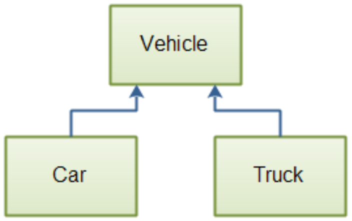
Output

```
Value of a is 100
Inside CHldChild Class!
```

3. Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.







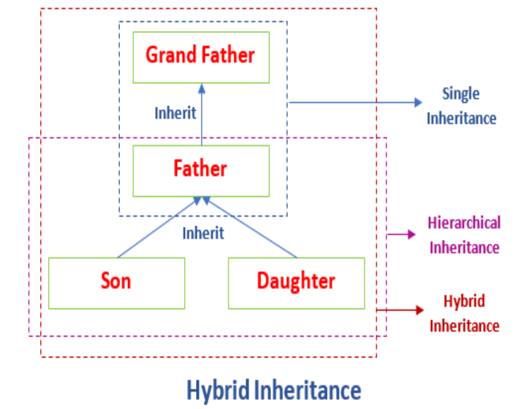
```
// Java program to illustrate the
// concept of Hierarchical inheritance
class ParentClass{
        int a;
        void setData(int a) {
                this.a = a;
        }
}
class ChildClass extends ParentClass{
        void showData() {
                System.out.println("Inside ChildClass!");
                System.out.println("Value of a is " + a);
        }
}
class ChildClassToo extends ParentClass{
        void display() {
                System.out.println("Inside ChildClassToo!");
                System.out.println("Value of a is " + a);
        }
}
public class HierarchicalInheritance {
        public static void main(String[] args) {
                ChildClass child_obj = new ChildClass();
```

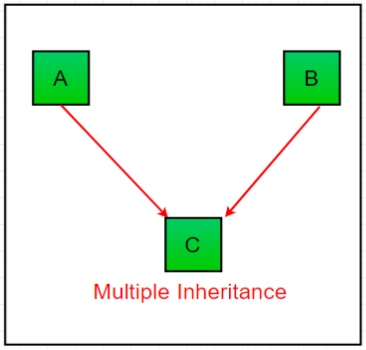
```
                    child_obj.setData(100);
                    child_obj.showData();

                    ChildClassToo childToo_obj = new ChildClassToo();
                    childToo_obj.setData(200);
                    childToo_obj.display();
          }
}
```

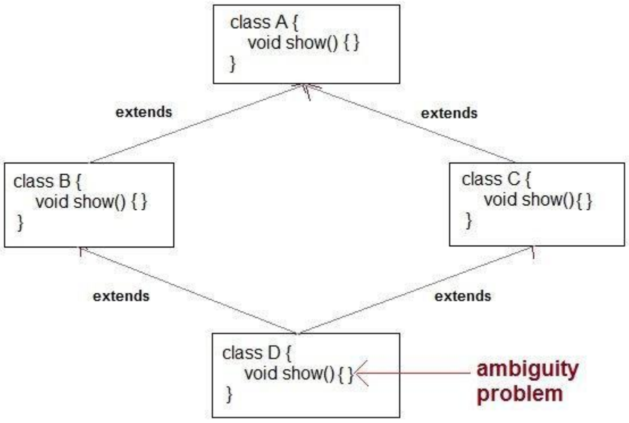
Output

```
Inside ChildClass!
Value of a is 100
Inside ChildClassToo!
Value of a is 100
```

4. Multiple Inheritance (Through Interfaces): In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interface A and B.







// Java program to illustrate the
// concept of Multiple inheritance

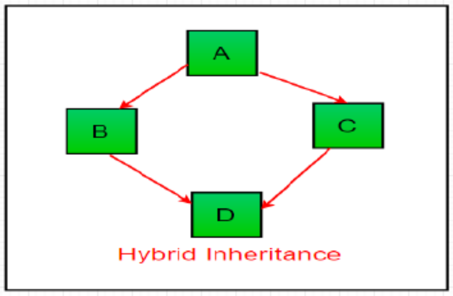
```
interface CollegeData{
   public void collegeDetail();
   public void studentData();
}

interface HostelData{
   public void hostelDetail();
   public void studentRecord();
}

public class StudentRecord implements CollegeData, HostelData{
   @Override
   public void hostelDetail()     {
      System.out.println("Hostel Name : RAMA");
      System.out.println("Hostel location : AITS");
   }

   @Override
   public void studentRecord()     {
      System.out.println("Student selected on based : Percentage, Financial condition");
   }

   @Override
   public void collegeDetail()     {
      System.out.println("College Name : AITS");
      System.out.println("College Grade : A");
      System.out.println("University of College : JNTUA");
   }

   @Override
   public void studentData()     {
      System.out.println("courses of Student : MCA, MTECH, MBA, BCA");
   }

   public static void main (String[] args)     {
     StudentRecord obj = new StudentRecord();
     obj.collegeDetail();
     obj.studentData();
     obj.hostelDetail();
     obj.studentData();
   }
}
```

Output

```
College Name : AITS
College Grade : A
```

```
University of College : JNTUA
courses of Student : MCA, MTECH, MBA, BCA
Hostel Name : RAMA
Hostel location : AITS
courses of Student : MCA, MTECH, MBA, BCA
```

5. Hybrid Inheritance(Through Interfaces): It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



```
public class ClassA {
   public void dispA()   {
      System.out.println("disp() method of ClassA");
   }
}
public interface InterfaceB {
   public void show();
}
public interface InterfaceC {
   public void show();
}
public class ClassD extends ClassA implements InterfaceB,InterfaceC{
   public void show()   {
      System.out.println("show() method implementation");
   }
   public void dispD()   {
      System.out.println("disp() method of ClassD");
   }
   public static void main(String args[])   {
      ClassD d = new ClassD();
      d.dispD();
      d.show();
   }
}
```

Output :

```
disp() method of ClassD
show() method implementation
```

Using super

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 or object of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

- Super() can be used to refer immediate parent class instance variable.
- Super() can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

super has two general forms.

The first calls the superclass' constructor.

The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of super: super(arg-list);

Here, arg-list specifies any arguments needed by the constructor in the superclass.

super( ) must always be the first statement executed inside a subclass' constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
Animal(){
System.out.println("animal is created");
}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}

class TestSuper{
public static void main(String args[]){
Dog d=new Dog();
}
}
```

OUTPUT:

```
animal is created dog is created
```

To initialize all the property, we are using parent class constructor from child class.

```
 class Person{
int id; String name;
Person(int id,String name){
this.id=id; this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
 this.salary=salary;
}
void display(){
System.out.println(id+" "+name+" "+salary);
}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}
}
```

A Second Use for super

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, member can be either a method or an instance variable.

This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.
class A{
int i;
}
// Create a subclass by extending class A
class B extends A{
int i; // this i hides the i in A
B(int a, int b){
super.i = a;         // i in A
i = b;     // i in B
}

void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper{
public static void main(String args[]) {
```

```
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1 i in subclass: 2
```

Important facts about inheritance in Java

Default superclass: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.

Superclass can only be one: A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by java.

Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Private member inheritance: A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

Java IS-A type of Relationship.

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class SolarSystem {
}
public class Earth extends SolarSystem {
}
public class Mars extends SolarSystem {
}
public class Moon extends Earth {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true:-
SolarSystem the superclass of Earth class.
SolarSystem the superclass of Mars class.
Earth and Mars are subclasses of SolarSystem class.
Moon is the subclass of both Earth and SolarSystem classes.

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
        public static void main(String args[])
        {       SolarSystem s = new SolarSystem();
                Earth e = new Earth();
                Mars m = new Mars();

                System.out.println(s instanceof SolarSystem);
                System.out.println(e instanceof Earth);
                System.out.println(m instanceof SolarSystem);
        }
}
```

Output

```
true
true
true
```

What all can be done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

The inherited fields can be used directly, just like any other fields.

We can declare new fields in the subclass that are not in the superclass.

The inherited methods can be used directly as they are.

We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in the example above, toString() method is overridden).

We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.

We can declare new methods in the subclass that are not in the superclass.

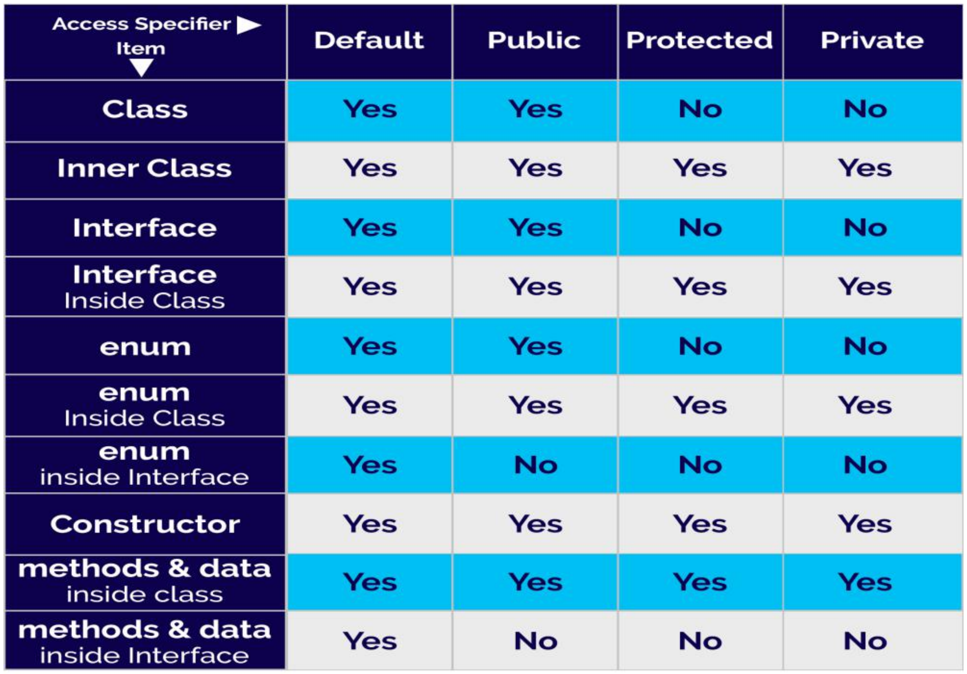
We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Java Access Modifiers

In Java, the access specifiers (also known as access modifiers) used to restrict the scope or accessibility of a class, constructor, variable, method or data member of class and interface. There are four access specifiers, and their list is below.

default (or) no modifier
public
protected
private

In java, we can not employ all access specifiers on everything. The following table describes where we can apply the access specifiers.

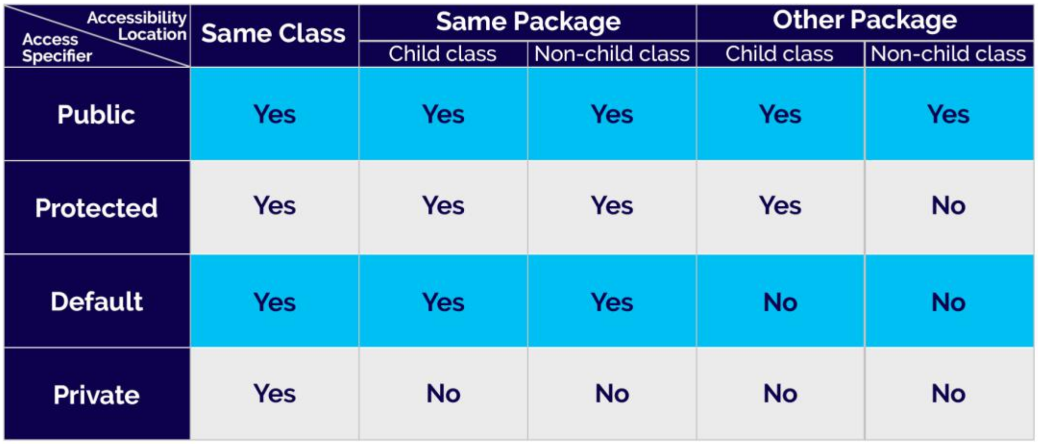


Access Specifier ▶ / Item ▼	Default	Public	Protected	Private
Class	Yes	Yes	No	No
Inner Class	Yes	Yes	Yes	Yes
Interface	Yes	Yes	No	No
Interface Inside Class	Yes	Yes	Yes	Yes
enum	Yes	Yes	No	No
enum Inside Class	Yes	Yes	Yes	Yes
enum inside Interface	Yes	No	No	No
Constructor	Yes	Yes	Yes	Yes
methods & data inside class	Yes	Yes	Yes	Yes
methods & data inside Interface	Yes	No	No	No

www.btechsmartclass.com

Let's look at the following example java code, which generates an error because a class does not allow private access specifier unless it is an inner class.
In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Access control for members of class and interface in java



Accessibility Location / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

www.btechsmartclass.com

- The public members can be accessed everywhere.
- The private members can be accessed only inside the same class.
- The protected members are accessible to every child class (same package or other packages).
- The default members are accessible within the same package but not outside the package.

```
class ParentClass{
        int a = 10;
        public int b = 20;
        protected int c = 30;
        private int d = 40;

        void showData() {
                System.out.println("Inside ParentClass");
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
                System.out.println("d = " + d);
        }
}

class ChildClass extends ParentClass{
        void accessData() {
                System.out.println("Inside ChildClass");
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
                //System.out.println("d = " + d); // private member can't be accessed
        }

}
public class AccessModifiersExample {
        public static void main(String[] args) {
                ChildClass obj = new ChildClass();
                obj.showData();
                obj.accessData();
```

Java Constructors in Inheritance

It is very important to understand how the constructors get executed in the inheritance concept. In the inheritance, the constructors never get inherited to any child class.

In java, the default constructor of a parent class called automatically by the constructor of its child class. That means when we create an object of the child class, the parent class constructor executed, followed by the child class constructor executed.

```
class ParentClass{
        int a;
        ParentClass(){
                System.out.println("Inside ParentClass constructor!");
        }
}
class ChildClass extends ParentClass{
        ChildClass(){
                System.out.println("Inside ChildClass constructor!!");
        }
}
class ChildChildClass extends ChildClass{
        ChildChildClass(){
                System.out.println("Inside ChildChildClass constructor!!");
        }
}
public class ConstructorInInheritance {
        public static void main(String[] args) {
                ChildChildClass obj = new ChildChildClass();
        }
}
```

Output:

```
Inside ParentClass constructor!
Inside ChildClass constructor!
Inside ChildChildClass constructor!
```

However, if the parent class contains both default and parameterized constructor, then only the default constructor called automatically by the child class constructor.

```
class ParentClass{
        int a;
        ParentClass(int a){
                System.out.println("Inside ParentClass parameterized constructor!");
                this.a = a;
        }
        ParentClass(){
                System.out.println("Inside ParentClass default constructor!");
        }
}
class ChildClass extends ParentClass{
        ChildClass(){
                System.out.println("Inside ChildClass constructor!!");
        }
}
public class ConstructorInInheritance {
        public static void main(String[] args) {
                ChildClass obj = new ChildClass();
        }
}
```

Output:

```
Inside ParentClass default constructor!
Inside ChildClass constructor!!
```

The parameterized constructor of parent class must be called explicitly using the super keyword.

Java final keyword

In java, the final is a keyword and it is used with the following things.

- **With variable (to create constant)**
- **With method (to avoid method overriding)**
- **With class (to avoid inheritance)**

Let's look at each of the above.

final with variables

When a variable defined with the **final** keyword, it becomes a constant, and it does not allow us to modify the value. The variable defined with the final keyword allows only a one-time assignment, once a value assigned to it, never allows us to change it again.

```
public class FinalVariableExample {
        public static void main(String[] args) {
                final int a = 10;
                System.out.println("a = " + a);
                a = 100;// Can't be modified
        }
}
```

final with methods

When a method defined with the final keyword, it does not allow it to override. The final method extends to the child class, but the child class can not override or re-define it. It must be used as it has implemented in the parent class.

```
class ParentClass{
        int num = 10;
        final void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }
}
class ChildClass extends ParentClass{
        void showData() {
                System.out.println("Inside ChildClass showData() method");
                System.out.println("num = " + num);
        }
}

public class FinalKeywordExample {
        public static void main(String[] args) {
                ChildClass obj = new ChildClass();
                obj.showData();
        }
}
```

final with class

When a class defined with final keyword, it can not be extended by any other class.

```
final class ParentClass{
        int num = 10;
        void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }
}

class ChildClass extends ParentClass{
}
public class FinalKeywordExample {
        public static void main(String[] args) {
                ChildClass obj = new ChildClass();
        }
}
```

Java Pylymorphism

The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism(Compile Time)

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept.

In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments.

The ad hoc polymorphism implemented within the class only.

```
// Java Program for Method overloading
// By using Different Types of Arguments

// Class 1
// Helper class
class Helper {
        // Method with 2 integer parameters
        static int Multiply(int a, int b)     {

                // Returns product of integer numbers
                return a * b;
        }

        // Method 2
        // With same name but with 2 double parameters
        static double Multiply(double a, double b)          {
                // Returns product of double numbers
                return a * b;
        }
}

// Class 2
// Main class
class GFG {
        // Main driver method
        public static void main(String[] args)      {
                // Calling method by passing
                // input as in arguments
                System.out.println(Helper.Multiply(2, 4));
                System.out.println(Helper.Multiply(5.5, 6.3));
        }
}
```

Output:

```
8
34.65
```

```
// Java program for Method Overloading
// by Using Different Numbers of Arguments

// Class 1
// Helper class
class Helper {

        // Method 1
        // Multiplication of 2 numbers
        static int Multiply(int a, int b)     {
                // Return product
                return a * b;
        }

        // Method 2
        // // Multiplication of 3 numbers
        static int Multiply(int a, int b, int c)        {
                // Return product
                return a * b * c;
        }
}

// Class 2
// Main class
class GFG {
        // Main driver method
        public static void main(String[] args)      {
                // Calling method by passing
                // input as in arguments
                System.out.println(Helper.Multiply(2, 4));
                System.out.println(Helper.Multiply(2, 7, 3));
        }
}
Output:
8
42
```

Pure polymorphism

The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept.

In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.

When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.

The pure polymorphism implemented in the inheritance concept only.

```
// Java Program for Method Overriding

// Class 1
// Helper class
class Parent {
        // Method of parent class
        void Print()      {
                // Print statement
                System.out.println("parent class");
        }
}

// Class 2
// Helper class
class subclass1 extends Parent {
        // Method
        void Print() { System.out.println("subclass1"); }
}

// Class 3
// Helper class
class subclass2 extends Parent {
        // Method
        void Print()      {
                // Print statement
                System.out.println("subclass2");
        }
}

// Class 4
// Main class
class GFG {
        // Main driver method
        public static void main(String[] args)      {
                // Creating object of class 1
                Parent a;
                // Now we will be calling print methods
                // inside main() method
                a = new subclass1();
                a.Print();
                a = new subclass2();
                a.Print();
        }
}
```

Output:

```
subclass1
subclass2
```

Java Method Overriding

The method overriding is the process of re-defining a method in a child class that is already defined in the parent class. When both parent and child classes have the same method, then that method is said to be the overriding method.

The method overriding enables the child class to change the implementation of the method which aquired from parent class according to its requirement.

In the case of the method overriding, the method binding happens at run time. The method binding which happens at run time is known as late binding. So, the method overriding follows late binding.

The method overriding is also known as dynamic method dispatch or run time polymorphism or pure polymorphism.

```
class ParentClass{
        int num = 10;
        void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }

}

class ChildClass extends ParentClass{
        void showData() {
                System.out.println("Inside ChildClass showData() method");
                System.out.println("num = " + num);
        }
}

public class PurePolymorphism {
        public static void main(String[] args) {
                ParentClass obj = new ParentClass();
                obj.showData();

                obj = new ChildClass();
                obj.showData();

        }
}
```

```
Inside ParentClass showData() method
Num=10
Inside ChildClass showData() method
Num=10
```

While overriding a method, we must follow the below list of rules.

- Static methods can not be overridden.
- Final methods can not be overridden.
- Private methods can not be overridden.
- Constructor can not be overridden.
- An abstract method must be overridden.
- Use super keyword to invoke overridden method from child class.
- The return type of the overriding method must be same as the parent has it.
- The access specifier of the overriding method can be changed, but the visibility must increase but not decrease.

For example, a protected method in the parent class can be made public, but not private, in the child class.

If the overridden method does not throw an exception in the parent class, then the child class overriding method can only throw the unchecked exception, throwing a checked exception is not allowed.

If the parent class overridden method does throw an exception, then the child class overriding method can only throw the same, or subclass exception, or it may not throw any exception.

Java Abstract Class

An abstract class is a class that created using abstract keyword. In other words, a class prefixed with abstract keyword is known as an abstract class.

In java, an abstract class may contain abstract methods (methods without implementation) and also non-abstract methods (methods with implementation).

An abstract class can not be instantiated but can be referenced. That means we can not create an object of an abstract class, but base reference can be created.

```
import java.util.*;

abstract class Shape {
        int length, breadth, radius;
        Scanner input = new Scanner(System.in);
        abstract void printArea();
}

class Rectangle extends Shape {
        void printArea() {
                System.out.println("*** Finding the Area of Rectangle ***");
                System.out.print("Enter length and breadth: ");
                length = input.nextInt();
                breadth = input.nextInt();
                System.out.println("The area of Rectangle is: " + length * breadth);
        }
}

class Triangle extends Shape {
        void printArea() {
                System.out.println("\n*** Finding the Area of Triangle ***");
                System.out.print("Enter Base And Height: ");
                length = input.nextInt();
                breadth = input.nextInt();
                System.out.println("The area of Triangle is: " + (length * breadth) / 2);
        }
}

class Cricle extends Shape {
        void printArea() {
                System.out.println("\n*** Finding the Area of Cricle ***");
                System.out.print("Enter Radius: ");
                radius = input.nextInt();
                System.out.println("The area of Cricle is: " + 3.14f * radius * radius);
        }
}

public class AbstractClassExample {
        public static void main(String[] args) {
                Rectangle rec = new Rectangle();
                rec.printArea();

                Triangle tri = new Triangle();
                tri.printArea();
```

```
                    Cricle cri = new Cricle();
                    cri.printArea();
          }
}
```

Output:

```
*** Finding the Area of Rectangle ***
Enter length and breadth: 2 4
The area of Rectangle is: 8
*** Finding the Area of Triangle ***
Enter Base And Height: 4 5
The area of Triangle is: 10
*** Finding the Area of Cricle ***
Enter Radius: 6
The area of Cricle is: 113.04
```

In the above example program, the child class objects are created to invoke the overridden abstract method. But we may also create base class reference and assign it with child class instance to invoke the same. The main method of the above program can be written as follows that produce the same output.

