GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

OOPs with JAVA (BCS403)

Unit: I

Subject Name : Object Oriented Programming with Java (BCS-403)

(B.Tech 4th Sem)

Dr. Birendra Kr. Saraswat
Associate Professor
Department of IT

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Content

- Course Objective
- Course Outcomes
- CO-PO Mapping
- CO-PSO Mapping
- Syllabus
- Prerequisite
- Basic Concept of OOPS

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Content

- Design Strategies
- Software Measurement and Metrics
- Cyclomatic Complexity Measures
- Video Links
- Daily Quiz
- Weekly Assignment
- MCQ
- Old Question Papers
- Expected Questions for University Exam
- Summary
- References

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Syllabus

Unit	TOPIC
I	Introduction: Why Java, History of Java, JVM, JRE, Java Environment, Java Source File Structure, and Compilation. Fundamental, Programming Structures in Java: Defining Classes in Java, Constructors, Methods, Access Specifies, Static Members, Final Members, Comments, Data types, Variables, Operators, Control Flow, Arrays & String. Object Oriented Programming: Class, Object, Inheritance Super Class, Sub Class, Overriding, Overloading, Encapsulation, Polymorphism, Abstraction, Interfaces, and Abstract Class. Packages: Defining Package, CLASSPATH Setting for Packages, Making JAR Files for Library Packages, Import and Static Import Naming Convention For Packages
II	Exception Handling: The Idea behind Exception, Exceptions & Errors, Types of Exception, Control Flow in Exceptions, JVM Reaction to Exceptions, Use of try, catch, finally, throw, throws in Exception Handling, In-built and User Defined Exceptions, Checked and Un-Checked Exceptions. Input /Output Basics: Byte Streams and Character Streams, Reading and Writing File in Java. Multithreading: Thread, Thread Life Cycle, Creating Threads, Thread Priorities, Synchronizing Threads, Inter-thread Communication.
III	Java New Features: Functional Interfaces, Lambda Expression, Method References, Stream API, Default Methods, Static Method, Base64 Encode and Decode, For Each Method, Try-with resources, Type Annotations, Repeating Annotations, Java Module System, Diamond Syntax with 08 Inner Anonymous Class, Local Variable Type Inference, Switch Expressions, Yield Keyword, TextBlocks, Records, Sealed Classes

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Syllabus

Unit	TOPIC
IV	Java Collections Framework: Collection in Java, Collection Framework in Java, Hierarchy of Collection Framework, Iterator Interface, Collection Interface, List Interface, ArrayList, LinkedList, Vector, Stack, Queue Interface, Set Interface, HashSet, LinkedHashSet, SortedSet Interface, TreeSet, Map Interface, HashMap Class, LinkedHashMap Class, TreeMap Class, Hashtable Class, Sorting, Comparable Interface, Comparator Interface, Properties Class in Java.
V	Spring Framework: Spring Core Basics-Spring Dependency Injection concepts, Spring Inversion of Control, AOP, Bean Scopes- Singleton, Prototype, Request, Session, Application, Web Socket, Auto wiring, Annotations, Life Cycle Call backs, Bean Configuration styles Spring Boot: Spring Boot Build Systems, Spring Boot Code Structure, Spring Boot Runners, Logger, BUILDING RESTFUL WEB SERVICES, Rest Controller, Request Mapping, Request Body, Path Variable, Request Parameter, GET, POST, PUT, DELETE APIs, Build Web Applications

GL BAJAJ Institute of Technology & Management
FIND YOUR SPARK

Course Outcome

At the end of the Course, the student will be able

Course Outcomes (CO)		Bloom's Knowledge Level (KL)
CO1	Develop the object-oriented programming concepts using Java	K3, K4
CO2	Implement exception handling, file handling, and multi-threading in Java	K2, K4
CO3	Apply new java features to build java programs	K3
CO4	Analyze java programs with java Collection Framework	K4
CO5	Test web and RESTful Web Services with Spring Boot using Spring Framework concepts	K5

CO-PO Mapping

CO-PO Correlation Matrices

Correlation levels are taken 1, 2 and 3 as defined below:

1: Slight (Low) **2:** Moderate (Medium) **3:** Substantial (High)

CO/PO	Programme Outcomes (PO)												PSO's	
	1	2	3	4	5	6	7	8	9	10	11	12	1	2
CO 1	2	2	2	2	2	-	-	-	-	-	-	2	2	-
CO 2	2	2	2	2	2	-	-	-	-	-	-	2	2	-
CO 3	2	3	2	2	2	-	-	-	-	-	-	2	2	-
CO 4	2	3	2	2	3	-	-	-	-	-	2	2	2	-
CO 5	1	3	2	3	3	-	-	-	-	-	2	3	3	-
PO Target	**1.8**	**2.6**	**2**	**2.2**	**2.4**	**-**	**-**	**-**	**-**	**-**	**2**	**2.2**	**2.17**	**-**

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

CO-PO and PSO Mapping

Program Specific Outcomes and Course Outcomes Mapping

CO	PSO1	PSO2	PSO3	PSO4
CO1	3	3	-	3
CO2	3	3	2	3
CO3	3	3	-	3
CO4	3	3	-	3
CO5	3	3	-	3

*3= High *2= Medium *1=Low

Prerequisite and Recap

- Basic knowledge of C/C++
- Innovative Thinking.
- Enthusiasm to learn Programming concepts.

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Evaluation Scheme

Subject	L	T	P	CT	TA	TOTAL	PS	TE	PE	TOTAL	CREDIT
	2	1	4	20	10	30		70		100	3

PO's

- Engineering Graduates will be able to:

PO1. Engineering knowledge: Apply the knowledge of mathematics ,science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems

PO2. 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.

PO3. Design/development of solutions: 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

PO4. Conduct investigations 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

PO5. 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's

PO6. The engineer and 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.

PO7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. 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's

PO11. Project management and 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.

PO12. 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.

Brief Introduction about the Subject with Videos

OOP stands for Object-Oriented Programming. Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods. The main ideas behind Java's Object-Oriented Programming, OOP concepts include abstraction, encapsulation, inheritance and polymorphism. Basically, Java OOP concepts let us create working methods and variables, then re-use all or part of them without compromising security.

Why JAVA/History of JAVA

Use Of Java

1. **Android Apps :** Java has a rich use in Android Applications. Open your Android phone and any app, they are actually written in Java programming language, with Google's Android API, which is similar to JDK (DVM in android). Couple of years back Android has provided much needed boost and today many Java programmer are Android App developer.

2. **Financial Services Industries :** Java is very big in Financial Services which demands more security. API's of Data Processing & Payment Gateways are created in java. It is mostly used to write server side application, mostly without any front end, which receives data form one server (upstream), process it and sends it other process (downstream).

Use Of Java

3. **Java Web Applications :** Java is also big on E-commerce and web application space. Java frameworks i.e. Spring MVC, Struts 2.0 and other frameworks are used for this web applications. Many of government, healthcare, insurance, education, defense and several other department have their web application built in Java.

4. **Software Tools :** Many useful software and development tools are written and developed in Java e.g. Netbans, Eclipse and InetelliJ IDE. They are also most used desktop applications for development of java and other languages.

Use Of Java

5. **Big Data Technologies :** Many social networking websites like facebook twitter etc and e-commerse websites generates a lot of data day to day. So to manage this huge amount of data, Hadoop and other big data technologies are developed using java only. So Java plays an important role for big data technologies.

6. **Scientific Applications :** As Java is more safe, portable, maintainable and comes with better high-level concurrency tools than C