```
public static void main(String[] args) {
                    Shape obj = new Rectangle();  //Base class reference to Child class instance
                    obj.printArea();

                    obj = new Triangle();
                    obj.printArea();

                    obj = new Cricle();
                    obj.printArea();
```

An abstract class must follow the below list of rules.
An abstract class must be created with abstract keyword.
An abstract class can be created without any abstract method.
An abstract class may contain abstract methods and non-abstract methods.
An abstract class may contain final methods that can not be overridden.
An abstract class may contain static methods, but the abstract method can not be static.
An abstract class may have a constructor that gets executed when the child class object created.
An abstract method must be overridden by the child class, otherwise, it must be defined as an abstract class.
An abstract class can not be instantiated but can be referenced.

Java Object Class

In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the java.lang package.
Every class in the java programming language is a subclass of Object class by default.

The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.
Methods of Object class
The following table depicts all built-in methods of Object class in java.

Method	Description	Return Value
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int
equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	eturns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

1. toString()
Syntax:
public String toString()
This function is used to extract the String representation of an object. This String representation can be easily modified according to the need For e.g., in case there is a class Office that needs that its head of that branch must be displayed with location whenever one calls this function. Thus representation depends entirely on the object and useful while debugging.
2. hashCode()
Syntax:
public int hashCode()
This function returns the hexadecimal representation of the object’s memory address, which is helpful in the case of equating 2 objects as 2 objects are said to equal only if their hashcode matched, as per

implementation present in the Object class. But if one overrides the equals() method of Object class, super implementation becomes inactive for that class's objects.

3. equals(Object obj)

Syntax:

public boolean equals(Object obj)

This method is used to override the default implementation of the equals method to compare 2 objects using their hashcode and returns true or false accordingly. In case you want to implement your own logic to compare 2 objects of your class, you must override the equals method.

4. getClass()

Syntax:

public final Class getClass()

This method returns the reference of a Class object that helps to retrieve various information about the current class. Below are examples of instance methods of Class

- **getSimpleName():** Returns the name of the class.
- **getSuperClass():** Returns the reference of the superclass of the specified class.
- **getInterfaces():** Returns the array of references of Class type for all the interfaces that have been implemented in a specified class.
- **isAnnotation():** Returns True is the specified class is an annotation otherwise false.
- **getFields():** Returns the list of fields of the specified class.
- **getMethods():** Returns the list of instance methods of the class.

5. finalize()

Syntax:

protected void finalize() throws Throwable

This method is called whenever JVM depicts that there is no more reference exit to that object so that garbage collection can be performed. There is nothing defined in finalize function in the Object class and returns nothing. Thus the subclasses can implement it to perform some action but must not rely on when it will be invoked as the thread that invokes it does not hold a user-visible authentication lock. Any exception occurs while finalize() halts its execution; thus, it must be handled with precautions.

6. clone()

Syntax:

protected Object clone() throws CloneNotSupportedException

This method of Object class is meant to be overridden in a subclass in case Cloneable Interface is implemented in it, and this method is used to clone, i.e. creating a copy of that object with values in member variables, using aobj.clone() syntax. In case one class does not implement the Cloneable interface, CloneNotSupportedException is thrown.

By default, this method checks if the Cloneable interface has been implemented in this case, but in case you want to override this method according to your logic, you can easily do this using the following syntax:-

protected Object clone() throws CloneNotSupportedException

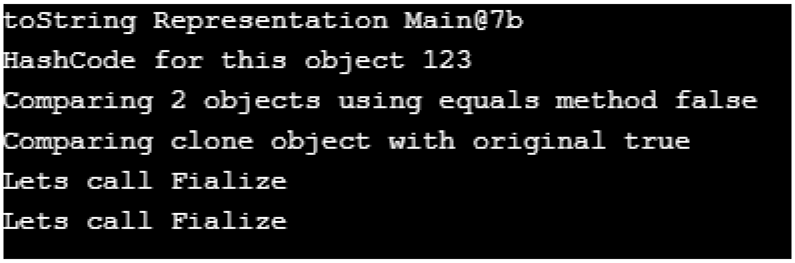
or

public Object clone() throws CloneNotSupportedException

Example:

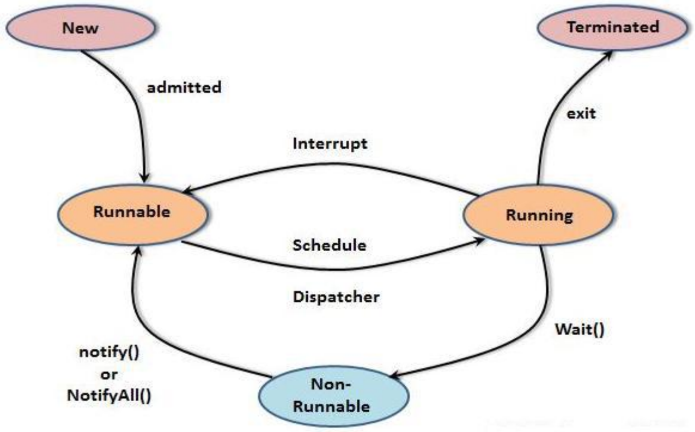
```
package Try;
public class Desk implements Cloneable{
private int id;
private String Mid;
Desk(int id,String mid){
this.id=id;
this.Mid = mid;
}
@Override
public String toString()
{
return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
@Override
public int hashCode()
{
return id;
}
public boolean equals(Desk dd){
return this.id == dd.id;
}
@Override
protected void finalize()
{
System.out.println("Lets call Fialize");
}
@Override
protected Desk clone() {
return this;
}
public static void main(String[] args){
Desk d1=new Desk(123,"344");
Desk d2=new Desk(234,"344");
System.out.println("toString Representation " + d1.toString());
System.out.println("HashCode for this object "+d1.hashCode());
System.out.println("Comparing 2 objects using equals method "+ d1.equals(d2));
Desk d3=d2.clone(); // cloning d2 object
System.out.println("Comparing clone object with original "+d2.equals(d3));
d1=d2=d3=null;
System.gc();
}
}
```

Output:



```
toString Representation Main@7b
HashCode for this object 123
Comparing 2 objects using equals method false
Comparing clone object with original true
Lets call Fialize
Lets call Fialize
```

Below three methods of Object class are used in case you need to implement multithreading behavior and need synchronization between different threads.



7. wait()

This method is used to put the current thread in the waiting state until any other thread notifies. The time needs to be specified in the function in milliseconds for which you need to resume the thread execution. There are three definitions for this function as given below:

- public final void wait()
- public final void wait(long timeout)
- public final void wait(long timeout, int nanos)

Note: InterruptedException is thrown by this method.

8. notify()

Syntax:

public final void notify()

9. notifyAll()

This method is used to wake up all the threads waiting in the waiting queue.

Syntax:

public final void notifyAll()

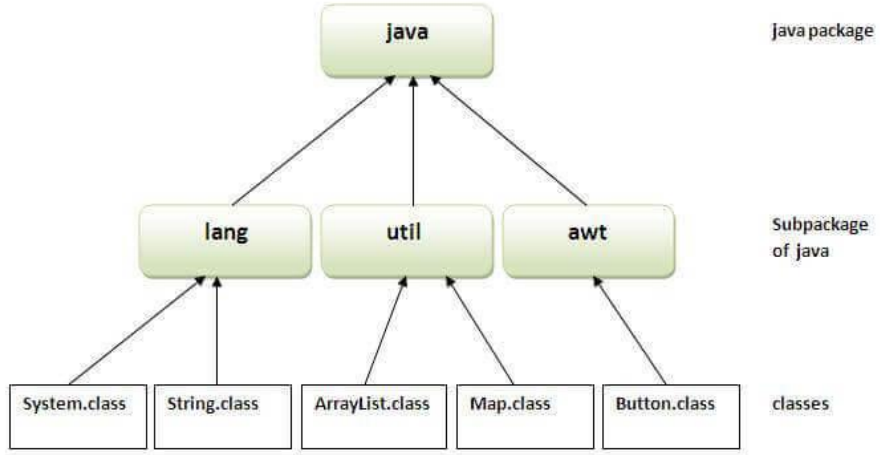
Java Package

A java package is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form,

- built-in package and
- user-defined package.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



Simple example of java package
The package keyword is used to create a package in java.

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

How to compile java package
If you are not using any IDE, you need to follow the syntax given below:
Syntax: javac -d directory javafilename

For example
javac -d . Simple.java

How to run java package program

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

How to access package from another package?

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

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

1.) Using packagename.*

If you use package.* 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 interface of another package accessible to the current package.

Example of package that import the packagename.*

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

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

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

Output:Hello

2) Using packagename.classname

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

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

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

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

3) Using fully qualified name

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

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

Example of package by import fully qualified name

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

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

Subpackage in java

Package inside the package is called the subpackage. It should be created to categorize the package further.

Example of Subpackage

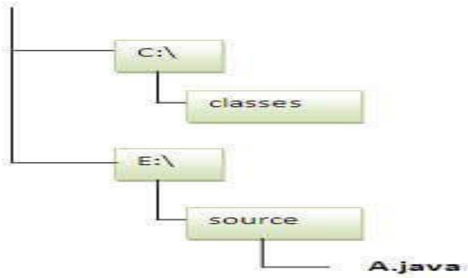
```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

How to send the class file to another directory or drive?
There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



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

To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file reside

```
e:\sources> set classpath=c:\classes;.;
e:\sources> java mypack.Simple
```

Another way to run this program by -classpath switch of java:
The -classpath switch can be used with javac and java tool.
To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

Temporary
By setting the classpath in the command prompt
By -classpath switch
Permanent
By setting the classpath in the environment variables
By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
//save as C.java otherwise Compilte Time Error

class A{}
class B{}
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java

package javatpoint;
public class A{}
//save as B.java

package javatpoint;
public class B{}
```

Interface in Java

- Interfaces are one of the core concepts of Java programming which are majorly used in Java design patterns.
- An interface provides specifications of what a class should do or not and how it should do. An interface in Java basically has a set of methods that class may or may not apply.
- It also has capabilities to perform a function. The methods in interfaces do not contain any body.
- These abstract methods are implemented by classes before accessing them.
- An interface in Java is a mechanism which we mainly use to achieve abstraction and multiple inheritances in Java.
- An interface provides a set of specifications that other classes must implement.
- We can implement multiple Java Interfaces by a Java class. All methods of an interface are implicitly public and abstract. The word abstract means these methods have no method body, only method signature.
- Java Interface also represents the IS-A relationship of inheritance between two classes.
- An interface can inherit or extend multiple interfaces.
- We can implement more than one interface in our class.
- Since Java 8, we can have static and default methods in an interface.
- Since Java 9, we can also include private methods in an interface.

Why use Java interface?

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.

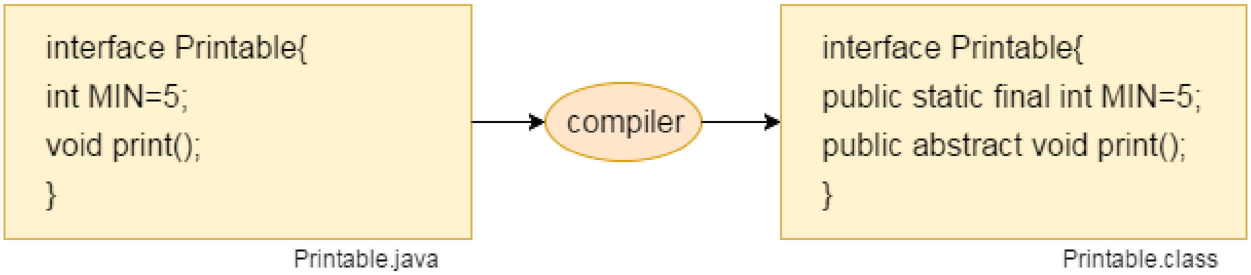
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the 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 <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

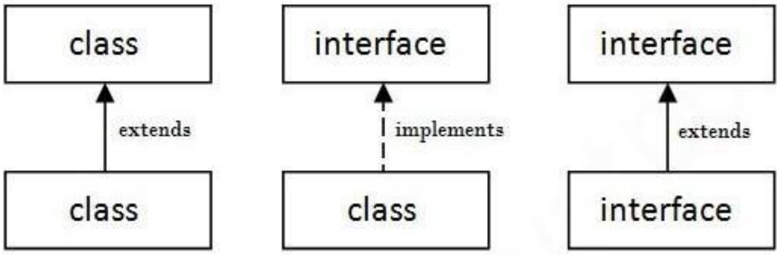


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.
File: TestInterface2.java

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
```

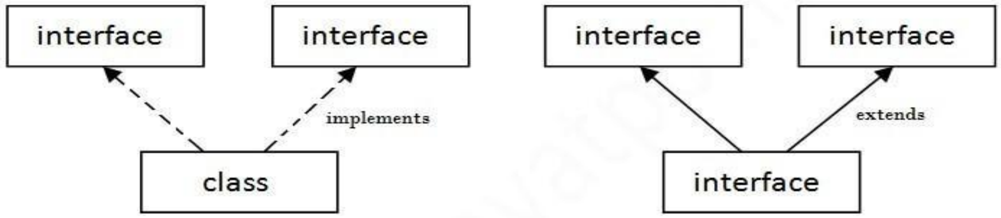
```
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

```
// Java program to demonstrate
// the multiple inheritance
// in interface

// Interface to implement the
// addition and subtraction methods
interface Add_Sub {
	public void add(double x, double y);
	public void subtract(double x, double y);
}

// Interface to implement the
// multiplication and division
interface Mul_Div {
        public void multiply(double x, double y);
        public void divide(double x, double y);
}

// Calculator interface which extends
// both the above defined interfaces
interface Calculator extends Add_Sub, Mul_Div {
        public void printResult(double result);
}

// Calculator class which
// implements the above
// interface
public class MyCalculator implements Calculator {

        // Implementing the addition
        // method
        public void add(double x, double y)
        {
                double result = x + y;
                printResult(result);
        }

        // Implementing the subtraction
        // method
        public void subtract(double x, double y)
        {
                double result = x - y;
                printResult(result);
        }

        // Implementing the multiplication
        // method
        public void multiply(double x, double y)
        {
                double result = x * y;
                printResult(result);
        }

        // Implementing the division
        // method
        public void divide(double x, double y)
        {
                double result = x / y;
                printResult(result);
        }

        // Implementing a method
        // to print the result
        public void printResult(double result)
        {
                System.out.println(
                        "The result is : " + result);
        }

        // Driver code
        public static void main(String args[])
        {

                // Creating the object of
                // the calculator
                MyCalculator c = new MyCalculator();
                c.add(5, 10);
                c.subtract(35, 15);
                c.multiply(6, 9);
                c.divide(45, 6);
        }
}
```

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Output:

```
drawing rectangle
```

default method

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Output:

```
drawing rectangle
27
```

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, Serializable , Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
/How Serializable interface is written?
public interface Serializable{
}
```

Java Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Syntax of nested interface which is declared within the interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Syntax of nested interface which is declared within the class

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Example of nested interface which is declared within the interface

In this example, we will learn how to declare the nested interface and how we can access it.

TestNestedInterface1.java

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Output:

```
hello nested interface
```

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates a public and static interface as displayed below:

```
public static interface Showable$Message
{
  public abstract void msg();
}
```

Example of nested interface which is declared within the class

Let's see how we can define an interface inside the class and how we can access it.

TestNestedInterface2.java

```
class A{
  interface Message{
   void msg();
  }
}

```

```
class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

Output:

```
hello nested interface
```

Can we define a class inside the interface?

Yes, if we define a class inside the interface, the Java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
interface M{
  class A{}
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

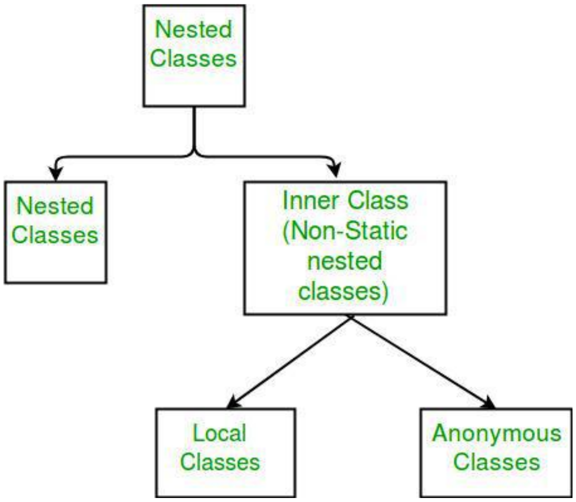
Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class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

like private, protected, etc.	

9)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Understanding Java Nested Classes and Java Inner Classes



In Java programming, nested and inner classes often go hand in hand. A class that is defined within another class is called a nested class. An inner class, on the other hand, is a non-static type, a particular specimen of a nested class. This article attempts to elaborate on these two ideas of designing classes.
Java Nested Class

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

Non-static nested class (inner class)

- Member inner class
- Anonymous inner class
- Local inner class

Static nested class

Type	Description

Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

A nested class refers to the idea of defining one class inside another class. The scope of the nested class is bounded by the scope of the enclosing classes. That means if *Pearl* is a class defined within, say, the *Oyster* class, the object of *Pearl* cannot exist without the existence of an *Oyster* object and the class within the class, *Pearl*, can access all the members of the *Oyster* class, including private members. But, the external class, such as *Oyster*, cannot access the members of *Pearl* class. Here's an example of the classes' use:

```
package ocean;

public class Oyster {

  class Pearl{
    String pearl="member";
    private String privatePearl="private member";
    protected String protectedPearl="protected member";
    public String publicPearl="public member";

    public Pearl() {
      oyster="sdfsds";           // OK!
      privateOyster="sdcdscs";    // OK!
      protectedOyster="svsfdf";   // OK!
      publicOyster="sdfdfsd";     // OK!
    }
  }

  String oyster="member";
  private String privateOyster="private member";
  protected String protectedOyster="protected member";
  public String publicOyster="public member";

  public Oyster(){
    pearl="sddsfsd";             // not OK!
    privatePearl="sdfdfdfs";       // not OK!
    protectedPearl="svsfdf";       // not OK!
    publicPearl="sdfdfsd";         // not OK!
  }
}
```

A nested class also can be declared locally within a block as well, such as seen in the following example:

```
package ocean;

public class Oyster {

  public void testfunc(){

    class A{
      private String s="A";
      public String getA(){
        return s;
      }
    }

    {
      class B{
        private String s="B";
        public String getB(){
          return s;
        }
      }

      B b=new B();
      System.out.println(b.getB());
    }

    A a=new A();
    System.out.println(a.getA());
  }
  public static void main(String[] args) {
    Oyster os=new Oyster();
    os.testfunc();
  }
}
```

Now, the nested class can be of two types, a static nested class and a non-static nested class. A nested class, Pearl, which is declared static with the static modifier, cannot access the members of the enclosing class Oyster directly because the nested class Pearl is now static. As a result, it needs an object of the Oyster class to access the members of its enclosing class. Static nested classes are very rarely used.

```
package ocean;

public class Oyster {
  static class Pearl{
    String pearl="member";
    private String privatePearl="private member";
    protected String protectedPearl="protected member";
    public String publicPearl="public member";

    public Pearl() {
      oyster="sdfsds";              // Not OK!
      privateOyster="sdcdscs";       // Not OK!
      protectedOyster="svsfdf";      // Not OK!
      publicOyster="sdfdfsd";        // Not OK!

      Oyster os=new Oyster();

      os.oyster="sdfsds";            // OK!
      os.privateOyster="sdcdscs";     // OK!
      os.protectedOyster="svsfdf";    // OK!
      os.publicOyster="sdfdfsd";      // OK!

    }
  }


  String oyster="member";
  private String privateOyster="private member";
  protected String protectedOyster="protected member";
  public String publicOyster="public member";

  // ...

}
```

Java Member Inner class

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

Java Member Inner Class Example

In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

TestMemberOuter1.java

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Output:

```
data is 30
```

How to instantiate Member Inner class in Java?

An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.

The general form of syntax to create an object of the member inner class is as follows:

Syntax:

OuterClassReference.new MemberInnerClassConstructor();

Example:

obj.new Inner();

Here, OuterClassReference is the reference of the outer class followed by a dot which is followed by the new operator.

Internal working of Java member inner class

The java compiler creates two class files in the case of the inner class. The class file name of the inner class is "Outer$Inner". If you want to instantiate the inner class, you must have to create the instance of the outer class. In such a case, an instance of inner class is created inside the instance of the outer class.

Internal code generated by the compiler

The Java compiler creates a class file named Outer$Inner in this case. The Member inner class has the reference of Outer class that is why it can access all the data members of Outer class including private.

```
import java.io.PrintStream;
class Outer$Inner
{
    final Outer this$0;
    Outer$Inner()
    {   super();
        this$0 = Outer.this;
    }
    void msg()
    {
        System.out.println((new StringBuilder()).append("data is ")
                .append(Outer.access$000(Outer.this)).toString());
    }
}
```

Java Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class. In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

- Class (may be abstract or concrete).
- Interface

Java anonymous inner class example using class

TestAnonymousInner.java

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Output:

```
nice fruits
```

Internal working of given code

```
Person p=new Person(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner$1 extends Person
{
  TestAnonymousInner$1(){}
  void eat()
   {
      System.out.println("nice fruits");
   }
}
```

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Output:

```
nice fruits
```

Internal working of given code

It performs two main tasks behind this code:

```
Eatable p=new Eatable(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created, but its name is decided by the compiler, which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p', a reference variable of the Eatable type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner1$1 implements Eatable
{
TestAnonymousInner1$1(){}
void eat(){System.out.println("nice fruits");}
}
```

Java Local inner class

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

LocalInner1.java

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Output:

```
30
```

Internal class generated by the compiler

In such a case, the compiler creates a class named Simple$1Local that has the reference of the outer class.

```
import java.io.PrintStream;
class localInner1$Local
{
    final localInner1 this$0;
    localInner1$Local()
    {
        super();
        this$0 = Simple.this;
    }
    void msg()
    {
        System.out.println(localInner1.access$000(localInner1.this));
    }
}
```

Rule: Local variables can't be private, public, or protected.

Rules for Java Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

Example of local inner class with local variable

LocalInner2.java

```
class localInner2{
 private int data=30;//instance variable
 void display(){
  int value=50;//local variable must be final till jdk 1.7 only
  class Local{
   void msg(){System.out.println(value);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner2 obj=new localInner2();
  obj.display();
 }
}
```

Output:

```
50
```

Java static nested class

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of the outer class, including private.
- The static nested class cannot access non-static (instance) data members or

Java static nested class example with instance method

TestOuter1.java

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

Output:

```
data is 30
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of the Outer class because the nested class is static and static properties, methods, or classes can be accessed without an object.

Internal class generated by the compiler

```
import java.io.PrintStream;
```

```
static class TestOuter1$Inner
{
TestOuter1$Inner(){}
void msg(){
System.out.println((new StringBuilder()).append("data is ")
.append(TestOuter1.data).toString());
}
}
```

Java static nested class example with a static method

If you have the static member inside the static nested class, you don't need to create an instance of the static nested class.

TestOuter2.java

```
public class TestOuter2{
  static int data=30;
  static class Inner{
   static void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter2.Inner.msg();//no need to create the instance of static nested class
  }
}
```

Output:

```
data is 30
```

Java local class

1. Java local class :
2. Scope of java Local classes
3. Accessing members in local class
4. Shadowing and java local class

1. Java local class :

Sometimes we can declare a class inside any block such as instance block or constructor or method or if block, such type of inner classes are called local inner classes or local classes. You can define a java local class inside any block. For example, you can define a local class in a method body, a for loop, or an if clause.

The main purpose of local inner class is to define logic that required instantly to use in local scope.

```
class Outer {
// a Class inside Constructor
Outer(){
class ConstructorLocal {
}
}
// a class inside instance block
{
class Local{
}
}
// a class inside static block
static {
class StaticLocal{
}
}
// a class inside method
public void add(int a, int b) {
class MethodLocal {
}
// a class inside if block
if(a > b) {
class Local{
}
}
}
}
```

2. Scope of java Local classes

We can access local class only within the method or block where we declared it. That is from outside of the block we can’t access. As the scope of local inner classes is very less, this type of inner classes are most rarely used type of inner classes. Mostly this kind of use cases we can see in multi threading programming.

Example

```
class Outer {
public void methodOne() {
class Inner{
public void sum(int i,int j) {
System.out.println("The sum:"+(i+j));
}
}
// out side of this method Local Inner class can't be accessed
Inner i = new Inner();
i.sum(10,20);
i.sum(100,200);
}
public static void main(String[] args) {
//Inner i = new Inner(); can not access
//new Outer().new Inner(); can not access
new Outer().methodOne();
}
}
```

Output :

```
The sum:30
The sum:300
```

3. Accessing members in local class

1. If we are declaring java local class inside instance block or method then we can access both static and non static members of outer class directly.
2. But if we are declaring inner class inside static method or block then we can access only static members of outer class directly and we can’t access instance members directly.
3. We can access constants directly inside local classes and we can declare also. A final variable is a constant, so that we can access final members directly.
4. Only allowed modifiers for local classes are final, abstract and strictfp.

Example

```
class Outer{
int x = 10;
static int y = 20;
static final int z = 30;
static {
// Defining class in static area
class Inner {
static int m = 40;// Cant declare static members in non static inner type
static final int n = 50;
public void methodTwo() {
System.out.println(x);// can not access instance members
System.out.println(y);//20
System.out.println(z);//30
}
}
}
public void methodOne() {
class Inner {
static int m = 40;// Cant declare static members in non static inner type
static final int n = 50;
public void methodTwo() {
System.out.println(x);//10
System.out.println(y);//20
System.out.println(n);//50
}
}
Inner i=new Inner();
i.methodTwo();
}
public static void main(String[] args) {
new Outer().methodOne();
}
}
```

4. Shadowing and java local class

If a declaration of a type such as a member variable or a parameter name in a particular scope has the same name as another declaration in the enclosing scope, then the declaration shadows the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone. The following example demonstrates this.

```
public class LocalClassShadowingExample {
int x = 10;
int y = 20;
static final int z = 30;
public void methodOne() {
class Inner {
//static int m = 40;// Cant declare static members in non static inner type
int x = 90;
static final int z = 100;
public void methodTwo(int x) {
System.out.println("x=> "+x);//111
System.out.println("y=> "+y);//20
System.out.println("z=> "+z);//100
System.out.println("this.x=> "+this.x);//90
System.out.println("LocalClassShadowingExample.this.x=> "+LocalClassShadowingExample.this.x);
System.out.println("LocalClassShadowingExample.z=> "+LocalClassShadowingExample.z);
}
}
Inner i=new Inner();
i.methodTwo(111);
}
public static void main(String[] args) {
new LocalClassShadowingExample().methodOne();
}
}
```

Output :

```
x=> 111
y=> 20
z=> 100
this.x=> 90
LocalClassShadowingExample.this.x=> 10
LocalClassShadowingExample.z=> 30
```

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name :** Object Oriented Programming Through JAVA	**AY:** 2021-2022
Unit 3 (Exception Handling,)			

EXCEPTION HANDLING

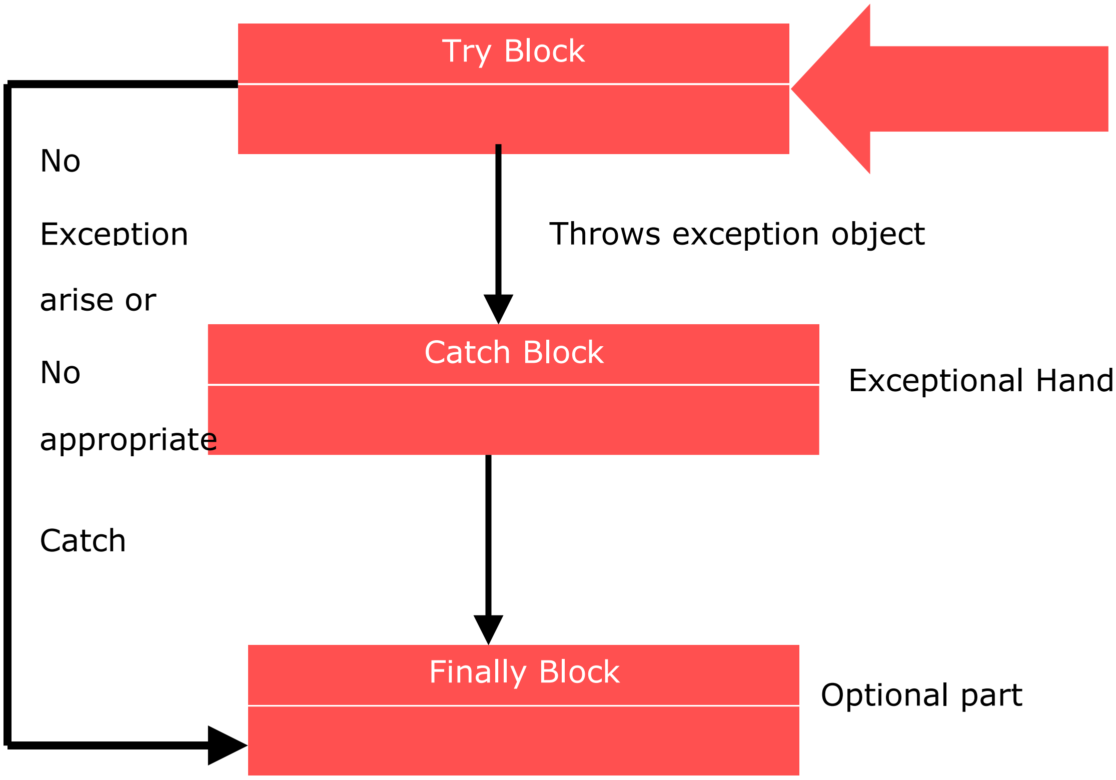
Introduction

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction.

  Or
- An abnormal condition that disrupts Normal program flow.
- There are many cases where abnormal conditions happen during program execution, such as
  - Trying to access an out - of – bounds array elements.
  - The file you try to open may not exist.
  - The file we want to load may be missing or in the wrong format.
  - The other end of your network connection may be non – existence.
- If these cases are not prevented or at least handled properly, either the program will be aborted abruptly, or the incorrect results or status will be produced.
- When an error occurs with in the java method, the method creates an exception object and hands it off to the runtime system.
- The exception object contains information about the exception including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error.
- In java creating an exception object and handling it to the runtime system is called throwing an exception.
- Exception is an object that is describes an exceptional (i.e. error) condition that has occurred in a piece of code at run time.
- When a exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- System generated exceptions are automatically thrown by the Java runtime system

```
General form of Exception
Handling block try {
// block of code to monitor for errors
}
        catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
```

```
            catch (ExceptionType2 exOb) {
            // exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```



- By using exception to managing errors, Java programs have have the following advantage over traditional error management techniques:
  - Separating Error handling code from regular code.
  - Propagating error up the call stack.
  - Grouping error types and error differentiation.

For Example:

```
class Exc0 {
public static void main(String
args[]) { int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: /
by zero at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

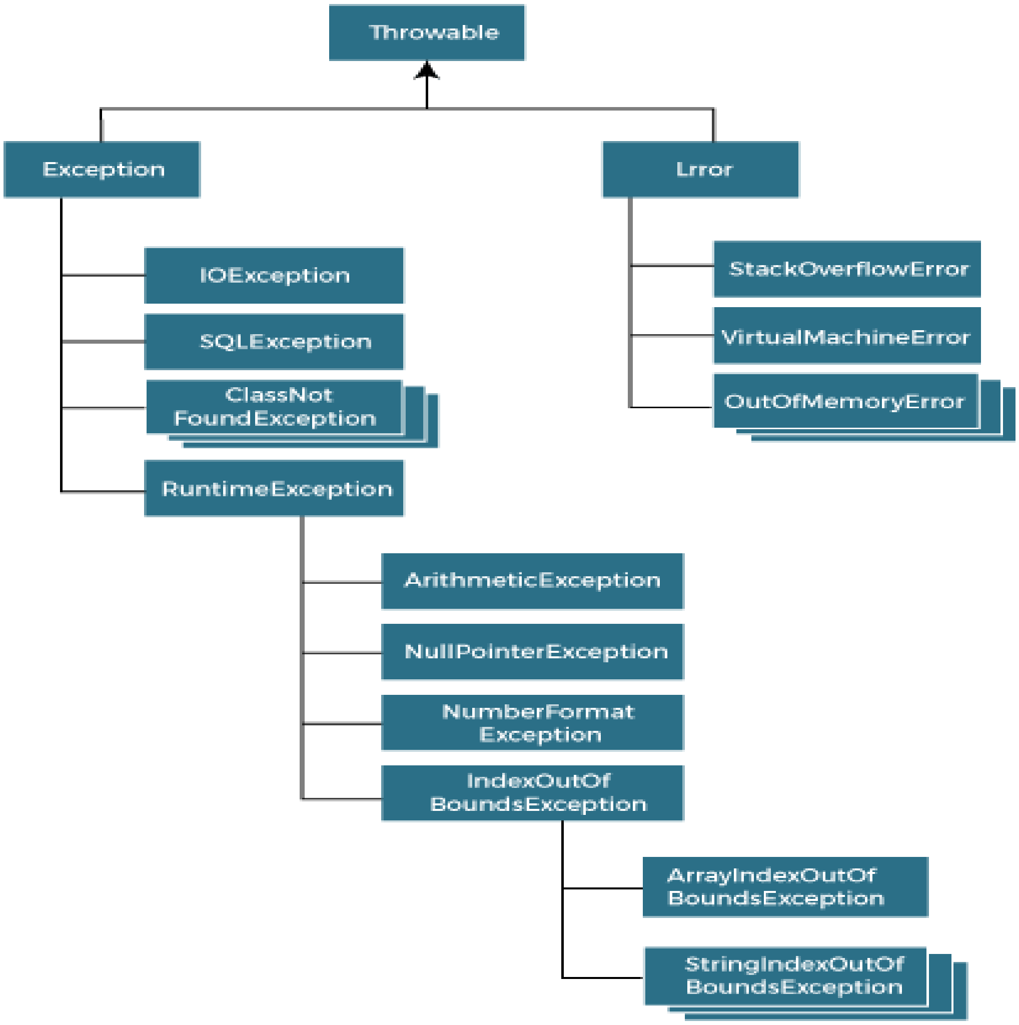
The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.



Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```java
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

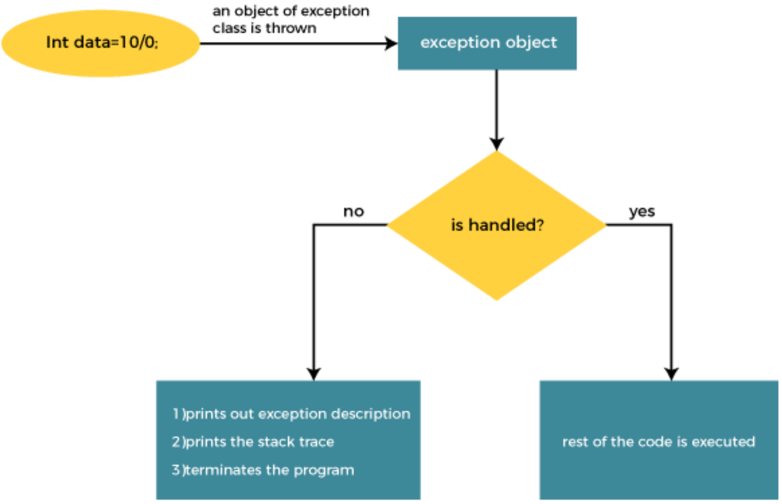
```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

TryCatchExample1.java

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```java
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

TryCatchExample3.java

```java
public class TryCatchExample3 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
                    // if exception occurs, the remaining statement will not exceute
        System.out.println("rest of the code");
        }
            // handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }

    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
```

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

TryCatchExample4.java

```java
public class TryCatchExample4 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception by using Exception class
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example 5

Let's see an example to print a custom message on exception.

TryCatchExample5.java

```java
public class TryCatchExample5 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
             // handling the exception
        catch(Exception e)
        {
                  // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }

}
```

Output:

```
Can't divided by zero
```

Example 6

Let's see an example to resolve the exception in a catch block.

TryCatchExample6.java

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output:

```
25
```

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

TryCatchExample7.java

```
public class TryCatchExample7 {

    public static void main(String[] args) {

        try
        {
        int data1=50/0; //may throw exception

        }
            // handling the exception
        catch(Exception e)
        {
           // generating the exception in catch block
        int data2=50/0; //may throw exception

        }
    System.out.println("rest of the code");
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

TryCatchExample8.java

```
public class TryCatchExample8 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception

        }
        // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException

    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println(e);
    }
    System.out.println("rest of the code");
 }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Example 9

Let's see an example to handle another unchecked exception.

TryCatchExample9.java

```
public class TryCatchExample9 {

  public static void main(String[] args) {
    try
    {
    int arr[]= {1,3,5,7};
    System.out.println(arr[10]); //may throw exception
    }
        // handling the array exception
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println(e);
    }
    System.out.println("rest of the code");
 }

}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
```

```
rest of the code
```

Example 10

Let's see an example to handle checked exception.

TryCatchExample10.java

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10 {

    public static void main(String[] args) {


        PrintWriter pw;
        try {
            pw = new PrintWriter("jtp.txt"); //may throw exception
            pw.println("saved");
        }
// providing the checked exception handler
 catch (FileNotFoundException e) {

            System.out.println(e);
        }
    System.out.println("File saved successfully");
    }
}
```

Output:

```
File saved successfully
```

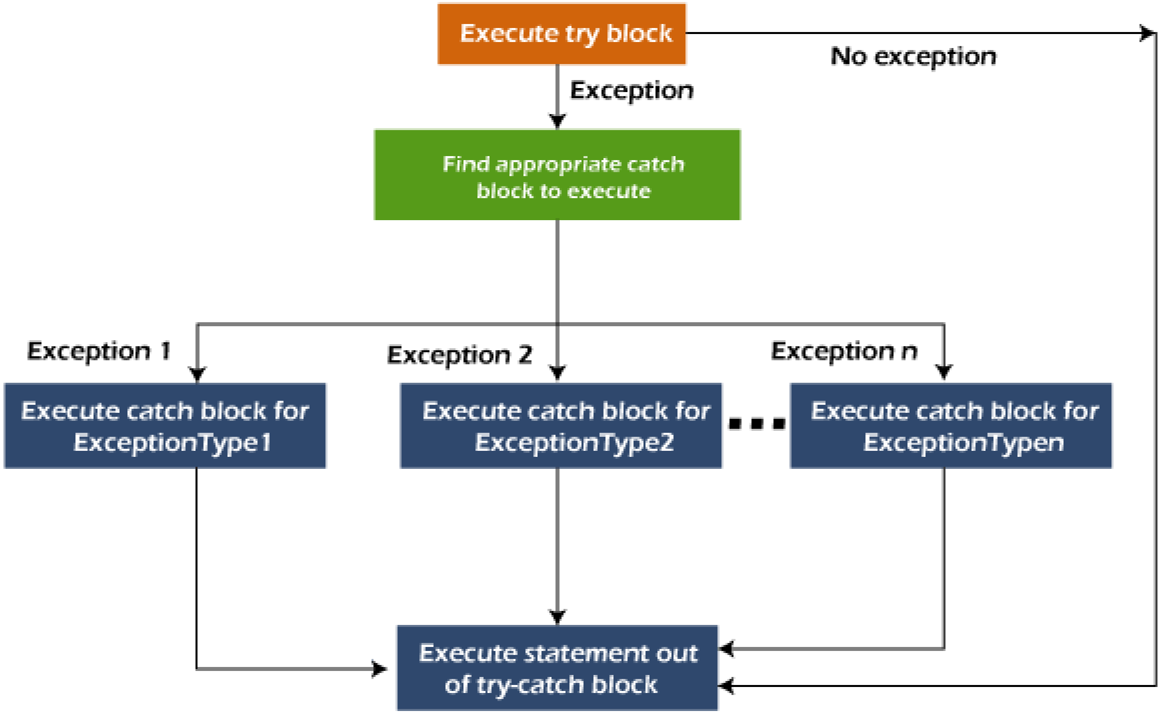
Java Multi-catch block

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.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- **All catch blocks must be ordered from most specific to most general,** i.e. catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType1
Execute catch block for ExceptionType2
...
Execute catch block for ExceptionTypen
Execute statement out of try-catch block

Example 1

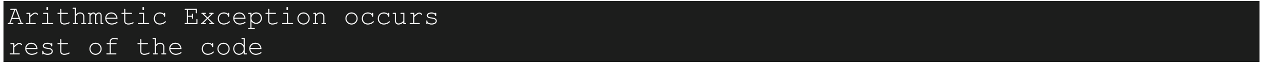
Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
public class MultipleCatchBlock1 {

    public static void main(String[] args) {

            try{
                int a[]=new int[5];
                a[5]=30/0;
               }
```

```
             catch(ArithmeticException e)
               {
                System.out.println("Arithmetic Exception occurs");
               }
             catch(ArrayIndexOutOfBoundsException e)
               {
                System.out.println("ArrayIndexOutOfBounds Exception occurs");
               }
             catch(Exception e)
               {
                System.out.println("Parent Exception occurs");
               }
             System.out.println("rest of the code");
    }
}
```

Output:



```
Arithmetic Exception occurs
rest of the code
```

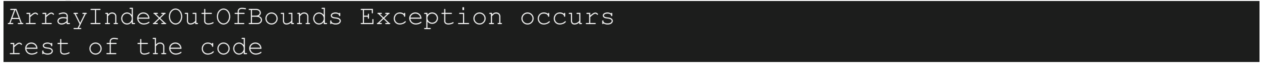
Example 2

MultipleCatchBlock2.java

```
public class MultipleCatchBlock2 {

    public static void main(String[] args) {

           try{
                int a[]=new int[5];

                System.out.println(a[10]);
               }
               catch(ArithmeticException e)
                  {
                   System.out.println("Arithmetic Exception occurs");
                  }
               catch(ArrayIndexOutOfBoundsException e)
                  {
                   System.out.println("ArrayIndexOutOfBounds Exception occurs");
                  }
```

```
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
    }
}
```

Output:



```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

MultipleCatchBlock3.java

```
public class MultipleCatchBlock3 {

    public static void main(String[] args) {

           try{
               int a[]=new int[5];
               a[5]=30/0;
               System.out.println(a[10]);
              }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
    }
}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

MultipleCatchBlock4.java

```
public class MultipleCatchBlock4 {

    public static void main(String[] args) {

           try{
                String s=null;
                System.out.println(s.length());
               }
               catch(ArithmeticException e)
                  {
                   System.out.println("Arithmetic Exception occurs");
                  }
               catch(ArrayIndexOutOfBoundsException e)
                  {
                   System.out.println("ArrayIndexOutOfBounds Exception occurs");
                  }
               catch(Exception e)
                  {
                   System.out.println("Parent Exception occurs");
                  }
               System.out.println("rest of the code");
    }
}
```

Output:

```
Parent Exception occurs
rest of the code
```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

MultipleCatchBlock5.java

```
class MultipleCatchBlock5{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
 }
}
```

Output:

```
Compile-time error
```

Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
//main try block
try
{
    statement 1;
    statement 2;
//try catch block within another try block
    try
    {
        statement 3;
        statement 4;
//try catch block within nested try block
        try
        {
            statement 5;
            statement 6;
     }
        catch(Exception e2)
        {
//exception message
        }

    }
    catch(Exception e1)
    {
//exception message
    }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Java Nested try Example

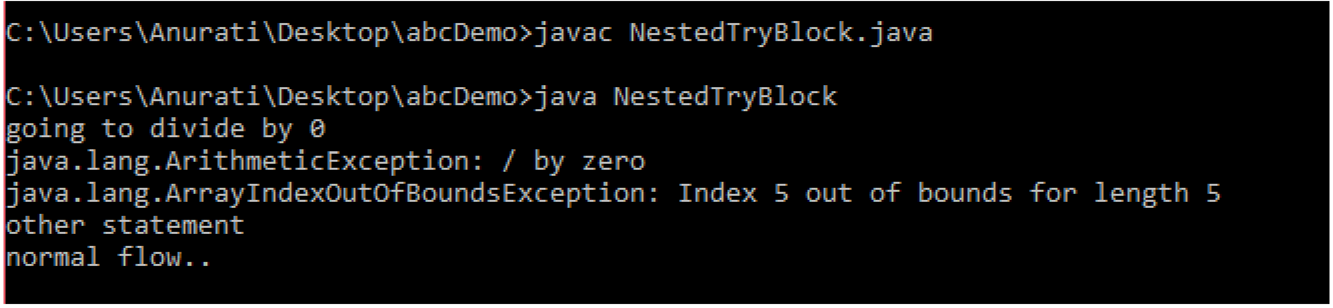
Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

NestedTryBlock.java

```
public class NestedTryBlock{
 public static void main(String args[]){
 //outer try block
  try{
  //inner try block 1
    try{
     System.out.println("going to divide by 0");
     int b =39/0;
   }
    //catch block of inner try block 1
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }


    //inner try block 2
    try{
    int a[]=new int[5];

    //assigning the value out of array bounds
     a[5]=4;
     }

    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
       System.out.println(e);
    }


    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  {
    System.out.println("handled the exception (outer catch)");
  }

  System.out.println("normal flow..");
 }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

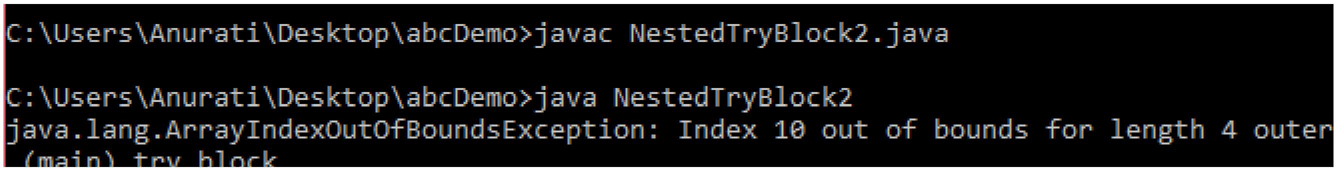
Example 2

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. **It is termed as nesting**.

NestedTryBlock.java

```
public class NestedTryBlock2 {

   public static void main(String args[])
   {
      // outer (main) try block
      try {

            //inner try block 1
            try {

                // inner try block 2
                try {
                    int arr[] = { 1, 2, 3, 4 };

                    //printing the array element out of its bounds
                    System.out.println(arr[10]);
                }

                // to handles ArithmeticException
                catch (ArithmeticException e) {
                    System.out.println("Arithmetic exception");
                    System.out.println(" inner try block 2");
                }
            }

            // to handle ArithmeticException
            catch (ArithmeticException e) {
                System.out.println("Arithmetic exception");
                System.out.println("inner try block 1");
            }
        }

        // to handle ArrayIndexOutOfBoundsException
        catch (ArrayIndexOutOfBoundsException e4) {
            System.out.print(e4);
            System.out.println(" outer (main) try block");
        }
        catch (Exception e5) {
            System.out.print("Exception");
            System.out.println(" handled in main try-block");
        }
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
 (main) try block
```

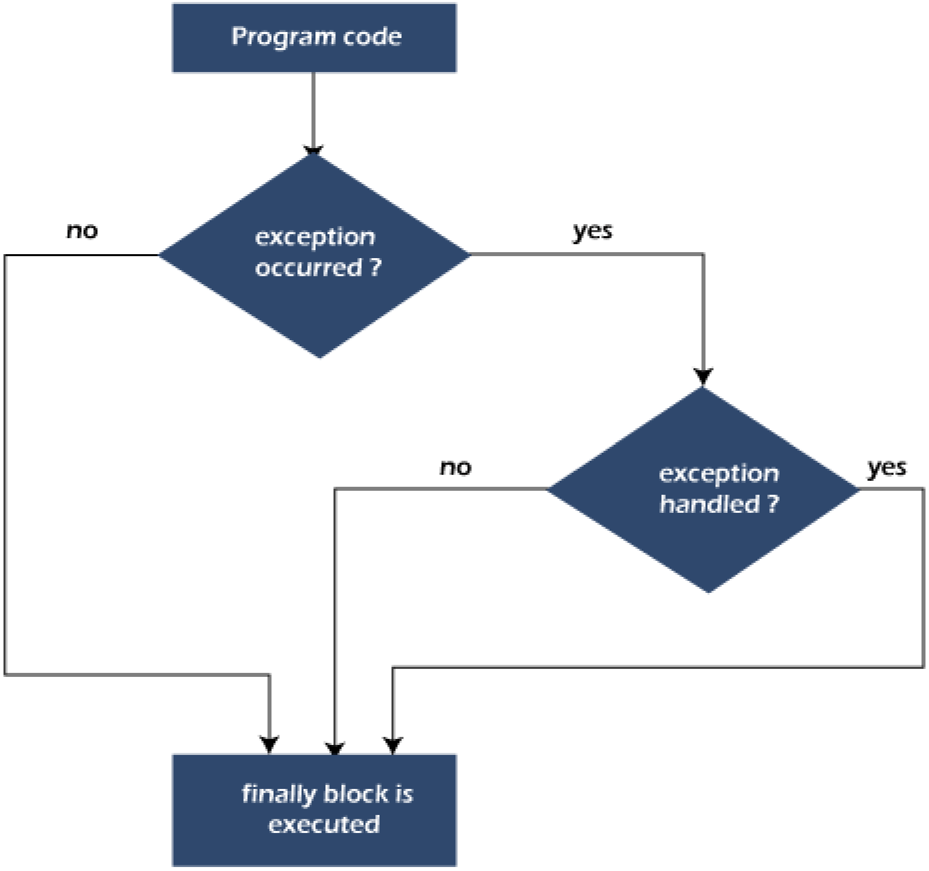
Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



> *Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).*

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

Let's see the different cases where Java finally block can be used.

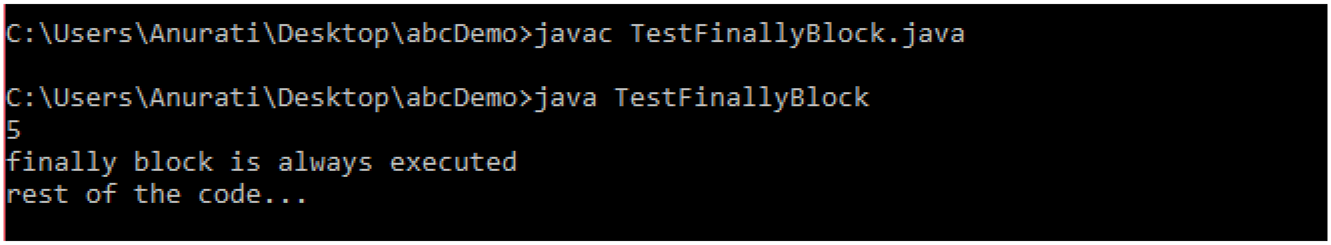
Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

TestFinallyBlock.java

```java
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

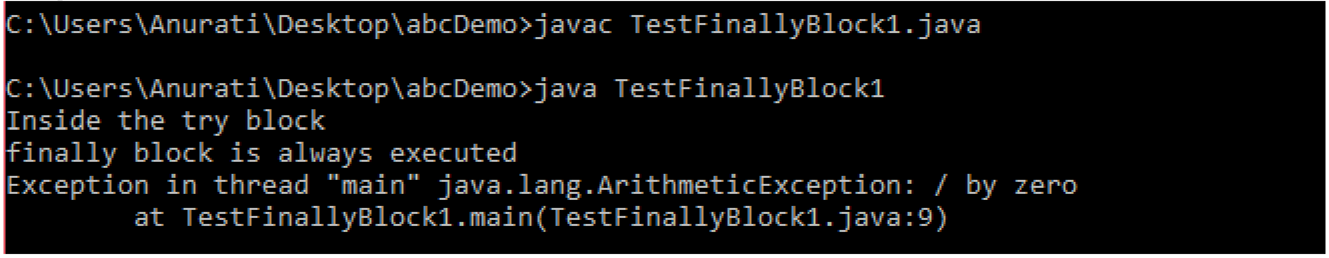
Case 2: When an exception occurs but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```
public class TestFinallyBlock1{
      public static void main(String args[]){

      try {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }
      //cannot handle Arithmetic type exception
      //can only accept Null Pointer type exception
      catch(NullPointerException e){
        System.out.println(e);
      }

      //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed");
      }

      System.out.println("rest of the code...");
      }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Case 3: When an exception occurs and is handled by the catch block

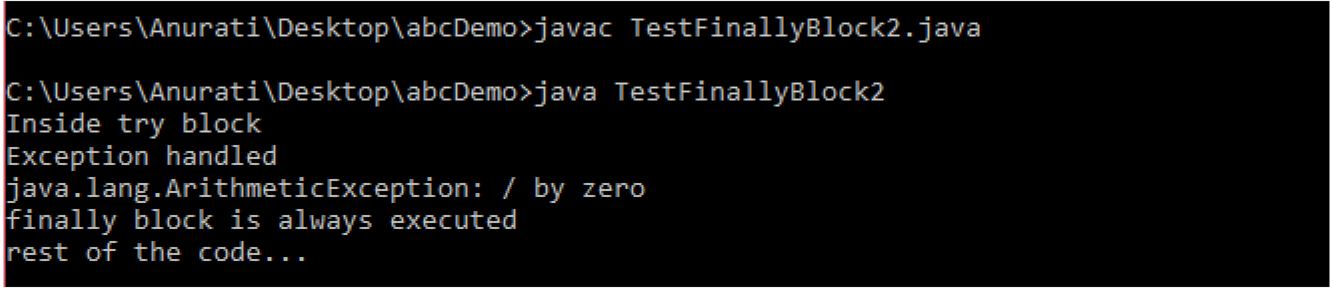
Example:

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

TestFinallyBlock2.java

```
public class TestFinallyBlock2{
    public static void main(String args[]){

    try {

        System.out.println("Inside try block");

        //below code throws divide by zero exception
      int data=25/0;
      System.out.println(data);
    }

    //handles the Arithmetic Exception / Divide by zero exception
    catch(ArithmeticException e){
        System.out.println("Exception handled");
        System.out.println(e);
    }

    //executes regardless of exception occurred or not
    finally {
      System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");
    }
  }
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. **For example**, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

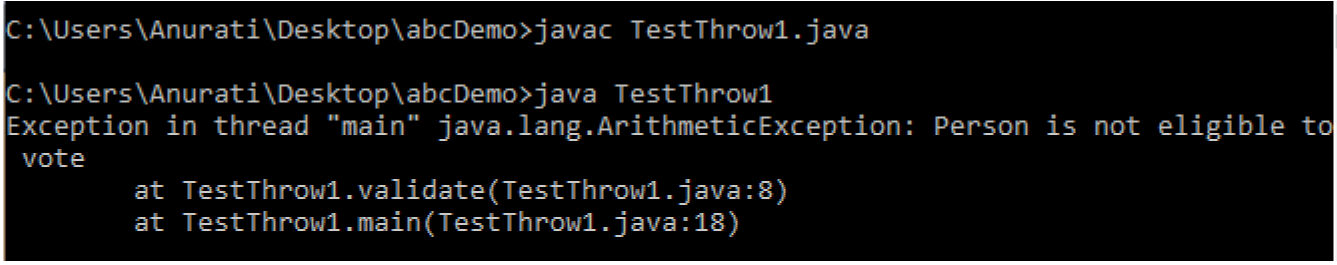
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
    //main method
    public static void main(String args[]){
        //calling the function
        validate(13);
        System.out.println("rest of the code...");
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

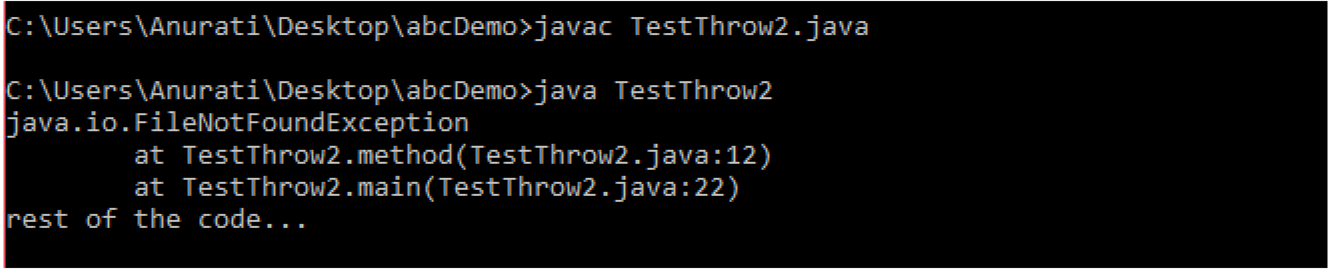
Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
import java.io.*;

public class TestThrow2 {

    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {

        FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);


        throw new FileNotFoundException();

    }
    //main method
    public static void main(String args[]){
        try
        {
            method();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

Example 3: Throwing User-defined Exception

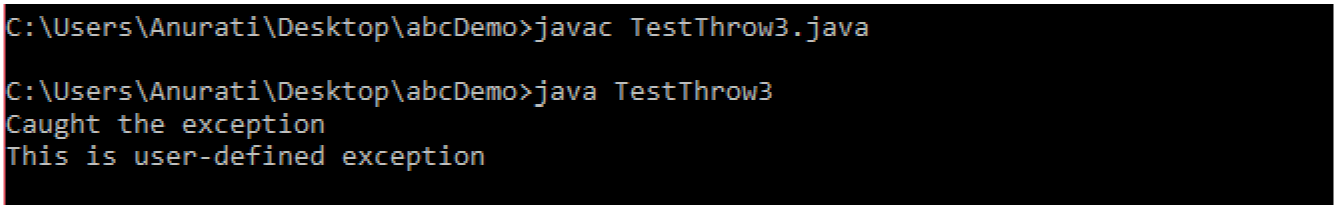
exception is everything else under the Throwable class.

TestThrow3.java

```
// class represents user-defined exception
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    }
}
// Class that uses above MyException
```

```
public class TestThrow3
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        }
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow3
Caught the exception
This is user-defined exception
```

Java Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Exception Propagation Example

TestExceptionPropagation1.java

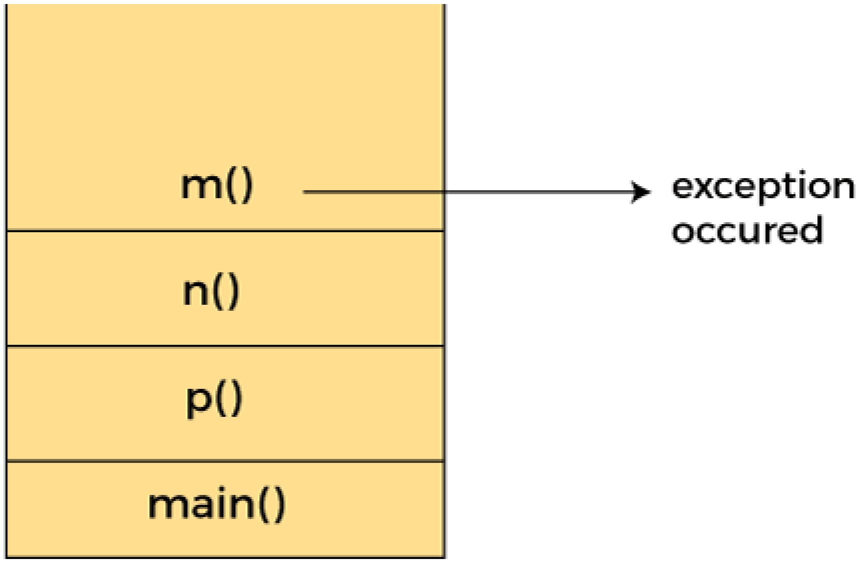
```
class TestExceptionPropagation1{
  void m(){
   int data=50/0;
  }
  void n(){
   m();
  }
  void p(){
   try{
   n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
       normal flow...
```

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



Call Stack

Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Exception Propagation Example

TestExceptionPropagation1.java

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handeled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation2 obj=new TestExceptionPropagation2();
   obj.p();
   System.out.println("normal flow");
  }
}
```

Output:

```
Compile Time Error
```

Java throws keyword

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 the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws Example

Let's see the example of Java throws clause which describes that **checked exceptions can be propagated by throws keyword.**

Testthrows1.java

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow...
```

> *Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.*

There are two cases:

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

Testthrows2.java

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
```

```
public class Testthrows2{
  public static void main(String args[]){
   try{
    M m=new M();
    m.method();
   }catch(Exception e){System.out.println("exception handled");}

   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
       normal flow...
```

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

Let's see examples for both the scenario.

A) If exception does not occur

Testthrows3.java

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
    M m=new M();
    m.method();

   System.out.println("normal flow...");
  }
}
```

Output:

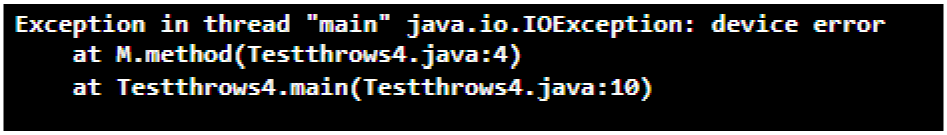
```
device operation performed
       normal flow...
```

B) If exception occurs

Testthrows4.java

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:



```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

Sr. no.	Basis of Differences	throw	throws

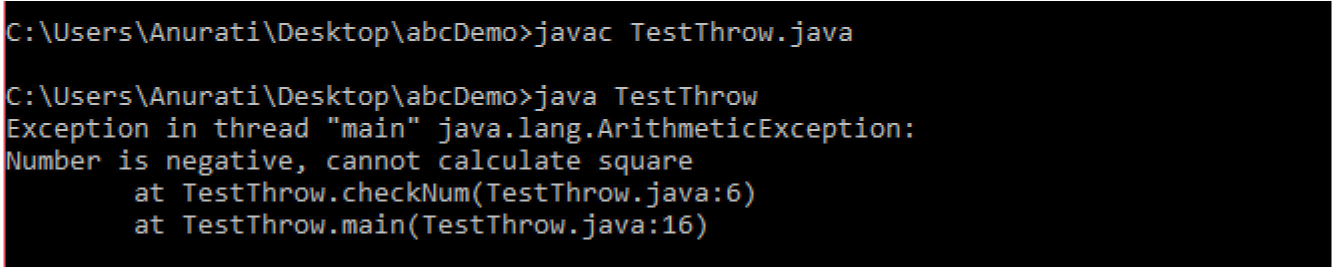
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java throw Example

TestThrow.java

```
public class TestThrow {
    //defining a method
    public static void checkNum(int num) {
        if (num < 1) {
            throw new ArithmeticException("\nNumber is negative, cannot calculate square");
        }
        else {
            System.out.println("Square of " + num + " is " + (num*num));
        }
    }
    //main method
    public static void main(String[] args) {
            TestThrow obj = new TestThrow();
            obj.checkNum(-3);
            System.out.println("Rest of the code..");
    }
}
```

Output:



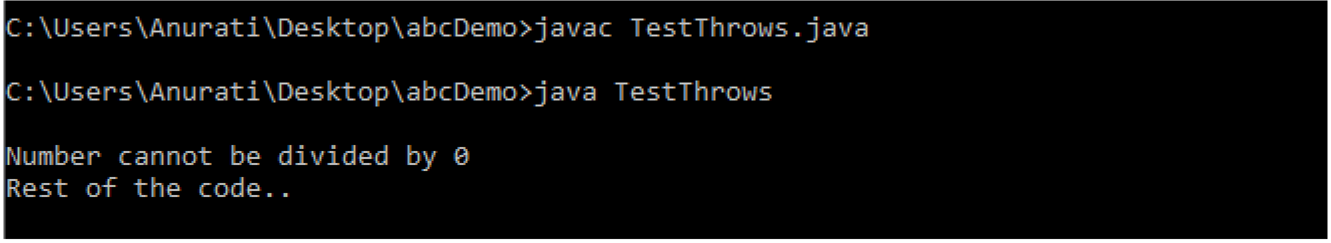
```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws Example

TestThrows.java

```
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmeticException {
        int div = m / n;
        return div;
    }
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
            System.out.println(obj.divideNum(45, 0));
        }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }

        System.out.println("Rest of the code..");
    }
}
```

Output:



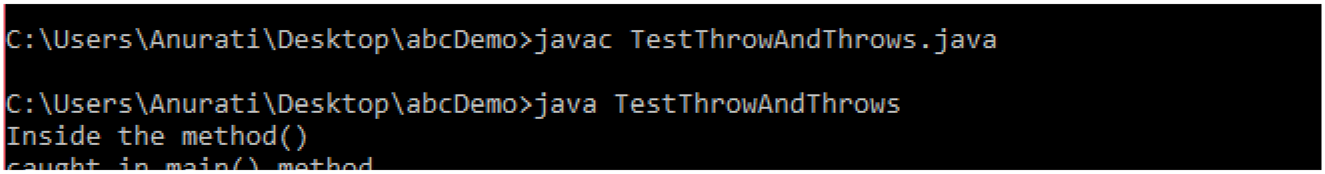
```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

TestThrowAndThrows.java

```
public class TestThrowAndThrows
{
    // defining a user-defined method
    // which throws ArithmeticException
    static void method() throws ArithmeticException
    {
        System.out.println("Inside the method()");
        throw new ArithmeticException("throwing ArithmeticException");
    }
    //main method
    public static void main(String args[])
    {
        try
        {
            method();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught in main() method");
        }
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.

4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

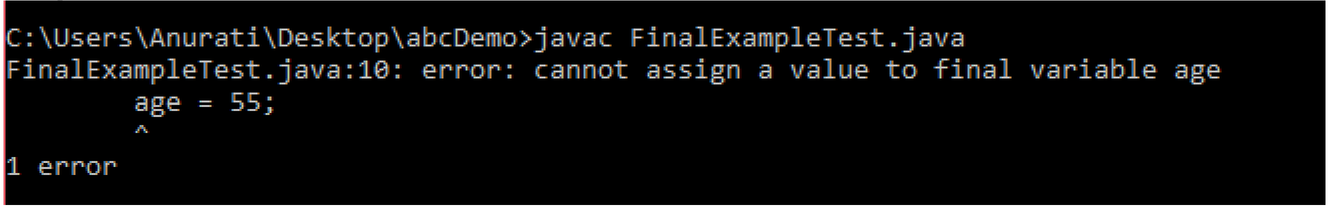
Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

FinalExampleTest.java

```java
public class FinalExampleTest {
    //declaring final variable
    final int age = 18;
    void display() {

    // reassigning value to age variable
    // gives compile time error
    age = 55;
    }

    public static void main(String[] args) {

    FinalExampleTest obj = new FinalExampleTest();
    // gives compile time error
    obj.display();
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
        age = 55;
        ^
1 error
```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

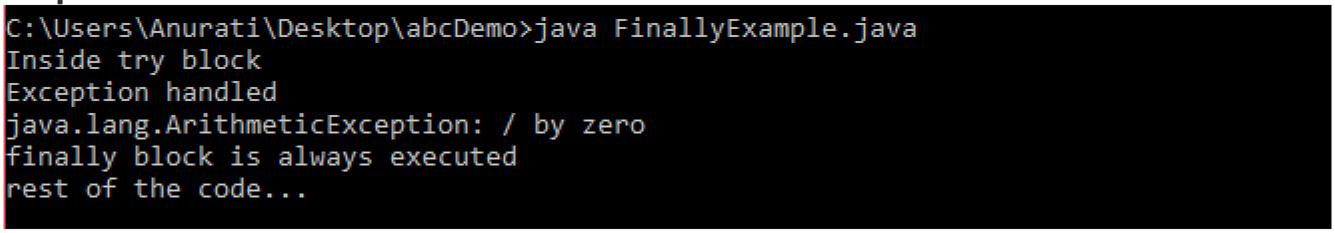
Java finally Example

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

FinallyExample.java

```
public class FinallyExample {
    public static void main(String args[]){
    try {
      System.out.println("Inside try block");
    // below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    // handles the Arithmetic Exception / Divide by zero exception
    catch (ArithmeticException e){
      System.out.println("Exception handled");
      System.out.println(e);
    }
    // executes regardless of exception occurred or not
    finally {
      System.out.println("finally block is always executed");
    }
    System.out.println("rest of the code...");
    }
  }
```

Output:



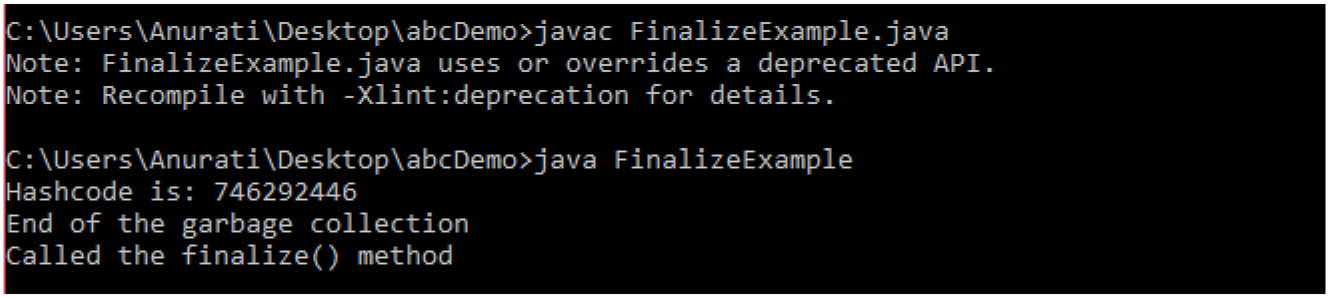
```
C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Java finalize Example

FinalizeExample.java

```
public class FinalizeExample {
     public static void main(String[] args)
    {
        FinalizeExample obj = new FinalizeExample();
        // printing the hashcode
        System.out.println("Hashcode is: " + obj.hashCode());
        obj = null;
        // calling the garbage collector using gc()
        System.gc();
        System.out.println("End of the garbage collection");
    }
   // defining the finalize method
    protected void finalize()
    {
        System.out.println("Called the finalize() method");
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
Hashcode is: 746292446
End of the garbage collection
Called the finalize() method
```

Exception Handling with Method Overriding in Java

There are many rules if we talk about method overriding with exception handling.

Some of the rules are listed below:

- **If the superclass method does not declare an exception**

    - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
    - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

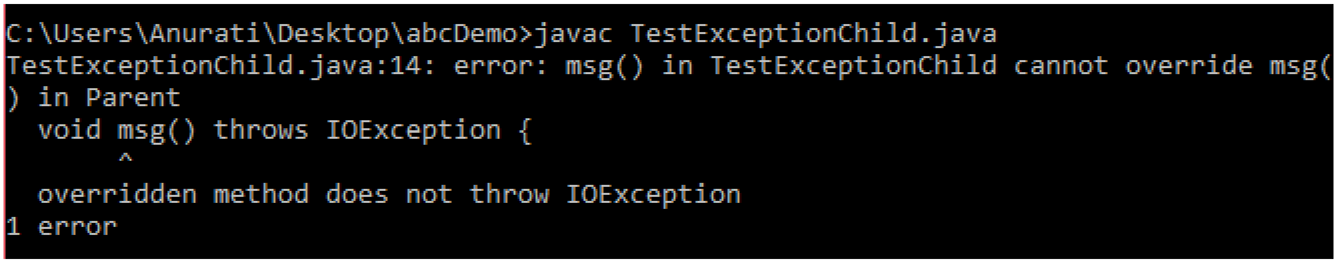
> *Rule 1: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.*

Let's consider following example based on the above rule.

TestExceptionChild.java

```
import java.io.*;
class Parent{

  // defining the method
  void msg() {
    System.out.println("parent method");
    }
}

public class TestExceptionChild extends Parent{

  // overriding the method in child class
  // gives compile time error
  void msg() throws IOException {
    System.out.println("TestExceptionChild");
  }

  public static void main(String args[]) {
   Parent p = new TestExceptionChild();
   p.msg();
  }
}
```

Output:



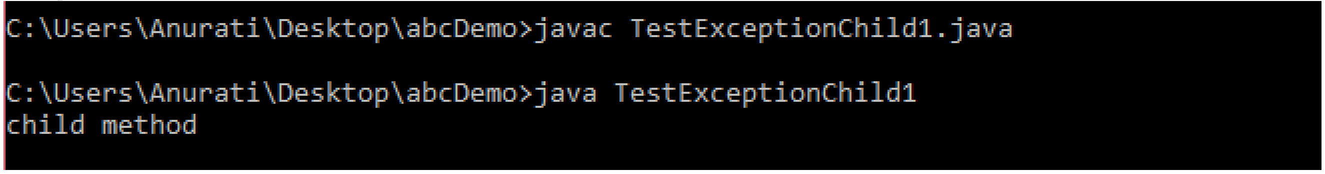
```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild.java
TestExceptionChild.java:14: error: msg() in TestExceptionChild cannot override msg(
) in Parent
  void msg() throws IOException {
       ^
  overridden method does not throw IOException
1 error
```

Rule 2: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

TestExceptionChild1.java

```
import java.io.*;
class Parent{
  void msg() {
    System.out.println("parent method");
  }
}

class TestExceptionChild1 extends Parent{
  void msg()throws ArithmeticException {
    System.out.println("child method");
  }

  public static void main(String args[]) {
   Parent p = new TestExceptionChild1();
   p.msg();
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild1.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild1
child method
```

If the superclass method declares an exception

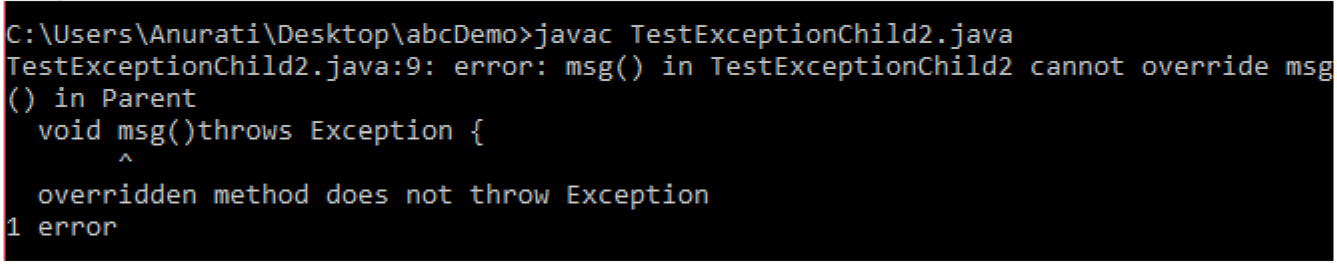
Rule 1: If the superclass method declares an exception, subclass overridden method can declare the same subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

TestExceptionChild2.java

```
import java.io.*;
class Parent{
  void msg()throws ArithmeticException {
    System.out.println("parent method");
  }
}

public class TestExceptionChild2 extends Parent{
  void msg()throws Exception {
    System.out.println("child method");
  }

  public static void main(String args[]) {
   Parent p = new TestExceptionChild2();

   try {
   p.msg();
   }
   catch (Exception e){}

  }
}
```

Output:



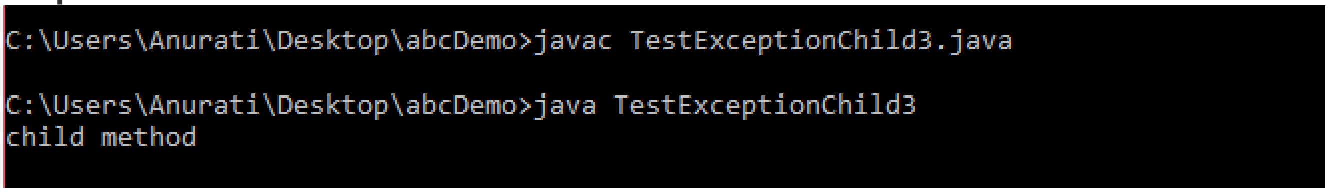
```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild2.java
TestExceptionChild2.java:9: error: msg() in TestExceptionChild2 cannot override msg
() in Parent
  void msg()throws Exception {
       ^
  overridden method does not throw Exception
1 error
```

Example in case subclass overridden method declares same exception

TestExceptionChild3.java

```java
import java.io.*;
class Parent{
  void msg() throws Exception {
    System.out.println("parent method");
  }
}

public class TestExceptionChild3 extends Parent {
  void msg()throws Exception {
    System.out.println("child method");
  }

  public static void main(String args[]){
   Parent p = new TestExceptionChild3();

   try {
   p.msg();
   }
   catch(Exception e) {}
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild3.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild3
child method
```

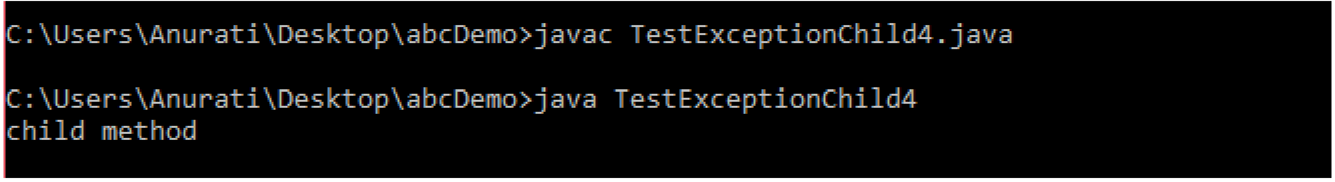
Example in case subclass overridden method declares subclass exception

TestExceptionChild4.java

```java
import java.io.*;
class Parent{
  void msg()throws Exception {
    System.out.println("parent method");
  }
}

```

```
class TestExceptionChild4 extends Parent{
  void msg()throws ArithmeticException {
    System.out.println("child method");
  }

  public static void main(String args[]){
   Parent p = new TestExceptionChild4();

   try {
   p.msg();
   }
   catch(Exception e) {}
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild4.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild4
child method
```

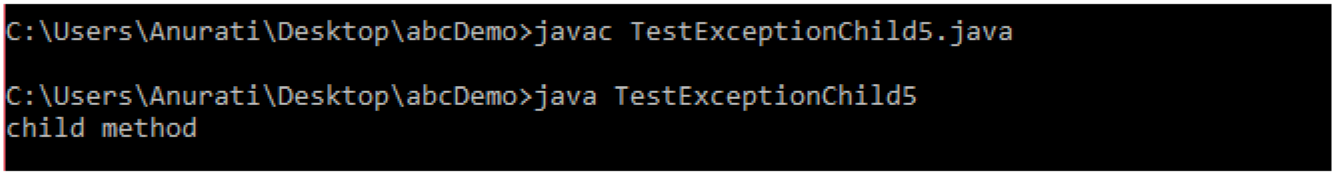
Example in case subclass overridden method declares no exception

TestExceptionChild5.java

```
import java.io.*;
class Parent {
  void msg()throws Exception{
    System.out.println("parent method");
  }
}

class TestExceptionChild5 extends Parent{
  void msg() {
    System.out.println("child method");
  }

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

```
Parent p = new TestExceptionChild5();

try {
p.msg();
}
catch(Exception e) {}

 }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild5.java

C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild5
child method
```

Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.

- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

```
public class WrongFileNameException extends Exception {
   public WrongFileNameException(String errorMessage) {
   super(errorMessage);
   }
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

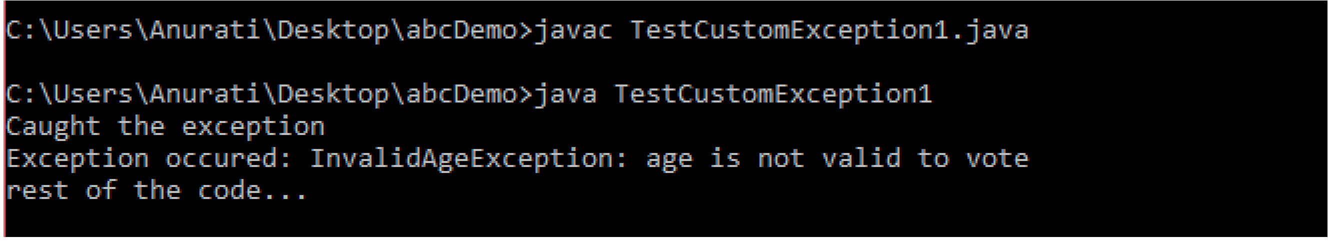
Example 1:

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

TestCustomException1.java

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

    // method to check the age
    static void validate (int age) throws InvalidAgeException{
      if(age < 18){

        // throw an object of user defined exception
        throw new InvalidAgeException("age is not valid to vote");
    }
      else {
        System.out.println("welcome to vote");
        }
     }

    // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }

        System.out.println("rest of the code...");
    }
}
```

Output:



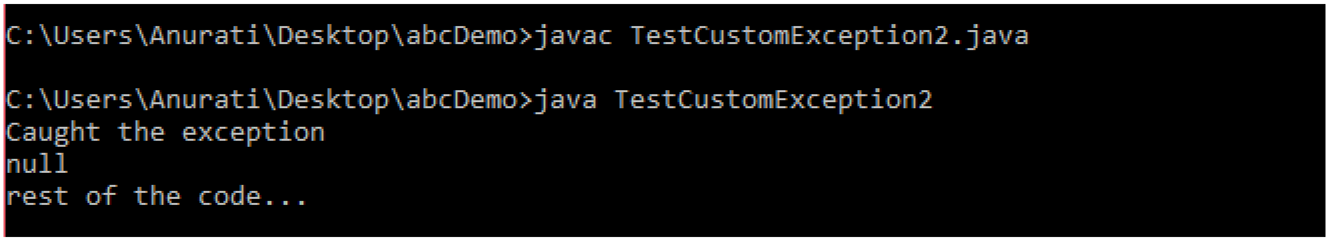
```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Example 2:

TestCustomException2.java

```
// class representing custom exception
class MyCustomException extends Exception
{

}

// class that uses custom exception MyCustomException
public class TestCustomException2
{
    // main method
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new MyCustomException();
        }
        catch (MyCustomException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }

        System.out.println("rest of the code...");
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
```

Multithreading

3. Explain with example, suspending and resuming threads in Java

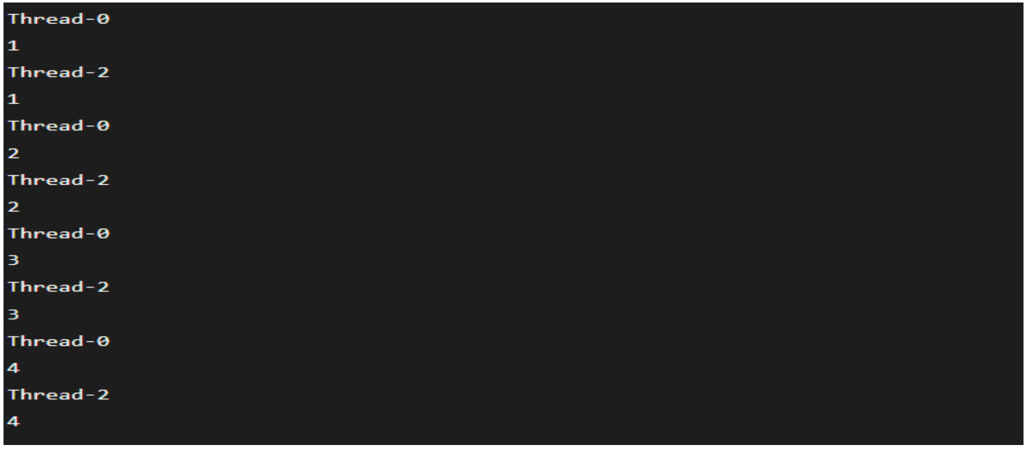
The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Program:

```java
public class JavaSuspendExp extends Thread
{
    public void run()
    {
        for(int i=1; i<5; i++)
        {
            try
            {
                // thread to sleep for 500 milliseconds
                 sleep(500);
                 System.out.println(Thread.currentThread().getName());
            }catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        // creating three threads
        JavaSuspendExp t1=new JavaSuspendExp ();
        JavaSuspendExp t2=new JavaSuspendExp ();
        JavaSuspendExp t3=new JavaSuspendExp ();
        // call run() method
        t1.start();
        t2.start();
        // suspend t2 thread
        t2.suspend();
```

```
            // call run() method
            t3.start();
        }
    }
```

Output:



```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

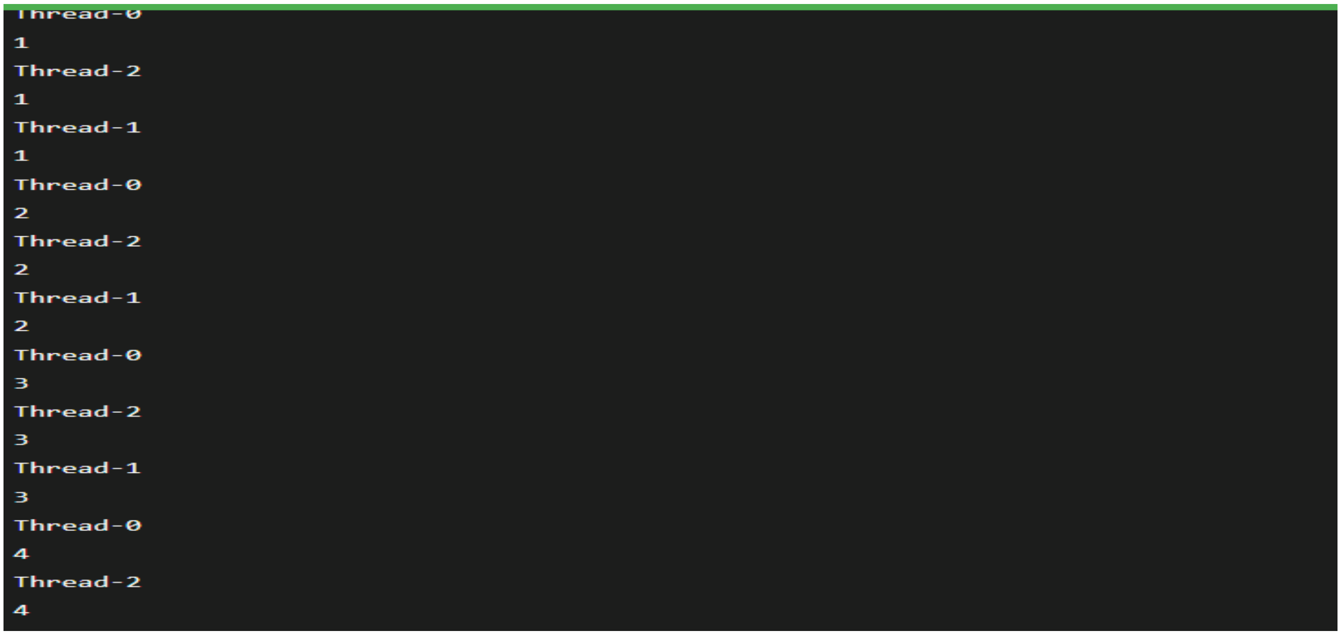
The resume() method of thread class is only used with suspend() method. This method is used to resume a thread which was suspended using suspend() method. This method allows the suspended thread to start again.

program:

```
public class JavaResumeExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
         try
         {
            // thread to sleep for 500 milliseconds
             sleep(500);
             System.out.println(Thread.currentThread().getName());
         }catch(InterruptedException e){System.out.println(e);}
         System.out.println(i);
      }
```

```
    }
    public static void main(String args[])
    {
        // creating three threads
        JavaResumeExp t1=new JavaResumeExp ();
        JavaResumeExp t2=new JavaResumeExp ();
        JavaResumeExp t3=new JavaResumeExp ();
        // call run() method
        t1.start();
        t2.start();
        t2.suspend(); // suspend t2 thread
        // call run() method
        t3.start();
        t2.resume(); // resume t2 thread
    }
}
```

output:



```
Thread-0
1
Thread-2
1
Thread-1
1
Thread-0
2
Thread-2
2
Thread-1
2
Thread-0
3
Thread-2
3
Thread-1
3
Thread-0
4
Thread-2
4
```

4.What is Daemon Thread

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

5.How Inter-thread Communication is done in java?
Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class:

wait()
notify()
notifyAll()

1) wait() method
The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

- It waits until object is notified.

2) notify() method
The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

3) notifyAll() method
Wakes up all threads that are waiting on this object's monitor.

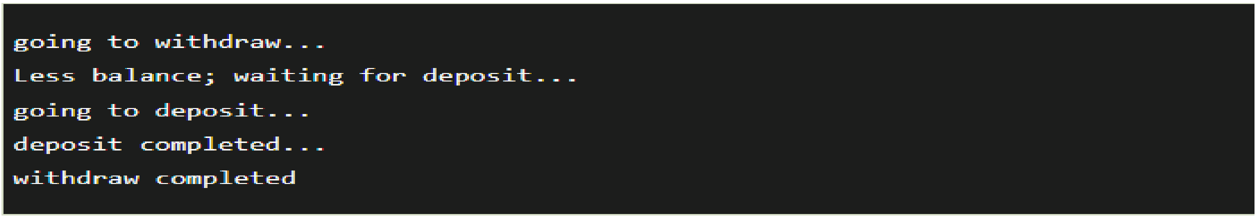
Syntax:

```
public final void notifyAll()
```

program:

```
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-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
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:



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

6.Write a java program to create threads with different priorities.

program:

```
import java.lang.*;
class ThreadDemo extends Thread {
        public void run()
        {t
                System.out.println("Inside run method");
        }
        public static void main(String[] args)
        {
                ThreadDemo t1 = new ThreadDemo();
                ThreadDemo t2 = new ThreadDemo();
                ThreadDemo t3 = new ThreadDemo();


                System.out.println("t1 thread priority : "
                                                + t1.getPriority());
                System.out.println("t2 thread priority : "
                                                + t2.getPriority());

                System.out.println("t3 thread priority : "
                                                + t3.getPriority());
                t1.setPriority(2);
                t2.setPriority(5);
                t3.setPriority(8);
                System.out.println("t1 thread priority : "
                                                + t1.getPriority());

                System.out.println("t2 thread priority : "
                                                + t2.getPriority());

                System.out.println("t3 thread priority : "
                                                + t3.getPriority());


                System.out.println(
                    "Currently Executing Thread : "
                    + Thread.currentThread().getName());

                System.out.println(
                    "Main thread priority : "
                    + Thread.currentThread().getPriority());
                Thread.currentThread().setPriority(10);

                System.out.println(
                    "Main thread priority : "
                    + Thread.currentThread().getPriority());
        }
}
```

output:

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

7.Why thread is called a light weight process? What are the different things shared by different threads of a single process? What are the benefits of this sharing

Threads are sometimes called lightweight processes because they have their own stack but can access shared data. Because threads share the same address space as the process and other threads within the process, the operational cost of communication between the threads is low, which is an advantage.
When a process starts, it is assigned memory and resources. Each thread in the process shares that memory and resources. In single-threaded processes, the process contains one thread. The process and the thread are one and the same, and there is only one thing happening.

9.Discuss about join keyword.
join() method
The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution. Suppose th be the object the class Thread whose thread is doing its execution currently, then the th.join(); statement ensures that th is finished before the program does the execution of the next statement. When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period. However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

Description of The Overloaded join() Method
join(): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

Syntax:

public final void join() throws InterruptedException
join(long mls): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

public final synchronized void join(long mls) throws InterruptedException, where mls is in milliseconds
join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

public final synchronized void join(long mls, int nanos) throws InterruptedException, where mls is in milliseconds.

10.Explain the synchronization of multiple threads in Java with an example.

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.

Example:

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

output:

```
         5
        10
        15
        20
        25
        100
```

```
200
300
400
500
```

11.Explain thread states with block diagram. And, Associate methods to each state

Thread States in Java
A thread is a program in execution created to perform a specific task. Life cycle of a Java thread starts with its birth and ends on its death.

The start() method of the Thread class is used to initiate the execution of a thread and it goes into runnable state and the sleep() and wait() methods of the Thread class sends the thread into non runnable state.

After non runnable state, thread again comes into runnable state and starts its execution. The run() method of thread is very much important. After executing the run() method, the lifecycle of thread is completed.

All these phases of threads are the states of thread in Java
To work with threads in a program, it is important to identify thread state. The following figure shows thread states in Java thread life cycle.

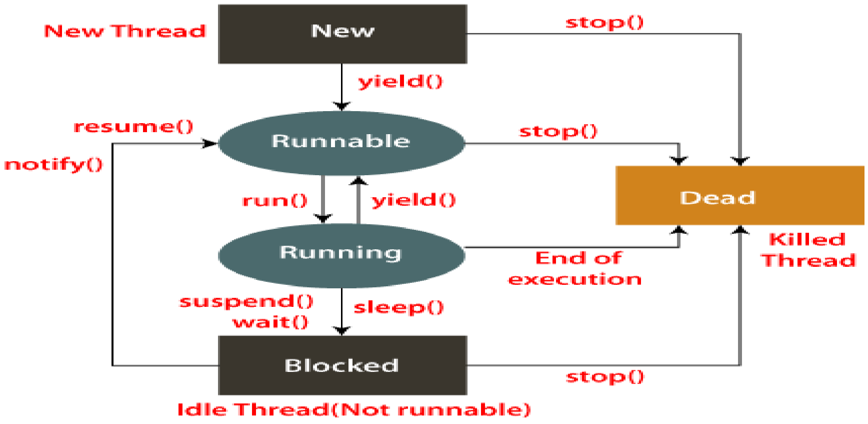


Fig: State Transition Diagram of a Thread

Thread States in Java

A thread is a path of execution in a program that goes through the following states of a thread. The five states are as follows:

New
Runnable
Running
Blocked (Non-runnable state)
Dead

New (Newborn State)

When an instance of the Thread class is created a new thread is born and is known to be in New-born state. That is, when a thread is born, it enters into new state but its execution phase has not been started yet on the instance.

In simpler terms, Thread object is created but it cannot execute any program statement because it is not in an execution state of the thread. Only start() method can be called on a new thread; otherwise, an IllegalThreadStateException will be thrown.

Runnable State

The second phase of a new-born thread is the execution phase. When the start() method is called on a the new instance of a thread, it enters into a runnable state.

In the runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). There are many threads that are ready for execution, they all are waiting in a queue (line).

If all threads have equal priority, a time slot is assigned for each thread execution on the basis of first-come, first-serve manner by CPU. The process of allocating time to threads is known as time slicing. A thread can come into runnable state from running, waiting, or new states.

Running State

Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.

In running state, processor gives its time to the thread for execution and executes its run method. It is the state where thread performs its actual functions. A thread can come into running state only from runnable state.

A running thread may give up its control in any one of the following situations and can enter into the blocked state.

When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.

Blocked State

A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.

Dead State

A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.

12.What is race condition?
A condition in which the critical section (a part of the program where shared memory is accessed) is concurrently executed by two or more threads. It leads to incorrect behavior of a program.

In layman terms, a race condition can be defined as, a condition in which two or more threads compete together to get certain shared resources.

For example, if thread A is reading data from the linked list and another thread B is trying to delete the same data. This process leads to a race condition that may result in run time error.

I/O Operations

1.What is a stream? Explain its types.

Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, i/o devise, other program etc.

In general, a Stream will be an input stream or, an output stream.

InputStream − This is used to read data from a source.
OutputStream − This is used to write data to a destination.
Based on the data they handle there are two types of streams −

Byte Streams − These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
Character Streams − These handle data in 16 bit Unicode. Using these you can read and write text data only.
Standard Streams
In addition to above mentioned classes Java provides 3 standard streams representing the input and, output devices.

Standard Input − This is used to read data from user through input devices. keyboard is used as standard input stream and represented as System.in.
Standard Output − This is used to project data (results) to the user through output devices. A computer screen is used for standard output stream and represented as System.out.

Standard Error – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

2.List classes that are used by Byte streams for input and output operation

ByteStream Classes

ByteStream classes are used to read bytes from the input stream and write bytes to the output stream. In other words, we can say that ByteStream classes read/write the data of 8-bits. We can store video, audio, characters, etc., by using ByteStream classes. These classes are part of the java.io package.

The ByteStream classes are divided into two types of classes, i.e., InputStream and OutputStream. These classes are abstract and the super classes of all the Input/Output stream classes.

InputStream Class

The InputStream class provides methods to read bytes from a file, console or memory. It is an abstract class and can't be instantiated; however, various classes inherit the InputStream class and override its methods. The subclasses of InputStream class are given in the following table.

SN	Class	Description
1	BufferedInputStream	This class provides methods to read bytes from the buffer.
2	ByteArrayInputStream	This class provides methods to read bytes from the byte array.
3	DataInputStream	This class provides methods to read Java primitive data types.
4	FileInputStream	This class provides methods to read bytes from a file.
5	FilterInputStream	This class contains methods to read bytes from the other input streams, which are used as the primary source of data.
6	ObjectInputStream	This class provides methods to read objects.
7	PipedInputStream	This class provides methods to read from a piped output stream to which the piped input stream must be connected.
8	SequenceInputStream	This class provides methods to connect multiple Input

OutputStream Class

The OutputStream is an abstract class that is used to write 8-bit bytes to the stream. It is the superclass of all the output stream classes. This class can't be instantiated; however, it is inherited by various subclasses that are given in the following table.

SN	Class	Description
1	BufferedOutputStream	This class provides methods to write the bytes to the buffer.
2	ByteArrayOutputStream	This class provides methods to write bytes to the byte array.
3	DataOutputStream	This class provides methods to write the java primitive data types.
4	FileOutputStream	This class provides methods to write bytes to a file.
5	FilterOutputStream	This class provides methods to write to other output streams.

6	ObjectOutputStream	This class provides methods to write objects.
7	PipedOutputStream	It provides methods to write bytes to a piped output stream.
8	PrintStream	It provides methods to print Java primitive data types.

3.List classes that are used by character streams for input and List output operation

CharacterStream Classes

The java.io package provides CharacterStream classes to overcome the limitations of ByteStream classes, which can only handle the 8-bit bytes and is not compatible to work directly with the Unicode characters. CharacterStream classes are used to work with 16-bit Unicode characters. They can perform operations on characters, char arrays and Strings.

However, the CharacterStream classes are mainly used to read characters from the source and write them to the destination. For this purpose, the CharacterStream classes are divided into two types of classes, I.e., Reader class and Writer class.

Reader Class

Reader class
is used to read the 16-bit characters from the input stream. However, it is an abstract class and can't be instantiated, but there are various subclasses that inherit the Reader class and override the methods of the Reader class. All methods of the Reader class throw an IOException. The subclasses of the Reader class are given in the following table.

SN	Class	Description
1.	BufferedReader	This class provides methods to read characters from the buffer.
2.	CharArrayReader	This class provides methods to read characters from the char array.
3.	FileReader	This class provides methods to read characters from the file.
4.	FilterReader	This class provides methods to read characters from the underlying character input stream.
5	InputStreamReader	This class provides methods to convert bytes to characters.
6	PipedReader	This class provides methods to read characters from the connected piped output stream.
7	StringReader	This class provides methods to read characters from a string.

Writer Class

Writer class is used to write 16-bit Unicode characters to the output stream. The methods of the Writer class generate IOException. Like Reader class, Writer class is also an abstract class that cannot be instantiated; therefore, the subclasses of the Writer class are used to write the characters onto the output stream. The subclasses of the Writer class are given in the below table.

SN	Class	Description
1	BufferedWriter	This class provides methods to write characters to the buffer.
2	FileWriter	This class provides methods to write characters to the file.

3	CharArrayWriter	This class provides methods to write the characters to the character array.
4	OutpuStreamWriter	This class provides methods to convert from bytes to characters.
5	PipedWriter	This class provides methods to write the characters to the piped output stream.
6	StringWriter	This class provides methods to write the characters to the string.

4.Illustrate PrintWriter Class to handle console output.

Java PrintWriter class
Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.
Program:

```
import java.io.PrintWriter;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

5.Write a program to copy one file to another using try with resources block

```
import java.io.FileOutputStream;
public class TryWithResources {
public static void main(String args[]){
        // Using try-with-resources
try(FileOutputStream fileOutputStream =newFileOutputStream("/java7-new-
features/src/abc.txt")){
String msg = "Welcome to javaTpoint!";
byte byteArray[] = msg.getBytes(); //converting string into byte array
fileOutputStream.write(byteArray);
System.out.println("Message written to file successfuly!");
}catch(Exception exception){
      System.out.println(exception);
}
}
}
```

6.Write a program to create and write a string in a file

Program:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

// Main class
public class GFG {

        // Main driver method
        public static void main(String[] args)
                throws IOException
        {
                // Assigning the content of the file
                String text
                        = "Welcome to geekforgeeks\nHappy Learning!";

                // Defining the file name of the file
                Path fileName = Path.of(
                        "/Users/mayanksolanki/Desktop/demo.docx");

                Files.writeString(fileName, text);
                String file_content = Files.readString(fileName);

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

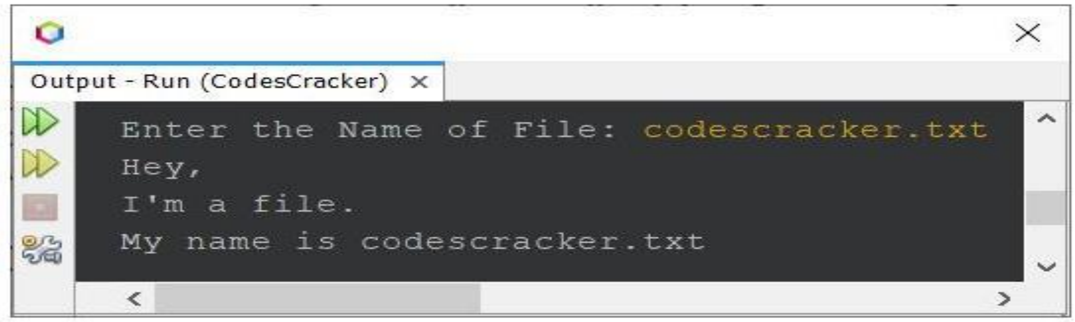
Output:

```
Welcome to geekforgeeks
Happy Learning!
```

7.WAP to display the contents of the file

```
import java.util.Scanner;
import java.io.*;

public class CodesCracker
{
   public static void main(String[] args)
   {
      String fname;
      Scanner scan = new Scanner(System.in);

      // enter filename along with its extension
      System.out.print("Enter the Name of File: ");
      fname = scan.nextLine();

      String line = null;
      try
      {
         FileReader fileReader = new FileReader(fname);

         // always wrap the FileReader in BufferedReader
         BufferedReader bufferedReader = new BufferedReader(fileReader);

         while((line = bufferedReader.readLine()) != null)
         {
            System.out.println(line);
            }

            // always close the file after its use
            bufferedReader.close();
        }
        catch(IOException ex)
        {
            System.out.println("\nError occurred");
            System.out.println("Exception Name: " +ex);
        }
    }
}
```

Output:



```
Enter the Name of File: codescracker.txt
Hey,
I'm a file.
My name is codescracker.txt
```

Generics

1.What are Generics?

Generics in Java

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

OR

Generics means parameterized types. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

2.Write a program to Illustrate the working of Generic class in Java

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

output:

```
2
```

3.Write a program to Illustrate the working of generic method in Java

Program:

```
public class TestGenerics4{

   public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
         }
         System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Output

```
Printing Integer Array
10
20
30
40
50
Printing Character Array
J
A
V
A
T
P
O
I
N
T
```

4.Implement bounded type parameters using extends Comparable object.

Program:

```
public class MaximumTest {
   // determines the largest of three Comparable objects

   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x;   // assume x is initially the largest

      if(y.compareTo(max) > 0) {
         max = y;   // y is the largest so far
      }

      if(z.compareTo(max) > 0) {
         max = z;   // z is the largest now
      }
      return max;   // returns the largest object
   }

   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

Output

```
Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear
```

5.Write a program to illustrate the working of generic constructors

Program:

```
class Test {
        //Generics constructor
        public <T> Test(T item){
                System.out.println("Value of the item: " + item);
                System.out.println("Type of the item: "
                                + item.getClass().getName());
        }
}

public class GenericsTest {
        public static void main(String args[]){
                //String type test
                Test test1 = new Test("Test String.");
                Test test2 = new Test(100);
        }
}
```

Output:

```
Value of the item: Test String.
Type of the item: java.lang.String
Value of the item: 100
Type of the item: java.lang.Integer
```

6.What are the restrictions to be remembered while using generics

Restrictions on Generics

- To use Java generics effectively, you must consider the following restrictions:
- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type.

Collection Framework

1.What is collection Framework

Collections in Java

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a g1roup of objects. It has:

1.Interfaces and its implementations, i.e., classes

2.Algorithm

2.What are legacy classes

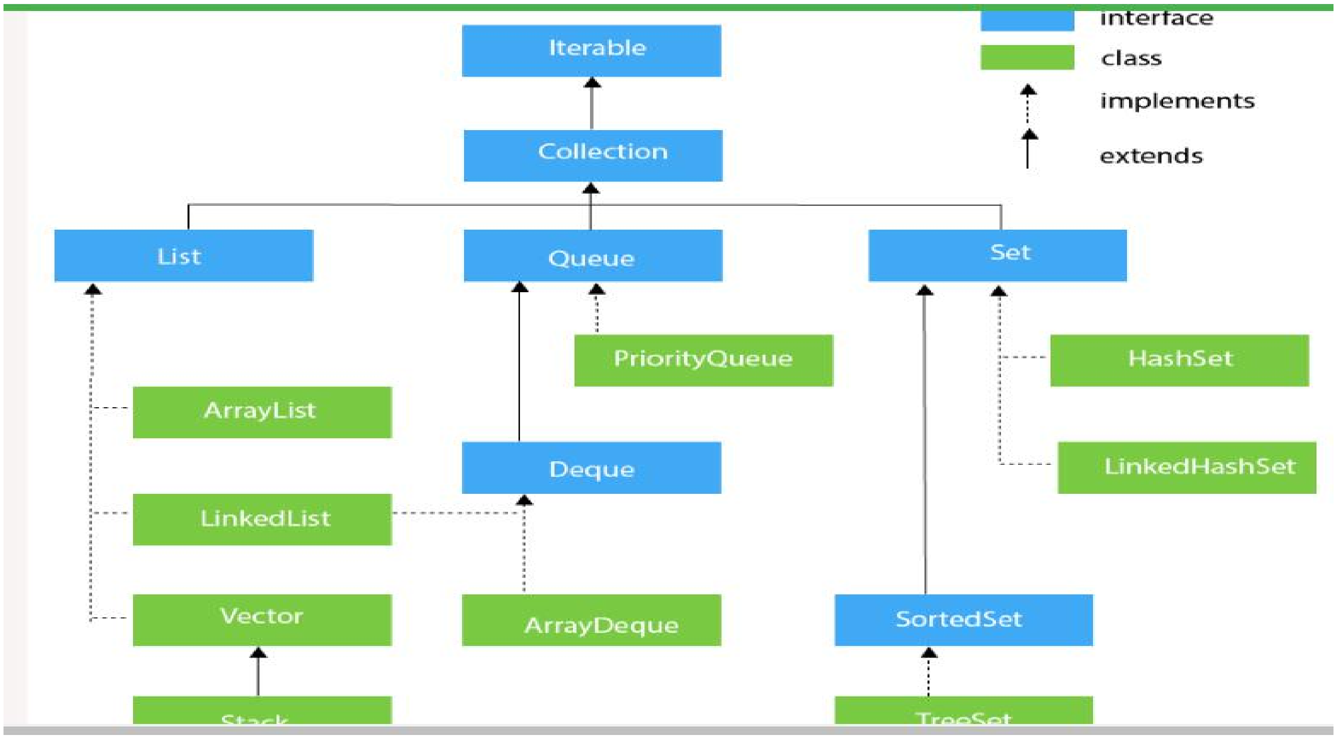
Legacy Class in Java

In the past decade, the Collection framework didn't include in Java. In the early version of Java, we have several classes and interfaces which allow us to store objects. After adding the Collection framework in JSE 1.2, for supporting the collections framework, these classes were re-engineered. So, classes and interfaces that formed the collections framework in the older version of Java are known as Legacy classes. For supporting generic in JDK5, these classes were re-engineered.

All the legacy classes are synchronized. The java.util package defines the following legacy classes:

1.HashTable
2.Stack
3.Dictionary
4.Properties
5.Vector

3.Draw the hierarchy of collection framework.
Hierarchy of Collection Framework
Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces
for the Collection framework.



4.WAP to traverse ArrayList elements using the Iterator interface

```
import java.util.ArrayList;
import java.util.Iterator;


class Main {
  public static void main(String[] args){
    ArrayList<String> languages = new ArrayList<>();


    // Add elements in the array list
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Swift");

    // Create a variable of Iterator
    // store the iterator returned by iterator()
    Iterator<String> iterate = languages.iterator();
    System.out.print("ArrayList: ");

    // loop through ArrayList till it has all elements
    // Use methods of Iterator to access elements
    while(iterate.hasNext()){
      System.out.print(iterate.next());
      System.out.print(", ");
    }
  }
}
```

Output

```
ArrayList: Java, Python, JavaScript, Swift
```

5.WAP to add and traverse elements in a Linked List.

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){

  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
     Ravi
     Vijay
     Ravi
     Ajay
```

6.WAP to insert, remove and display elements in a stack

```
import java.util.Stack;
public class Demo {
   public static void main (String args[]) {
      Stack stack = new Stack();
      stack.push("Apple");
      stack.push("Mango");
      stack.push("Guava");
      stack.push("Pear");
      stack.push("Orange");
      System.out.println("The stack elements are: " + stack);
   }
}
```

Output

```
The stack elements are: [Apple, Mango, Guava, Pear, Orange]
```

7.WAP to insert, remove and display elements in a queue

```
import java.util.LinkedList;
import java.util.Queue;
public class Example {
   public static void main(String[] args) {
      Queue<Integer> q = new LinkedList<>();
      q.add(6);
      q.add(1);
      q.add(8);
      q.add(4);
      q.add(7);
      System.out.println("The queue is: " + q);
      int num1 = q.remove();
System.out.println("The element deleted from the head is: " + num1);
System.out.println("The queue after deletion is: " + q);
int head = q.peek();
System.out.println("The head of the queue is: " + head);
int size = q.size();
System.out.println("The size of the queue is: " + size);
}
}
```

Output

```
The queue is: [6, 1, 8, 4, 7]
The element deleted from the head is: 6
The queue after deletion is: [1, 8, 4, 7]
The head of the queue is: 1
The size of the queue is: 4
```

8.Demonstrate the methods in StringTokenizer class with an example

Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.
int countTokens()	It returns the total number of tokens.

EXAMPLES:

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

Output:

```
my
name
is
```

khan

Applets

1.What is an applet?

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.

OR

An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to make the website more dynamic and entertaining.

2.Write a simple applet that displays “hello world” and explain its working.

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorld extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("Hello World", 20, 20);
        }
}
```

3.Explain the concept of applet to applet communication with suitable program

Applet Communication

java.applet.AppletContext class provides the facility of communication between applets. We provide the name of applet through the HTML file. It provides getApplet() method that returns the object of Applet.

Syntax:public Applet getApplet(String name){}

Example of Applet Communication

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ContextApplet extends Applet implements ActionListener{
Button b;
public void init(){
b=new Button("Click");
b.setBounds(50,50,60,50);

add(b);
b.addActionListener(this);
}

public void actionPerformed(ActionEvent e){

AppletContext ctx=getAppletContext();
Applet a=ctx.getApplet("app2");
a.setBackground(Color.yellow);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="ContextApplet.class" width="150" height="150" name="app1">
</applet>

<applet code="First.class" width="150" height="150" name="app2">
</applet>
</body>
</html>
```

4.**What for repaint( ) method is used?**

The repaint method is an asynchronous method of applet class. When call to repaint method is made, it performs a request to erase and perform redraw of the component after a small delay in time.

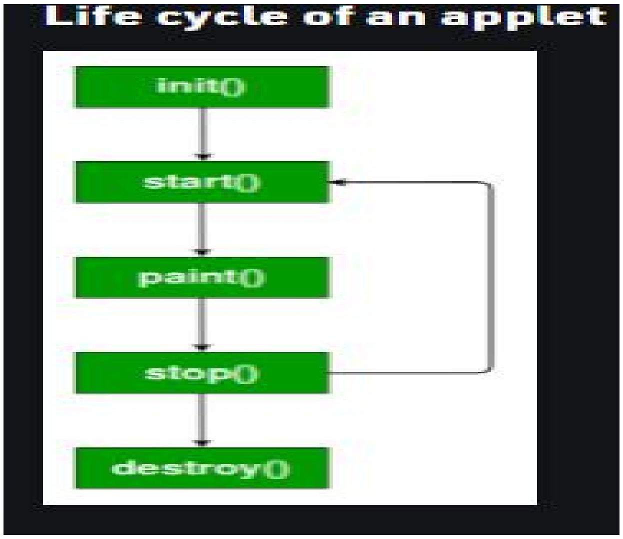
5.Differentiate between paint() and repaint() functions.

Paint() and Repaint()

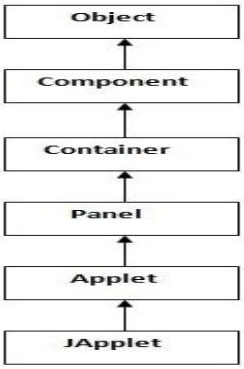
paint(): This method holds instructions to paint this component. In Java Swing, we can change the paintComponent() method instead of paint() method as paint calls paintBorder(), paintComponent() and paintChildren() methods. We cannot call this method directly instead we can call repaint().

repaint(): This method cannot be overridden. It controls the update() -> paint() cycle. We can call this method to get a component to repaint itself. If we have done anything to change the look of the component but not the size then we can call this method.

6.Present the structure of a Java Applet



OR



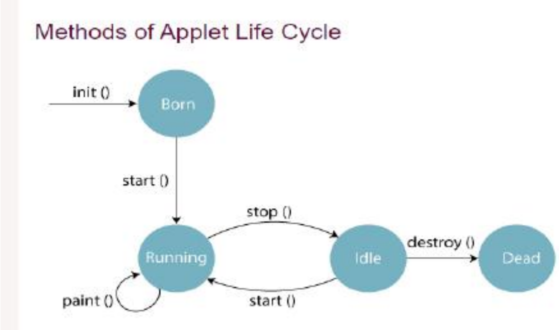
7.Discuss in detail about Applet life cycle

Applet Life Cycle in Java

In Java, an applet
is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

Along with the browser, the applet also works on the client side, thus having less processing time.



init(): The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.
start(): The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.
stop(): The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
destroy(): The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
paint(): The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

8.How to pass the parameters to an Applet? Explain with example

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax:

```
public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```
import java.applet.Applet;
import java.awt.Graphics;

public class UseParam extends Applet{

public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}
```

Swings:

1.Explain the delegation event model.

Delegation Event Model

The Delegation Event model is defined to handle events in GUI programming languages. The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system. The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

In this section, we will discuss event processing and how to implement the delegation event model in Java. We will also discuss the different components of an Event Model.

The Delegation Event Model has the following key participants namely:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

Listener - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

2.Explain procedure to handle mouse events in Java with an example program

Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

```
public abstract void mouseClicked(MouseEvent e);
public abstract void mouseEntered(MouseEvent e);
public abstract void mouseExited(MouseEvent e);
public abstract void mousePressed(MouseEvent e);
public abstract void mouseReleased(MouseEvent e);
```

program:

```
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);
        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();
}
}
```

4,Discuss about list box and choice boxes

Java 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.

Java AWT Choice

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 Component class.

5.Explain Jtree Swing components with suitable example.

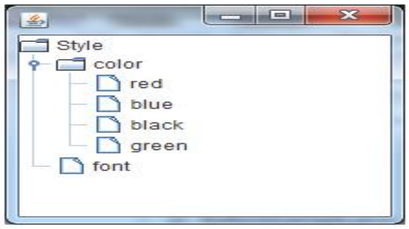
Java JTree

The JTree class is used to display the tree structured data or hierarchical data. JTree is a complex component. It has a 'root node' at the top most which is a parent for all nodes in the tree. It inherits JComponent class.

Program:

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample {
JFrame f;
TreeExample(){
    f=new JFrame();
    DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
    DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
    DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");
    style.add(color);
    style.add(font);
    DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
    DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
    DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
    DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
    color.add(red); color.add(blue); color.add(black); color.add(green);
    JTree jt=new JTree(style);
    f.add(jt);
    f.setSize(200,200);
    f.setVisible(true);
}
public static void main(String[] args) {
    new TreeExample();
}}
```

Output:



6.What is MVC architecture in Java?

MVC Architecture in Java

The Model-View-Controller (MVC) is a well-known design pattern in the web development field. It is way to organize our code. It specifies that a program or application shall consist of data model, presentation information and control information.

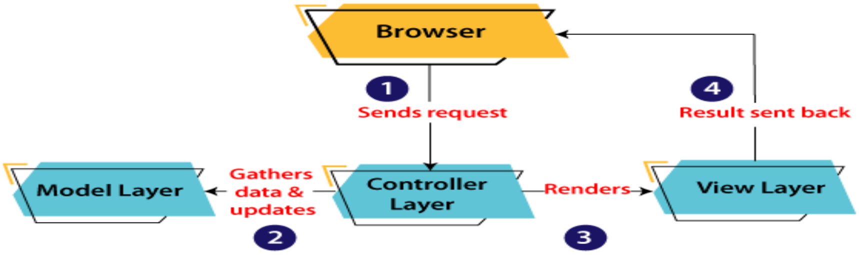
The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.

The MVC pattern architecture consists of three layers:

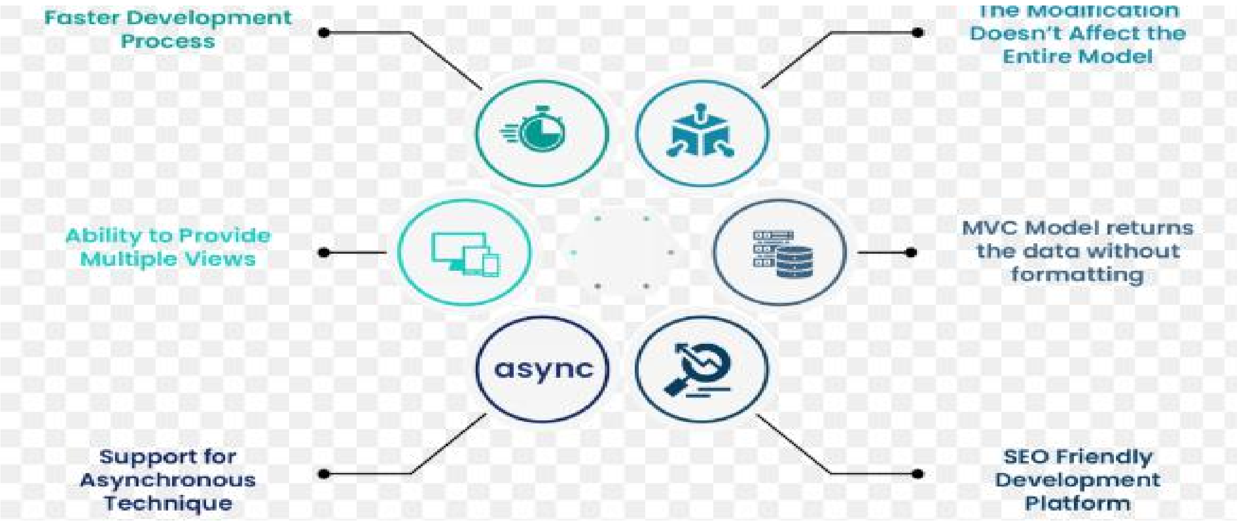
Model: It represents the business layer of application. It is an object to carry the data that can also contain the logic to update controller if data is changed.

View: It represents the presentation layer of application. It is used to visualize the data that the model contains.

Controller: It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.



7.What are the advantages of MVC architecture?


Faster Development Process
The Modification Doesn't Affect the Entire Model
Ability to Provide Multiple Views
MVC Model returns the data without formatting
async
Support for Asynchronous Technique
SEO Friendly Development Platform

ANNAMACHARYA INSTITUTE OF TECHNOLOGY & SCIENCES :: TIRUPATHI
AUTONOMOUS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name : Object Oriented Programming Through JAVA**	**AY:** 2021-2022
Unit Wise Bit Bank			

QNO	Questions
	UNIT-1 (OOPS Basics, Class, Object, String Handling)
1	**Bytecode is executed by JVM** leads to the portability and security of Java?
2	**Use of pointers** is not a Java features?

3	What should be the execution order, if a class has a method, static block, instance block, and constructor, as shown below?
	public class First_C {
	public void myMethod()
	{
	System.out.println("Method");
	}
	{
	System.out.println(" Instance Block");
	}
	public void First_C()
	{
	System.out.println("Constructor ");
	}
	static {
	System.out.println("static block");
	}
	public static void main(String[] args) {
	First_C c = new First_C();
	c.First_C();
	c.myMethod();
	}
	}
	Ans: Static block, instance block, constructor, and method

4	Evaluate the following Java expression, if x=3, y=5, and z=10: ++z + y - y + z + x++ **Ans: 24**
5	Which of the following for loop declaration is not valid? a. for (int i = 99; i >= 0; i / 9) b. for (int i = 7; i <= 77; i += 7) c. for (int i = 20; i >= 2; - -i) d. for (int i = 2; i <= 20; i = 2* i) **Answer: (a)**
6	a. An interface with no fields or methods is known as a **Marker Interface**
7	Which option is false about the *final* keyword? a. A *final* method cannot be overridden in its subclasses.

	b. A *final* class cannot be extended. c. A *final* class cannot extend other classes. d. A *final* method can be inherited. **Answer: (c)**
8	**String memory** memory a String is stored, when we create a string using new operator?
9	**Import** keyword is used for accessing the features of a package?
10	In java, jar stands for **Java ARchive**
11	How many threads can be executed at a time? **Multiple threads**
12	Modulus operator, %, can be applied to **Both Integers and floating – point numbers**
13	**Box obj = new Box();** is a valid declaration of an object of class Box?
14	**super** can be used in a subclass to call the constructor of superclass?
15	**Polymorphism** technique is used to method overriding in java
16	A class member declared protected becomes a member of subclass of which type? **private member**
17	multiple inheritance in Java can be implemented by using **interfaces**
18	All classes in Java are inherited from which class? **java.lang.Object**
19	If super class and subclass have same variable name, which keyword should be used to use super class? **super**
20	In order to restrict a variable of a class from inheriting to subclass, how variable should be declared? **private**
	UNIT-2 (Inheritance, Packages, Interfaces)
1	The **import** statement is used to include another Java package in a Java source file.
2	A subclass can call a constructor method defined by its super class by use of the **super** keyword.
3	URL stands for **Uniform Resource Locator**
4	**super** keyword can be used in a subclass to call the constructor of superclass?
5	**polymorphism** is supported by method overriding in Java?
6	**extends** keyword must be used to inherit a class?
7	A class member declared protected becomes a member of subclass of **private member**
8	multiple inheritance in Java can be implemented by using **interfaces**
9	All classes in Java are inherited from which class? **java.lang.Object**
10	If super class and subclass have same variable name, which keyword should be used to use super class? **super.variablename**
11	In order to restrict a variable of a class from inheriting to subclass, how variable should be declared? **private**
12	**Java.lang** is superclass of String and StringBuffer class?
13	**+** operator can be used to concatenate two or more String objects?
14	**length()** method of class String is used to obtain a length of String object?
15	**charAt()** method of class String is used to extract a single character from a String object?

16	**java.lang** package contain all the Java's built in exceptions?
17	Can a class implement more than one interfaces? **Yes, a class can implement more than one interfaces.**
18	What methods would a class that implements the java.lang.CharSequence interface have to implement? **charAt, length, subSequence, and toString.**

19	Is the following interface valid?
	public interface Marker {
	}
	Ans: Yes. Methods are not required.

20	For every interface written in a java file, .class file will be generated after compilation? True or False? **True. For every interface written in a java file, .class file will be generated after compilation.**
	UNIT-3 (Exception Handling, I/O Operations, Generic Programming)
1	**NullPointerException** exception will be thrown if we use null reference for an arithmetic operation?
2	**Exception** class is used to create user defined exception?
3	What is the use of try & catch? It allows us to manually handle the exception, It allows to fix errors, It prevents automatic terminating of the program in cases when an exception occurs
4	**throw** Keywords are used for generating an exception manually?
5	**package** keywords is used to define packages in Java?
6	**Packages** is a mechanism for naming and visibility control of a class and its content?
7	**public** access specifiers can be used for a class so that its members can be accessed by a different class in the different package?
8	**java** package stores all the standard java classes?
9	**interfaces** can be used to fully abstract a class from its implementation?
10	**implements** is a keyword used by a class to use an interface defined previously?
11	Which of these is an incorrect array declaration? a) int arr[] = new int[5] b) int [] arr = new int[5] c) int arr[] = new int[5] **d) int arr[] = int [5] new**

12	What will be the output of the following Java code?
	int arr[] = new int [5];
	System.out.print(arr);
	Ans: Class name@ hashcode in hexadecimal form

13	When does method overloading is determined? **At compile time**
14	**java.lang** class is superclass of String and StringBuffer class?
15	**super** keyword can be used in a subclass to call the constructor of superclass?

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name : Object Oriented Programming Through JAVA**	**AY:** 2021-2022
Unit Wise Bit Bank			

16	**final** keywords can be used to prevent Method overriding?

17	At line number 2 in the following code, choose 3 valid data-type attributes/qualifiers among “final, static, native, public, private, abstract, protected”
	
	public interface Status
	{
	/* insert qualifier here */ int MY_VALUE = 10;
	}
	a) final, native, private
	b) final, static, protected
	c) final, private, abstract
	d) final, static, public

18	**abstract** keywords are used to define an abstract class?
19	**java.lang** packages contains abstract keyword?

20	Which of these is correct way of inheriting class A by class B?
	a) class B + class A {}
	b) class B inherits class A {}
	c) class B extends A {}
	d) class B extends class A {}

1. Which of these is used to perform all input & output operations in Java?

A. streams **B.** Variables **C.** classes **D.** Methods

Answer: A

2. What does AWT stands for?

A. All Window Tools B. All Writing Tools C. Abstract Window Toolkit D. Abstract Writing Toolkit

Answer: Option C

3. In java, how many streams are created for us automatically?

A. 2 B. 3 C. 4 D. 5

Answer :B
4._Which method is used to write a byte to the current output stream?
A. public void write(int)throws IOException
B. public void write(byte[])throws IOException
C. public void flush()throws IOException
D. public void close()throws IOException

Answer :A
5.Which method is used to write an array of byte to the current output stream?
A. public void write(int)throws IOException
B. public void write(byte[])throws IOException
C. public void flush()throws IOException
D. public void close()throws IOException

Answer :B
6._ ____________ returns true if called on a file and returns false when called on a directory.

A. IsFile()
B. Isfile()
C. isFile()
D. isfile()

Answer:C

7.What will be output for the following code?

```
import java.io.*;

class files

{

    public static void main(String args[])

    {

        File obj = new File(""/java/system"");

        System.out.print(obj.getName());
        }


}
```

8. Which is Commonly used Methods of ByteArrayOutputStream class?

A) ByteArrayOutputStream()

B) ByteArrayOutputStream(int size)

C) public synchronized void writeTo(OutputStream out) throws IOException

D) Both A & B

Answer:C

9. Which class can be used to read data line by line by readLine() method?

A) BufferedReader

B) InputStreamReader

C) DataInputStream

D) None of the above

Answer:A

10. Which package includes StringTokenizer which tokenizes a string into independent words?

A) java.awt
B) java.applet
C) java.util
D) java.lang

Answer:C

11. Which of these is a process of writing the state of an object to a byte stream?
a) Serialization
b) Externalization
c) File Filtering
d) All of the mentioned

Answer:A

12. Which of these is a method of ObjectOutput interface used to finalize the output state so that any buffers are cleared?
a) clear()
b) flush()
c) fflush()
d) close()

Answer:B

13. Consider the following two statements

int x = 25;

Integer y = new Integer(33);

What is the difference between these two statements?

A. Primitive data types

B. primitive data type and an object of a wrapper class

C. Wrapper class

D. None of the above

14. What will be the output of the following Java program?

```
class Output
{
    public static void main(String args[])
    {
        Integer i = new Integer(257);
        byte x = i.byteValue();
        System.out.print(x);
    }
}
```

15. What will be the output of the following Java program?

```
class Output
{
    public static void main(String args[])
    {
        Integer i = new Integer(257);
        float x = i.floatValue();
        System.out.print(x);
    }
}
```

16. What will be the output of the following Java program?

```
class Output
{
    public static void main(String args[])
    {
        Long i = new Long(256);
        System.out.print(i.hashCode());
    }
}
```

17. Which of these type parameters is used for a generic methods to return and accept any type of object?
a) K
b) N
c) T
d) V

Answer:C

18. Which of these type parameters is used for a generic methods to return and accept a number?
a) K
b) N
c) T
d) V

Answer:B

19. What will be the output of the following Java code?

```
  class multithreaded_programing

  {

      public static void main(String args[])

      {

          Thread t = Thread.currentThread();

          System.out.println(t);

      }
```

20. Which method is called internally by Thread start() method?

A. execute()
B. run()
C. launch()
D. main()

Answer:B

21. Number of threads in below java program is

```
public class ThreadExtended extends Thread {

  public void run() {
              System.out.println("\nThread is running now\n");
  }

  public static void main(String[] args) {

              ThreadExtended threadE = new ThreadExtended();

              threadE.start();
  }
}
```

22. Which method is used to make main thread to wait for all child threads

A. Join ()
B. Sleep ()
C. Wait ()
D. Stop ()

Answer:A

23. Which of these are types of multitasking?
A. Process basedB. Thread basedC. Process and Thread basedD. None of the mentioned
Answer:C

24. The . . . . method of the thread is called before the . . . . method and carries out any initialisation.
A) suspend, resume
B) start, run
C) start, stop
D) resume, suspend
Answer:B

25. What will be the output of the program?

```
class Test extends Thread {
public
    void run()
    {
        System.out.println("Run");
    }
} class Myclass {
public
    static void main(String[] args)
    {
        Test t = new Test();
        t.start();
    }
}
```

26. Which of these packages contain all the collection classes?
a) java.lang
b) java.util
c) java.net
d) java.awt
Answer:B

27. What is Collection in Java?
a) A group of objects
b) A group of classes
c) A group of interfaces
d) None of the mentioned
Answer:A

28. What will be the output of the following Java program?

```
    import java.util.*;
    class Array
    {
        public static void main(String args[])
        {
            int array[] = new int [5];
            for (int i = 5; i > 0; i--)
                array[5-i] = i;
            Arrays.fill(array, 1, 4, 8);
            for (int i = 0; i < 5 ; i++)
```

```
                System.out.print(array[i]);
        }
    }
```

29. What will be the output of the following Java program?

```
    import java.util.*;
    class Bitset
    {
        public static void main(String args[])
        {
            BitSet obj = new BitSet(5);
            for (int i = 0; i < 5; ++i)
                obj.set(i);
            obj.clear(2);
            System.out.print(obj);
        }
    }
```

30. Which of this interface must contain a unique element?
A. Set B. List   C. Array d. Collection
Answer:A

31. What is the output of this program?

```
    import java.util.*;
    class Array
    {
        public static void main(String args[])
        {
            int array[] = new int [5];
            for (int i = 5; i > 0; i--)
                array[5 - i] = i;
            Arrays.sort(array);
            for (int i = 0; i < 5; ++i)
             System.out.print(array[i]);;
        }
    }
```

32. What is the output of this program?

```
    import java.util.*;
    class Collection_Algos
    {
        public static void main(String args[])
        {
            LinkedList list = new LinkedList();
            list.add(new Integer(2));
            list.add(new Integer(8));
            list.add(new Integer(5));
            list.add(new Integer(1));
            Iterator i = list.iterator();
            Collections.reverse(list);
     Collections.sort(list);
            while(i.hasNext())
          System.out.print(i.next() + " ");
        }
    }
```

33. Which class is used to generate random number?

a. java.lang.Object b. java.util.randomNumber
c.java.util.Random d. java.util.Object
Answer:C

34. The interface Comparable contains the method ___________
A toCompare
B compare
C compareTo
D compareWith
Answer:C

35. Iterator and ListIterator can iterate through __________
A List
B Set
C Map
D All the answers are true
Answer:A

36. The collection is a ____________

A.framework and interface

B.framework and class

C.only interface

D.only class
Answer:A

37._WT stands for ?

A. All Window Toolkit
B. Abstract Work Toolkit
C. Abstract Window Toolkit
D. Abstract Window Text
Answer:C

38._What is the super class of all components of Java?

A. java.all.Component
B. all.awt.Component
C. java.awt.Component
D. awt.Component

Answer:C
39._How many layout managers defined in java.awt package?

A. 2
B. 3
C. 4
D. 5

Answer:D
40._ A ___________ dictates the style of arranging the components in a container.

A. border layout
B. grid layout
C. panel
D. layout manager

Answer:D

41._Which method used to place some text in the text field?

A. getText(String str)
B. setText(String str)
C. putText(String str)
D. None of the above

Answer:B

42.Which method used to change the foreground (text) color of components like text field?

A. setBackground(Color clr)
B. setForeground(Color clr)
C. setColor(Color clr)
D. setEditable(boolean state)

Answer:B

43. Which of these functions is called to display the output of an applet?

a) display()
b) paint()
c) displayApplet()
d) PrintApplet()

Answer:B

44. What is the Message is displayed in the applet made by the following Java program?

```
import java.awt.*;

import java.applet.*;

public class myapplet extends Applet

{

    public void paint(Graphics g)

    {

        g.drawString("A Simple Applet", 20, 20);

    }
}
```

45. What is the length of the application box made by the following Java program?

```
import java.awt.*;

import java.applet.*;

public class myapplet extends Applet
```

```
{

    public void paint(Graphics g)

    {

        g.drawString("A Simple Applet", 20, 20);

    }
}
```

46._Applets are designed to be embedded within an __________.

A. Javascript
B. Css
C. HTML
D. SQL
Answer:C

47._Which of the following is required to view an applet?

A. JCM
B. JDM
C. JVM
D. Java class
Answer:C

48.___________ method is defined in Graphics class, it is used to output a string in an applet.

A. display()
B. Print()
C. drawString()
D. transient()

Answer:C

49. when an applet is terminated the following sequence of methods calls takes place?

A) stop(),paint(),destroy()

B) destroy(),stop(),paint()

C) destroy(),stop()
D) stop(),destroy()
Answer:D

50. Package of drawstring() method is
A. java.applet B. java.io C. javax.swing D. java.awt

Answer:D

51. Which of these methods can be used to know which key is pressed?
A. getActionEvent()B. getActionKey()C. getModifier()D. getKey()

Answer:C

52. MouseEvent is subclass of which of these classes?

a) ComponentEvent

b) ContainerEvent

c) ItemEvent
d) InputEvent
Answer:D

53. Which of the following contains both date and time?

a) java.io.date

b) java.sql.date

c) java.util.date

d) java.util.dateTime
Answer:D

54. Which of the following is advantage of using PreparedStatement in Java?

a) Slow performance

b) Encourages SQL injection

c) Prevents SQL injection
d) More memory usage
Answer:C

55. Which of the following is used to call stored procedure?

a) Statement

b) PreparedStatement

c) CallableStatment
d) CalledStatement
Answer:C

56. he doGet() method in the example extracts values of the parameter’s type and number by using __________

a) request.getParameter()

b) request.setParameter()

c) responce.getParameter()

d) responce.getAttribute()
Answer:A

57. How many JDBC driver types does Sun define?
a) One
b) Two
c) Three
d) Four
Answer:D

58. Which JDBC driver Type(s) is(are) the JDBC-ODBC bridge?
a) Type 1
b) Type 2
c) Type 3
d) Type 4
Answer:A

59. DriverManager.getConnection(_______ , ______ , ______)

What are the two parameters that are included?

a) URL or machine name where server runs, Password, User ID

b) URL or machine name where server runs, User ID, Password

c) User ID, Password, URL or machine name where server runs
d) Password, URL or machine name where server runs, User ID
Answer:B

60. What are the major components of the JDBC?

A. DriverManager, Driver, Connection, Statement, and ResultSet

B. DriverManager, Driver, Connection, and Statement

C. DriverManager, Statement, and ResultSet

D. DriverManager, Connection, Statement, and ResultSet

Answer:A

61. Select the packages in which JDBC classes are defined?

A. jdbc and javax.jdbc

B. rdb and javax.rdb

C.jdbc and java.jdbc.sql

D. sql and javax.sql

Answer:D

62. ________ is an open source DBMS product that runs on UNIX, Linux and Windows.

A. MySQL

B. JSP/SQL

C. JDBC/SQL

D. Sun ACCESS

Answer:A

63. Which of the following class is derived from the container class?

(a) Component (b) Panel

(c) MenuComponent (d) List

Answer:

64. .Name of the class used to represent a GUI application window, which is optionally resizable and can have a title bar, an icon, and menus.

(a)Window (b)Panel

(c)Dialog (d)Frame

Answer:

65. Package of drawString() method is________

(a)java.applet (b)java.io

(c)javax.swing (d)java.awt

Answer:

66. Object which can store group of other object is called______

(a)Collection object (b)Java object

(c)Package (d)Wrapper

Answer:A

67. Which of these methods can be used to know which key is pressed?

(a) getModifier() (b)getActionKey

(c) getActionEvent() (d)getKey()

Answer:A

68. Swing components that don't rely on Native GUI are reffered to as __________

(a)Ligthweight component (b)heavy weight component

(c) GUI component (d) Non GUI component

Answer:A

69. When applet is dead, it automatically invokes the_____method when We quit the browser.

(a)Paint() (b) Stop()

(c) Destroy() (d) Final()

Answer:C

70. can you use setBackground() method to set the background color for ____?

(a) Component (b) Container

(c) JComponent (d) All three

Answer:A

71. Which of the following is used to interpret and execute Java Applet Classes Hosted by HTML?

(a)Appletviewer (b)Appletscreen

(c)Appletwatcher (d)Appletshow

Answer:A

72. ______package contains all the classes and methods required for Event handling in java.

(a) java.applet (b) java.awt

(c) java.event (d) java.awt.event

Answer:B

73. _____ method are used to register a mouse motion listener.

(a) addMouseO

(b) addMouseListenerO

(c) addMouseMotionListnerO

(d) eventMouseMotionListenerO

Answer:C

74. ____ is super class of all the events.

(a) EventObjeet (b) EventClass

(c) AetionEvent (d) ItemEvent

Answer:A

75. If scroll bar is manipulated______ event will be notified.

(a) AetionEvent (b) ComponentEvent

(c) AdjustmentEvent (d) WindowEvent

Answer:C

76. ______method can be used to obtain the command name for invoking ActionEvent object.

(a) getCommand() (b) getActionCommand()

(c) getActionEvent() (d) getActionEventCommand()

Answer:B

77. When the size of component is changed, __ event is generated.

(a) ComponentEvent (b) ContainerEvent

(c) FocusEvent (d) InputEvent

Answer:A

78. ____ is superclass of Container Event class.

(a) WindowEvent (b) ComponentEvent

(c) ItemEvent (d) InputEvent

Answer:B

79. The_______interface is used to handle the menu events.

(a) ContainerListener (b) FocusListener

(c) ActionListener (d) WindowListner

Answer:C

80. __________ creates a dropdown list of textual entries

a. Choice b. Checkbox c. Textbox d. TextComponent

Answer:A

ANNAMACHARYA INSTITUTE OF TECHNOLOGY & SCIENCES :: TIRUPATHI
AUTONOMOUS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name :** Object Oriented Programming Through Java	**AY:** 2021-2022
Object Oriented Programming Through Java Unit Wise Bit Bank			

QNO	Questions
	UNIT-1 (OOPS Basics, Class, Object, String Handling)
	2 Marks Questions
1	What is meant by Object Oriented Programming?
2	What is a constructor? What is the purpose of a constructor in a class?
3	What is the advantage of using “this” keyword, explain?
4	What are the core OOP’s concepts?
5	What are objects? How are they created from a class?
6	List some java keywords?
7	When do we declare the members of the class as static explain with example?
8	Explain what is meant by garbage collection?
9	What is a Method? What is the purpose of Methods in a class?
10	What is an array?
11	What is the difference between String and String Buffer?
12	What is static variable and static method?
13	What is short-Circuit operator?
14	What is Garbage collection?
15	What is arraycopy method? Explain with syntax.
16	What is wrapper class?
17	Name some JavaDoc Comments.
18	What is vector? How is it different from an array?
19	Give any 4 differences between C++ and Java.
	8 Marks Questions
1	Write about the Java language BUZZWORDS.
2	Explain the structure of Java program.
	Explain the working of Java Virtual Machine (JVM).
3	How many data types are in java? Explain with ranges.
	What is an operator? Explain different types of operators.
4	Explain briefly type Conversion and type casting.
	Explain the general Syntax of writing an application program in java. Also explain the steps to run an application Java programs.
5	What are the shortcomings of procedural programming? How does object oriented programming overcome those shortcomings?
6	Write a java Program to demonstrate Constructor Overloading.
7	Explain the various methods of Parameter passing.

8	Compare in terms of their Functions the following pairs of statements.
	for loop
	while & do-while.

9	Explain selection Statements with an example.

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name :** Object Oriented Programming Through Java	**AY:** 2021-2022
Object Oriented Programming Through Java Unit Wise Bit Bank			

10	Explain about String Handling functions with an example
11	Explain differences between StringBuffer and StringBuilder classes
12	Explain about static block and static methods in java
13	Illustrate with examples: static and final.
	UNIT-2 (Inheritance, Packages, Interfaces)
	2 Marks Questions
1	Explain the usage of Java packages.
2	What are different types of access modifiers (Access specifiers)?
3	What is an Object and how do you allocate memory to it?
4	What is a package? How to define a package?
5	What is the difference between this () and super ()?
6	Define super class and subclass?
7	What is meant by Inheritance?
8	What is meant by Static binding, Dynamic binding?
9	What is an Interface?
10	What is final modifier?
11	What are object wrappers? Give example.
12	Define polymorphism.
13	What are the methods under "object" class / java.lang.Object.
14	Explain toString method of object class.
15	What is reflection?
16	What is object cloning?
17	Write the application of proxies.
18	Differentiate shallow copy and deep copy in cloning.
19	
	8 Marks Questions
1	What are the different forms of inheritance? Explain.
2	Write a Java Program to implement method Overloading and method overriding?
3	Explain Abstract classes with an example program? Also describe the properties of abstract classes?
4	What is a Package? Explain the Packages with an example and how to import packages?
5	Describe interfaces & how to implement it with a Java Program?
6	Describe Dynamic Method dispatch and the use of final keyword in Java with relevant Programs?
7	What is multiple inheritance and how to perform it in Java?
8	Give a detailed sketch of the differences between Single, Multilevel & Hierarchial Inheritance?
9	Is there multiple inheritance in Java? If not, whether there is an alternative to implement Multiple inheritance?
10	Explain about inner classes and its importance in Java

Regulation: AK20	**Subject Code:** 20APC3004	**Subject Name :** Object Oriented Programming Through Java	**AY:** 2021-2022
Object Oriented Programming Through Java Unit Wise Bit Bank			

11	Develop a message abstract class which contains playMessage abstract method. Write a different sub-classes like TextMessage, VoiceMessage and FaxMessage classes for to implementing the playMessage method.
12	Develop an Interest interface which contains simpleInterest and compInterest methods and static final field of Rate 25%. Write a class to implement those methods.
13	Explain the different methods supported in Object class with example.
14	What is object cloning? Explain deep copy and shallow copy with examples.
15	Develop a abstract Reservation class which has Reserve abstract method. Implement the sub-classes like ReserveTrain and ReserveBus classes and implement the same.
16	Develop a Library interface which has drawbook(), returnbook() (with fine), checkstatus() and reservebook() methods. All the methods tagged with public.
17	Develop a static Inner class called Pair which has MinMax method for finding min and max values from the array.
18	Explain static nested class and inner class with examples.
19	What is the difference between an abstract class and an interface? What is the use of interface? Write a program in java to illustrate the use of an interface.
20	Explain the procedure to call super class members with examples
	UNIT-3 (Exception Handling, I/O Operations, Generic Programming)
	2 Marks Questions
1	What is Exception Handling?
2	What are the 2 methods by which we may stop threads?
3	What is the use of throw & throws clause?
4	What is an uncaught exception?
5	Describe Built in Exception?
6	What is the use of try and catch block in java?
7	What are runtime exceptions?
8	How to create custom exceptions?
9	How to create custom exceptions?
	8 Marks Questions
1	How the exceptions are handled in java? OR Explain exception handling mechanism in java?
2	what are checked and unchecked exceptions in java?
3	With a program illustrate user defined exception handling.
4	How to handle multiple catch blocks for a nested try block? Explain with an example.
5	What are the uses of ‘throw’ and ‘throws’ clauses for exception handling?
6	How Java handle overflows and underflows?

MultiThreading

1. What is the use of isAlive()? (2M)
2. What is multithreading? In how many ways Java implements multithreading? Explain one of these ways with suitable example. (4M)
3. Explain with example, suspending and resuming threads in Java. (4M)
4. What is a daemon thread? (2M)
5. How inter thread communication is done in Java? (4M)
6. Write a java program to create threads with different priorities. (4M)
7. Why thread is called a light weight process? What are the different things shared by different threads of a single process? What are the benefits of this sharing? (4M)
8. Mention any application for which multithreading is not desired. (2M)
9. Discuss about join keyword. (2M)
10. Explain the synchronization of multiple threads in Java with an example. (4M)
11. Explain thread states with block diagram. And, Associate methods to each state.
12. What is race condition? (2M)

a. A _____________ is a program that is executing.
b. ____________ multitasking is the feature that allows your computer to run two or more programs concurrently.
c. Single-threaded systems use an approach called an __________ with polling.
d. A thread's priority is used to decide when to switch from one running thread to the next is called a __________
e. Java's multithreading system is built upon the ____ class, its methods, and its companion interface, _____.
f. static Thread __________ method returns a reference to the thread in which it is called.
g. The sleep( ) method in Thread might throw an __________.
h. A _________ is a data structure that controls the state of a collection of threads as a whole.
i. The _______ method causes the thread from which it is called to suspend execution for the specified period of milliseconds.
j. To implement Runnable, a class need only implement a single method called ________
k. _________ method executes a call to run( ).
l. The ___________ method returns true if the thread upon which it is called is still running.
m. To set a thread's priority, use the __________ method, which is a member of Thread.
n. _______ wakes up a thread that called wait( ) on the same object.

o. What will be the output of the following Java code? What is the name of the thread in the following Java Program? What is the priority of the thread in the following Java Program?

```
class multithreaded_programing
    {
       public static void main(String args[])
       {
          Thread t = Thread.currentThread();
          System.out.println(t);
       }
    }
```

I/O Operations

1. What is a stream? Explain its types. (2M)
2. List classes that are used by Byte streams for input and output operation. (2M)
3. List classes that are used by character streams for input and output operation. (2M)
4. Illustrate PrintWriter Class to handle console output. (2M)
5. Write a program to copy one file to another using try with resources block.(4M)
6. Write a program to create and write a string in a file
7. WAP to display the contents of the file

a. A _______ is linked to a physical device by the Java I/O system.
b. __________ are used to perform all input & output operations in Java
c. _______________ class is used to read from byte array
d. System class contains three predefined stream variables: ________________________
e. ________ class contains the methods print() & println()
f. int read( ) throws ______________
g. ______________ streams uses Writer and Reader classes for input & output operations.
h. _____________ class is used to read from a file
i. FileInputStream(String fileName) throws ____
j. _____________ is returned by read() method is end of file (EOF) is encountered
k. The try-with-resources statement can be used only with those resources that implement the __________ interface defined by java.lang.
l. AutoCloseable is inherited by the ___________ interface in java.io.
m. The list of suppressed exceptions can be obtained by using the ____________ method defined by Throwable.

n. What will be the output of the following Java program if input given is 'abcqfghqbcd'?

```
    class Input_Output
    {
       public static void main(String args[]) throws
IOException
       {
          char c;
          BufferedReader obj = new
BufferedReader(new
InputStreamReader(System.in));
          do
          {
             c = (char) obj.read();
```

```
                System.out.print(c);
            } while(c != 'q');
        }
    }
```

o. What will be the output of the following Java program if input given is “abc’def/’egh”?

```
    class Input_Output
    {
        public static void main(String args[]) throws
IOException
        {
            char c;
            BufferedReader obj = new
BufferedReader(new
InputStreamReader(System.in));
            do
            {
                c = (char) obj.read();
                    System.out.print(c);
            } while(c!='\'');
        }
    }
```

p. What will be the output of the following Java program?

```
    class output
    {
        public static void main(String args[])
        {
            String a="hello i love java";
            System.out.println(indexof('i')+"
"+indexof('o')+" "+lastIndexof('i')+" "+lastIndexof('o')
));
        }
    }
```

q. What will be the output of the following Java program?

```
    class output
    {
        public static void main(String args[])
        {
            char c[]={'a','1','b',' ','A','0'];
            for (int i = 0; i < 5; ++i)
            {
                    if(Character.isDigit(c[i]))
                     System.out.println(c[i]" is a digit");
                if(Character.isWhitespace(c[i]))
                    System.out.println(c[i]" is a
Whitespace character");
                if(Character.isUpperCase(c[i]))
                    System.out.println(c[i]" is an Upper
case Letter");
                if(Character.isUpperCase(c[i]))
                    System.out.println(c[i]" is a lower case
Letter");
                i = i + 3;
            }
        }
    }
```

r. What will be the output of the following Java program if input given is “Hello stop World”?

```
class Input_Output
    {
        public static void main(String args[]) throws
IOException
        {
            string str;
            BufferedReader obj = new
BufferedReader(new
InputStreamReader(System.in));
            do
            {
                str = (char) obj.readLine();
                    System.out.print(str);
            } while(!str.equals("strong"));
        }
    }
```

s. What will be the output of the following Java program?

```
    class output
    {
        public static void main(String args[])
        {
          StringBuffer s1 = new StringBuffer("Hello");
          s1.setCharAt(1,x);
          System.out.println(s1);
        }
    }
```

Generic Programming

1. What are Generics? (2M)
2. Write a program to Illustrate the working of Generic class in Java. (4M)
3. Write a program to Illustrate the working of generic method in Java (4M)
4. Implement bounded type parameters using extends Comparable object. (4M)
5. Write a program to illustrate the working of generic constructors (4M)
6. What are the restrictions to be remembered while using generics. (2M)

a. generics means _____________

b. ___________ is the superclass of all other classes

c. The syntax for declaring a generic class: ____________________________________

d. The full syntax for declaring a reference to a generic class and instance creation: ________________________

e. A class that implements ___________ defines objects that can be ordered.

Collection Framework

1. What is collection Framework?(2M)
2. What are legacy classes? (2M)
3. Draw the hierarchy of collection framework. (2M)

4. WAP to traverse ArrayList elements using the Iterator interface. (2M)
5. WAP to add and traverse elements in a Linked List. (2M)
6. WAP to insert, remove and display elements in a stack. (4M)
7. WAP to insert, remove and display elements in a queue. (4M)
8. Demonstrate the methods in StringTokenizer class with an example. (4M)

a. The ________________ package contains the Collections Framework

b. An __________ offers a general-purpose, standardized way of accessing the elements within a collection, one at a time

c. ___________ class object can be used to form a dynamic array

d. ___________ is the interface of legacy

e. ArrayList class implements a dynamic array by extending ____________ class.

f. ______________ method can be used to increase the capacity of ArrayList object manually

g. What will be the output of the following Java program?

```
import java.util.*;
class Arraylist
{
    public static void main(String args[])
    {
        ArrayList obj = new ArrayList();
        obj.add("A");
        obj.add("B");
        obj.add("C");
        obj.add(1, "D");
        System.out.println(obj);
    }
}
```

h. What will be the output of the following Java program?

```
import java.util.*;
class Output
{
    public static void main(String args[])
    {
        ArrayList obj = new ArrayList();
        obj.add("A");
        obj.ensureCapacity(3);
        System.out.println(obj.size());
    }
}
```

Applets

1. What is an applet? (2M)
2. Write a simple applet that displays "hello world" and explain its working.(4M)
3. Explain the concept of applet to applet communication with suitable program. (4M)
4. What for repaint( ) method is used? (2M)
5. Differentiate between paint() and repaint() functions. (2M)
6. Present the structure of a Java Applet (2M)
7. Discuss in detail about Applet life cycle. (4M)
8. How to pass the parameters to an Applet? Explain with example. (4M)

a. Applets do not need a ________ method.

b. The ____________ method is the first method to be called in an applet.

c. The __________ method is called each time an AWT-based applet's output must be redrawn.

d. Whenever the applet requires to redraw its output, it is done by using method _____________

e. ___________ method is defined in Graphics class, it is used to output a string in an applet.

Swings

1. Explain the delegation event model. (4M)
2. Explain procedure to handle mouse events in Java with an example program. (4M)
3. Design a Java front end for a login page using swing components with the Gridlayout manager (4M)
4. Discuss about list box and choice boxes. (2M)
5. Explain Jtree Swing components with suitable example. (4M)
6. What is MVC architecture in Java? (4M)
7. What are the advantages of MVC architecture? (2M)
8. Write a Java Swing Program to design a form that contains button, text field, password field, radio button and combo button in a frame. (8M)

a. The ActionListener Interface defines the _________ method that is invoked when an action event occurs.

b. public class JButton extends _____________ implements _____________

c. public class JLabel extends _____________

d. public class JPasswordField extends ______________

e. The BorderLayout provides ___________ constants for each region

f. The Java ________________ class is used to arrange the components in a rectangular grid.

g. The BoxLayout class provides __________ constants.

h. The Java _____________ class manages the components such that only one component is visible at a time.

Code: 15A05403

R15

B.Tech II Year II Semester (R15) Regular Examinations May/June 2017

OBJECT ORIENTED PROGRAMMING USING JAVA

(Common to CSE & IT)

Time: 3 hours Max. Marks: 70

PART - A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) Explain about commands javac, java.

(b) List any four predefined packages in java.

(c) What is multitasking?

(d) Define an event in java.

(e) Demonstrate the use of "?" operator.

(f) Differences between the object oriented program and procedural oriented programming.

(g) Explain about Bitwise operators in java.

(h) Explain the normal flow of a thread with neat diagram.

(i) List out event sources.

(j) Explain parameter passing methods in java.

PART - B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT - I

2 (a) Explain briefly buzzwords of java.

(b) Explain any four object oriented programming features.

OR

3 (a) Explain about arrays in java with an example program.

(b) Write a java program to perform matrix multiplication.

UNIT - II

4 (a) Explain about StringTokenizer class in java with example.

(b) In how many ways a package can be imported. Explain with an example program.

OR

5 (a) What is a constructor? Explain constructor overloading with an example.

(b) What is a method? Explain method overloading with example.

UNIT - III

6 (a) Define a package. Write down the steps to create a package.

(b) Define an interface. Explain about implementing an interface with example.

OR

7 (a) What is an exception? Explain various exception types.

(b) Write a java program using all keywords of exception handling.

Contd. in page 2

UNIT - IV

8 (a) Write a java program that creates a thread by extending the thread class.

(b) Explain about thread priorities in java with suitable example.

OR

9 (a) Explain about the ways to create an applet with example.

(b) How to pass parameters to an applet? Explain with an example.

UNIT - V

10 (a) List and explain various AWT components in java.

(b) Explain about event delegation model.

OR

11 Explain the following layout managers.

(a) Border layout.

(b) Grid layout.

(c) Flow layout.

*****

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Java Programming
Model Paper – R13

I Semester

Duration: 3hrs **Max Marks: 75**

Answer all the following

PART-A **(Marks 25)**

1. (a) What are the properties of object oriented programming?

(b)what is method overriding?

(c)Define an Exception.What is meant by Exception Handling?

(d)List some of the classes available in collection?

(e)List the compponents of Swing?

(f)Discuss briefly about streams.

(g)What is inheritance?

(h)What is thraed priority?

(i)What are the steps involved in connecting the database?

(j)What is an event?

Answer all the questions either (a) or (b)

PART – B **(Marks: 5*10=50)**

2. (a)Discuss in detail about inheritance. Also write its benefits.

(OR)

(b)Describe about Type conversion. Also explain how casting is used to perform type conversion between incompatible types.

3. (a) What is inheritance ? Explain different types of inheritance.

(OR)

(b) How a method can be overridden? Explain.

4. (a) Give the class hierarchy in Java related to exception handling. Briefly explain each class.

(OR)

(b)What is a thread? Explain the states of a thread with an example.

5. (a) Explain in detail about collection interfaces.

(OR)

(b) Explain in details about primary input and output operations.

6. (a) Explain in detail about the classification of swing components.

(OR)

(b)Explain in brief about events and event sources.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Answer all the following

PART-A **(Marks 25)**

1 (a) Discuss briefly about recursion.
(b) Define package
(c) Differences between multi tasking and multi threading.
(d) Discuss briefly about hash table class.
(e) Explain in brief about layout manager.
(f) What is an operator? list various tupes.
(g) List different types of access specifies.
(h) List the keywords used to handle exceptions.
(i) Define character streams.
(j) Define Applet.

PART – B **(Marks:5*10=50)**

Answer all the questions (Either (a) or (b))

2 (a)What is constraint explain the constant types with examples.

(OR)

(b)What is a method ?How a method is used in the class? Explain.

3 (a) Explain the usage of Abstract classes and methods.

(OR)

(b)Discuss how inheritances are defined and implemented.

4 (a)What is multithreading? Explain.

(OR)

(b)What is synchronization? Explain with suitable example.

5 (a)Write short notes on the following collection framework classes.
1) Random 2) Scanner

(OR)

(b)Write a short notes on
1) Connection interface
2)Statement object
3)Inner join
4)Execute Query Method.

6 (a)Write a simple awing application in java.

(OR)

(b)Write the difference between applets and applications.

Java Programming
Model Paper –3 (R13)
II CSE II Semester

Duration: 3hrs **Max Marks: 75**

Answer all the following

PART-A **(Marks 25)**

1. (a) List the data typed]s present in java.
   (b)Explain in brief about interfaces.
   (c)What is meant by checked exception and unchecked exception.
   (d)How statements call can be used? Also list the types of methods in statement class.
   (e)Discuss about Jframe and Jpanel.
   (f)Discuss briefly about enumerated data types.
   (g)what is CLASSPATH.
   (h)What is multithreading?
   (i)List the types of JDBC drivers present in java.
   (j)What are event sources?

PART – B **(Marks:5*10=50)**

Answer all the questions (Either (a) or (b))

1. (a)List the primitive data types of java. Explain each of them in detail.

   (OR)

   (b)What are the different types of array? List out the advantages of using arrays?

2. (a)Write in detail about super class and subclasses.

   (OR)

   (b)Write the differences between interfaces and abstract.

3. (a)How are finally statements used in java? Explain in detail.

   (OR)

   (b)Is it possible to interrupt a thread? Explain.

4. (a) Explain inn detail about hash table class.

   (OR)

   (b) Explain in detail about the types of drivers in JDBC.

5. (a) Discuss in detail about swing components.

   (OR)

   (b)Explain about various event classes.

Java Programming
Model Paper –4 (R13)
II CSE II Semester

Max Marks: 75

Answer all the following

PART-A **(Marks 25)**

1. (a) What are the OOPs features?

(b) Compare Procedural and OOP Languages?.

(c) Explain about control statements in java?.

(d) Explain about method overloading with example?

(e) Explain about the usage of super keyword with an example?

(f) Explain how interfaces are implemented with an example?.

(g) Explain the following: try, catch, throw, throws, finally

(h) Explain the creation of threads with an example?

(i)List the types of JDBC drivers present in java.

(j)What are event sources.and Explain the life cycle of an applet?

PART – B **(Marks:5*10=50)**

Answer all the questions (Either (a) or (b))

2. (a) What is type casting and conversion? When it is required?

(b). What is an array? How arrays are declared in javawith an example?

(OR)

(c) Explain about method overloading with example? Explain about constructor overloading with example?

3 (a) What is method overriding? How methods overriding is achieved in Java, with an example?.

(OR)

(b) How multiple inheritances are achieved in java with the interfaces? Explain with an example?

4 (a) What are the checked Exceptions and Unchecked Exceptions? Explain some of these exceptions with an example and also give the difference between them.

(OR)

(b) How the priorities can be assigned to threads? Explain with example?

5 (a) Explain the difference between: i) Vector and Array List. ii) Enumeration and Iterator.

(OR)

(b)Explain in deatil about the types of drivers in JDBC.

6 (a) Define event. Give examples of events. Define event handler. How it handles events?

(OR)

(b) Explain about layout manager? With an example?.

Code No: R09220503

R09

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY, HYDERABAD

B. Tech II Year - II Semester Examinations, November/December, 20

OBJECT ORIENTED PROGRAMMING

(COMMON TO CSE, IT)

Time: 3 hours **Max. Ma**

Answer any five questions
All questions carry equal marks

1.a) What are the shortcomings of procedure oriented programming? Explain how does object oriented programming overcome these shortcomings.

b) What is encapsulation? Explain with the help of an example using JAVA. [10+5]

2.a) Explain copy constructor? Write a program which reads complex numbers and copy that into another. Use copy constructor for writing program?

b) Differentiate between regular variable and automatic variable? Explain other storage classes for variables. [8+7]

3. Write a superclass Worker and subclasses Hourly Worker and Salaried Worker. Every worker has a name and a salary rate. Write a method computePay (int hours) that computes the weekly pay for every worker. An hourly worker gets paid the hourly wage for the actual number of hours worked, if hours is at most 40. If the hourly worker worked more than 40 hours, the excess is paid at time and a half. The salaried worker gets paid the hourly wage for 40 hours, no matter what the actual number of hours is. Write a static method that uses polymorphism to compute the pay of any worker. Supply a test program that tests these classes and methods. [15]

4.a) Define an interface using JAVA that contains a method to calculate the perimeter of an object. Define two classes-circle and rectangle with suitable fields and methods. Implement the interface "perimeter" in these classes. Write the appropriate main() method to create object of each class and test all the methods.

b) Discuss basic in-built packages in java and their uses in application development in brief. Which package is the default package? [9+6]

5.a) What is importance of exception handling mechanism in Java? Define and distinguish checked and unchecked exceptions.

b) Create a try block that will generate three types of exception and also create necessary catch blocks to catch these exceptions and handle these. You should also use finally statement in your block. [8+7]

6.a) How inter thread communication is done in Java? Write a java program to create multithreads with different priorities.

b) Explain Multithreading. In how many ways java implements multithreading? Explain at least one of these ways with appropriate example.

7.a) Write a java program to create a frame with exit capabilities. Handl mouse pressed, mouse released, mouse dragged, and mouse clicked b appropriate message. Print the co-ordinates at which the event took pla

b) Explain layout types in AWT with the help of an example.

8. Design an Applet to display three buttons "Red", "Green" and "Blue". The color of the background changes according to the button pressed by the user. Also, write the HTML code to display the Applet. [15]

Code No: R09220503 **R09** **Set No. 2**

II B.Tech II Semester Examinations,December-January, 2012
OBJECT ORIENTED PROGRAMMING
Common to Information Technology, Computer Science And Engineering, Electronics And Communication Engineering, Electrical And Electronics Engineering

Max Marks: 75

Answer any FIVE Questions
All Questions carry equal marks

⋆⋆⋆⋆⋆

1. (a) Discuss member access using Inheritance.
   (b) What are the advantages of Inheritance? [8+7]

2. (a) Explain about various expressions in java.
   (b) Discuss in detail about type conversion and casting. [7+8]

3. (a) What is meant by uncaught exceptions? Explain it with suitable examples.
   (b) Explain with suitable examples about the usage of *try* and *catch* statements. [7+8]

4. What is meant by encapsulation? Explain how encapsulation is achieved in JAVA. [15]

5. (a) State and explain the simple Applet display methods.
   (b) Write Java code for a simple Banner Applet and explain it. [7+8]

6. (a) Give illustrations on explicit interface member implementations.
   (b) How Interfaces can be validated? Explain with example. [7+8]

7. What is meant by virtual key codes? Write a sample Java program to demonstrate some virtual key codes. [15]

8. (a) Discuss about wait( ), notify( ) and notifyAll( ) methods in Java.
   (b) What is meant by a Deadlock? Is it possible to occur in a multithreaded program? Justify your answer with a sample Java program. [7+8]

⋆⋆⋆⋆⋆

Code No: R09220503 **R09** **Set No. 4**

II B.Tech II Semester Examinations,December-January, 2012
OBJECT ORIENTED PROGRAMMING
Common to Information Technology, Computer Science And Engineering, Electronics And Communication Engineering, Electrical And Electronics Engineering

Time: 3 hours **Max Marks: 75**

Answer any FIVE Questions
All Questions carry equal marks
⋆⋆⋆⋆⋆

1. (a) Briefly explain Stream Tokenizer class.
   (b) Write a program to illustrate the use of stream tokenizer class. [8+7]

2. Discuss about various methods defined in the following classes:
   (a) ImageIcon
   (b) JLabel
   (c) JTextField
   (d) JButton [15]

3. (a) Write short notes on history of Java.
   (b) What is a data type? Explain various data types available in Java. [7+8]

4. (a) Write a Java program to concatenate two given strings.
   (b) Write a Java program to convert all the letters in a string to upper case.
   (c) Discuss the usage of the method "valueof( )" with an example. [5+5+5]

5. (a) How to request a service from an object through a message?
   (b) Write a simple Java program to display a message "welcome to oops" by using the member function creating an object to call particular member function. [7+8]

6. (a) Explain in detail about various forms of inheritance?
   (b) Discuss about Specialization in detail. [7+8]

7. Discuss in detail about Menu bars, Menus and Dialog boxes. [15]

8. Discuss in detail about various Enumerations and Annotations in Java. Write sample Java programs to describe each of them. [15]

⋆⋆⋆⋆⋆

Code No: R09220503 **R09** **Set No. 1**

II B.Tech II Semester Examinations,December-January, 2012
OBJECT ORIENTED PROGRAMMING
Common to Information Technology, Computer Science And Engineering, Electronics And Communication Engineering, Electrical And Electronics Engineering

Time: 3 hours **Max Marks: 75**

Answer any FIVE Questions
All Questions carry equal marks

⋆ ⋆ ⋆ ⋆ ⋆

1. (a) Explain about type conversion with an example.
   (b) Explain about wrapper class with an examples. [7+8]

2. Discuss the following string operations:
   (a) String Literals
   (b) String concatenation
   (c) String conversion
   (d) String constructors [15]

3. (a) What is meant by Auto-Unboxing? Explain with an example.
   (b) Discuss clearly about type wrappers in Java. [8+7]

4. (a) Does a super class variable be used to refer a sub class object. Explain with an example.
   (b) Explain about the extension of a class from another class with a program. [8+7]

5. (a) Discuss various constructors for FlowLayout? Explain them.
   (b) Write a sample Java program to demonstrate the usage of BoarderLayout. [7+8]

6. List out various Swing component classes and explain them clearly. [15]

7. (a) Discuss in detail about Buffered Byte Stream class with an example.
   (b) Demonstrate Sequence Input Stream with a program.
   (c) Write down the benefits of streams in java. [7+5+3]

8. Explain following terms with an example for each
   (a) abstraction
   (b) information hiding
   (c) dynamic binding
   (d) reusability. [15]

⋆ ⋆ ⋆ ⋆ ⋆

Code No: R09220503 **R09** **Set No. 3**

II B.Tech II Semester Examinations,December-January, 2012
OBJECT ORIENTED PROGRAMMING
Common to Information Technology, Computer Science And Engineering, Electronics And Communication Engineering, Electrical And Electronics Engineering

Time: 3 hours **Max Marks: 75**

Answer any FIVE Questions
All Questions carry equal marks

⋆⋆⋆⋆⋆

1. State and explain various constructors and methods defined in the JTree, JScrollPane and JComboBox classes. [15]

2. (a) Explain the term “Annotation” in Java. Discuss how to specify the annotation retention policy.
   (b) Discuss in detail about the single-member annotations with a sample Java program. [8+7]

3. (a) Write a program illustrating ‘this’ keyword. Explain it.
   (b) Write short note on garbage collection. [7+8]

4. (a) What is inheritance? How inheritance promotes software reuse?
   (b) How to create class by inheriting from existing class? [7+8]

5. (a) Discuss about various methods defined by Pleader and Writer classes.
   (b) Write a program that illustrates file writes. [8+7]

6. (a) Give illustration on finalize( ), equals and to string( ) methods.
   (b) Define substitutability and discuss its needs. [8+7]

e various adapter classes that implements commonly used Listener
Vrite a sample Java program to demonstrate an Adapter. [15]

8. What are the Java’s built-in exceptions? List the checked exceptions defined in the *Java.lang* and explain them clearly with suitable examples. [15]

⋆⋆⋆⋆⋆

Code: 13A05403 **R13**

B.Tech II Year II Semester (R13) Supplementary Examinations December 2016

JAVA PROGRAMMING

(Common to CSE and IT)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) What is the role of JVM?

(b) Compare and contrast equals ( ) versus = =.

(c) Discuss garbage collection.

(d) Demonstrate the usage of super keyword.

(e) Explain throws clause.

(f) Discuss Byte Array Output Stream.

(g) Explicate the life cycle of a thread.

(h) Why do we need cookies?

(i) Discuss components and containers of swings.

(j) What are the two swing key features?

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 Write a program for matrix multiplication using two dimensional arrays.

OR

3 (a) Discuss string constructors.

(b) Differentiate between string and string buffer with an apt example, signifying their usage.

UNIT – II

4 What is the difference between an abstract class and an interface? What is the use of interface? Write a program in java to illustrate the use of an interface.

OR

5 Define a package. Why do we need packages? Illustrate with suitable example, class member access with respect to packages.

UNIT – III

6 Describe the java throwable class hierarchy and the types of exceptions. Can you claim multiple exceptions in a method declaration? Illustrate by means of an example.

OR

7 (a) What is the difference between the reader and writer classes? Show the stream class hierarchy as defined in java.io.package.

(b) Discuss the stream benefits.

UNIT – IV

8 Illustrate with an example, the two ways we can synchronize the code when two or more threads are trying to access it.

OR

9 Discuss Applet's life cycle methods. Illustrate with an example passing parameters to applets.

UNIT – V

10 What are swings in java? Write a program in java to illustrate the use of swing controls.

OR

11 Compare and contrast Java AWT and Java Swing. Give a brief synopsis of methods and their description of component class widely used in swings.

*****

Code: R7411302

R07

B.Tech IV Year I Semester (R07) Supplementary Examinations, May 2013

OBJECT ORIENTED PROGRAMMING

(Electronics & Control Engineering)

Time: 3 hours Max. Marks: 80

Answer any FIVE questions
All questions carry equal marks

*****

1. (a) Explain various data types in java. Write at least 8 types with examples for each.
   (b) What is meant by type conversion and type casting? List out the differences between the two.

2. (a) What is an object? Explain how do we create objects in java? Explain with suitable example.
   (b) What are constructors and destructor functions? Explain different types of constructors.

3. (a) Explain the peculiarity of java in providing multiple inheritances.
   (b) List out the differences between method overloading and method overriding.

4. (a) Explain the way of implementing exploration of package using java.io.
   (b) Discuss the various levels of access protection available for package and their implications.

5. (a) Give the list of unchecked exceptions in java with their meaning.
   (b) Explain multithreading by extending thread class. Give an example.

6. What are the methods supported by mouse listener and mouse motion listener interface? Explain each of them with examples.

7. (a) Explain the life cycle of an applet.
   (b) List out the differences between an applet program and application program.

8. Explain the following:
   (a) String handling.
   (b) Datagram's.
   (c) Inet addresses.

*****

R07

Code: R7220506

B.Tech II Year II Semester (R07) Supplementary Examinations December/January 2015/2016

OBJECT ORIENTED PROGRAMMING

(Common to CSE, IT, ECC & CSS)

(For 2008 regular admitted batch only)

Time: 3 hours Max. Marks: 80

Answer any FIVE questions

All questions carry equal marks

*****

1 (a) What is the difference between object based and object oriented programming?
(b) How does java differ from C language?

2 (a) Write a java program to find greatest common divisor of given two numbers.
(b) Write a java program to check whether the given number is palindrome or not.

3 (a) What is abstract class? Explain with an example.
(b) Explain the procedure to call super class members with examples.

4 (a) Write a brief note on variables in interfaces.
(b) Discuss briefly about Math class.

5 (a) Explain multithreading by creating multiple threads (more than two).
(b) What is the use of Alive ( ) and join ( ) functions in multithreading?

6 What is the task performed by Layout manager? Explain different layout managers.

7 (a) Explain the use of JTable class with an example.
(b) What are the mandatory attributes of applet tag? Explain them.

8 (a) Explain the usage of URL Connection class with an example.
(b) Explain any two classes available in java.util package with suitable examples.

*****

Code: 9A05402

R09

B.Tech II Year II Semester (R09) Supplementary Examinations December/January 2015/2016

OBJECT ORIENTED PROGRAMMING

(Common to CSS, IT & CSE)

Time: 3 hours Max. Marks: 70

Answer any FIVE questions
All questions carry equal marks

*****

1 (a) Explain the need for object oriented programming paradigm.
(b) What are the components of java architecture? Explain in detail.
(c) Describe with a flow chart, how various tools are used in application development.

2 (a) What is a method? Write the differences between method and constructor.
(b) Explain “this” keyword.

3 (a) Explain dynamic method dispatch with an example.
(b) List and explain the methods defined in the Object class.

4 Explain the classes:
(a) CharacterArrayWriter.
(b) BufferedReader.
(c) BufferedWriter.
(d) Pushbackreader.

5 (a) What is synchronization? Why is thread synchronization important for multithreaded programs?
(b) What is multitasking? Give an example.

6 (a) Describe about the Canvas component in AWT.
(b) Explain about Scrollbar AWT control.

7 (a) Explain the use of JTable class with an example.
(b) What are the mandatory attributes of applet tag? Explain them.

8 (a) Discuss about internet addressing in TCP/IP.
(b) What is the use of Inet Address class?

*****

Code: 9A05402

R09/SS

B.Tech II Year II Semester (R09) Supplementary Examinations May/June 2016

OBJECT ORIENTED PROGRAMMING

(Common to ECE, CSS, IT & CSE)

Time: 3 hours Max. Marks: 70

Answer any FIVE questions
All questions carry equal marks
*****

1 Distinguish the following terms:
(a) Objects and classes.
(b) Data abstraction and data encapsulation.

2 (a) Write a java program to find greatest common divisor of given two numbers.
(b) Write a java program to check whether the given number is palindrome or not.

3 (a) Compare method overloading with method overriding.
(b) Define inheritance and write its uses.
(c) Write short notes on early binding and late binding.

4 (a) Explain about StringTokenizer class?
(b) Write a java program to find Date and Time.

5 (a) Give the general form of an exception handling block.
(b) What are the consequences that occur when having multiple catch clauses?

6 (a) What is an InputEvent? Briefly discuss its sub classes.
(b) Describe any four window events.

7 Explain the following swing control with an example:
(a) JCheckBox.
(b) JradioButton.

8 Explain sockets 'send', write a simple file transfer server program in java that takes input file name from client and contents of the file to the client.

*****

R13

Code: 13A05403

B.Tech II Year II Semester (R13) Regular & Supplementary Examinations May/June 2016

JAVA PROGRAMMING

(Common to CSE and IT)

Time: 3 hours Max. Marks: 70

PART - A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) What is JIT?

(b) Write short note on JDK.

(c) What is type conversion in java?

(d) What is constructor?

(e) What is super keyword?

(f) What is wrapper?

(g) Define multithreaded programming.

(h) List the inetaddress class methods.

(i) Define cookies.

(j) What is socket?

PART - B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT - I

2 Explain working of java virtual machine (JVM) also explain how java is architectural neutral.

OR

3 Explain data types in java.

UNIT - II

4 What is inheritance? Explain the different types of inheritance supported by java with example program.

OR

5 Explain the constructor and method overloading in java.

UNIT - III

6 What is exception handling? Explain how exceptions are handled in java programming.

OR

7 Explain the process of reading the contents of a file by using file input stream class in java with suitable program.

UNIT - IV

8 Define multithreading. Explain with an example of an application that needs multithreading.

OR

9 What is TCP? Explain the process of creating TCP connections in client and server side in java programming.

UNIT - V

10 What is swing component? Explain any three components of swing with syntax.

OR

11 How to create a main menu by using swings with suitable program?

*****

Code: 9A05402

R09

B.Tech II Year II Semester (R09) Supplementary Examinations May/June 2017

OBJECT ORIENTED PROGRAMMING

(Common to CSS, IT & CSE)

Time: 3 hours Max. Marks: 70

Answer any FIVE questions
All questions carry equal marks

*****

1 (a) Differentiate message passing and procedure call.
(b) Explain different types of inheritance.

2 (a) Develop a java program to simulate the working of simple calculator operations.
(b) For any given triangle, the length of the hypotenuse is given by the square root of the sum of squares of other two sides. Develop a program that calculates the longest side when other two sides are provided as arguments.

3 (a) What is dynamic method dispatch? Write a java program to illustrate dynamic method dispatch.
(b) Explain the procedure to call super class members with examples.

4 (a) How is polymorphism achieved in java? Write a program to show the same.
(b) Write a sample program to illustrate packages.

5 (a) Define synchronization. Write a program to solve producer-consumer problem.
(b) Describe any two types of exceptions with example program.

6 Explain AWT user interface components in detail.

7 (a) List the limitations of AWT. Describe the MVC architecture.
(b) Write an example and explain JApplet.

8 Write note on the following:
(a) Difference between ports and sockets.
(b) Enumerations and auto boxing.

*****

Code: 13A05403

R13

B.Tech II Year II Semester (R13) Supplementary Examinations December/January 2015/2016

JAVA PROGRAMMING

(Common to CSE and IT)

Time: 3 hours Max. Marks: 70

PART – A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) State any four features of java.

(b) What is byte code? Explain why java is called as true object oriented language.

(c) Describe general structure of Java class with example.

(d) Explain method overloading with Java example.

(e) What is four major differences between interface & class?

(f) What is exception? How it is handled? Explain with suitable example.

(g) What is synchronization? Explain with suitable example.

(h) Differentiate between java applet & java application.

(i) Explain TCP/IP client socket & TCP/IP server sockets.

(j) Discuss Components & containers in swings.

PART – B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT – I

2 What is operator? Explain in detail arithmetic operator, bitwise operator, incremental & decrement operator with proper example. Write a program to demonstrating increment & decrement operator using conditional operator.

OR

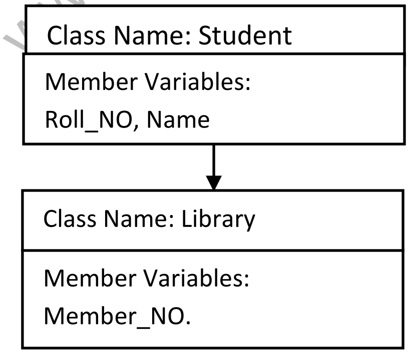
3 What is array? Discuss one dimensional & two dimensional arrays. Write a program for declaring & initializing two dimensional arrays.

UNIT – II

4 What are constructors & parameterized constructors? Explain in detail with example. Write a program for constructor to use 'this' keyword.

OR

5 Explain different types of inheritance in detail with example. Write a program to implement following inheritance, assume suitable method.



Contd. in page 2

UNIT – III

6 Explain the following terms in detail with respect to exception handling: Try, catch, and throw, finally. Write a program using try, catch, and throw, finally statement

OR

7 Explain classification of stream classes. Explain various types of byte stream classes & character stream classes in detail.

UNIT – IV

8 What is thread? Explain in detail about life cycle of thread with diagram. Explain resuming & stopping threads.

OR

9 What is an applet? Explain in detail about applet life cycle with suitable diagram. Write a program to draw circle & rectangle filled with red color.

UNIT – V

10 Explain Jtree & give the sequential steps to use Jtree control of swing, also give syntax of four constructors of Jtree class.

OR

11 Explain the following terms in detail with examples:
(i) Component.
(ii) Container.
(iii) Layout managers.

*****

Code: 13A05403

R13

B.Tech II Year II Semester (R13) Supplementary Examinations May/June 2017

JAVA PROGRAMMING

(Common to IT & CSE)

Time: 3 hours Max. Marks: 70

PART - A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) What is overridden method?
(b) What are the logical operators?
(c) What is operator?
(d) What is function overloading?
(e) Define switch statement.
(f) What is overloading constructor?
(g) Define nested loops.
(h) What are try and catch keywords in java? Explain.
(i) What is synchronization?
(j) What is deadlock?

PART - B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT - I

2 (a) Explain object oriented programming.
(b) Explain control statements in java.

OR

3 (a) Explain iteration using multidimensional array in java.
(b) Write a java program for factorial of a given number n using recursion.

UNIT - II

4 (a) Write java program to add methods width, height and length for box class.
(b) Explain implementation of nested interfaces in java.

OR

5 (a) Write a java program subclass contain cube for super class contain width, height and length for inheritance.
(b) Explain multithreading in java.

UNIT - III

6 (a) Explain exceptional handling with an example program.
(b) Explain generic interfaces.

OR

7 (a) Explain reading and writing strings in java with an example program.
(b) Explain reading and writing files in java.

Contd. in page 2

UNIT - IV

8 (a) Explain creation of threads in Java with an example program.

(b) Write thread communication.

OR

9 (a) Explain parameter passing in applets with an example program.

(b) Explain handling mouse events.

UNIT - V

10 (a) Explain java swings different buttons with an example program.

(b) Explain network interfaces.

OR

11 (a) Write java program to create main menu and drawing rectangle.

(b) Explain event handling using swings.

*****

Code: 15A05403

R15

B.Tech II Year II Semester (R15) Regular Examinations May/June 2017

OBJECT ORIENTED PROGRAMMING USING JAVA

(Common to CSE & IT)

Time: 3 hours Max. Marks: 70

PART - A

(Compulsory Question)

*****

1 Answer the following: (10 X 02 = 20 Marks)

(a) Explain about commands javac, java.

(b) List any four predefined packages in java.

(c) What is multitasking?

(d) Define an event in java.

(e) Demonstrate the use of “?” operator.

(f) Differences between the object oriented program and procedural oriented programming.

(g) Explain about Bitwise operators in java.

(h) Explain the normal flow of a thread with neat diagram.

(i) List out event sources.

(j) Explain parameter passing methods in java.

PART - B

(Answer all five units, 5 X 10 = 50 Marks)

UNIT - I

2 (a) Explain briefly buzzwords of java.

(b) Explain any four object oriented programming features.

OR

3 (a) Explain about arrays in java with an example program.

(b) Write a java program to perform matrix multiplication.

UNIT - II

4 (a) Explain about StringTokenizer class in java with example.

(b) In how many ways a package can be imported. Explain with an example program.

OR

5 (a) What is a constructor? Explain constructor overloading with an example.

(b) What is a method? Explain method overloading with example.

UNIT - III

6 (a) Define a package. Write down the steps to create a package.

(b) Define an interface. Explain about implementing an interface with example.

OR

7 (a) What is an exception? Explain various exception types.

(b) Write a java program using all keywords of exception handling.

Contd. in page 2

UNIT - IV

8 (a) Write a java program that creates a thread by extending the thread class.

(b) Explain about thread priorities in java with suitable example.

OR

9 (a) Explain about the ways to create an applet with example.

(b) How to pass parameters to an applet? Explain with an example.

UNIT - V

10 (a) List and explain various AWT components in java.

(b) Explain about event delegation model.

OR

11 Explain the following layout managers.

(a) Border layout.

(b) Grid layout.

(c) Flow layout.

*****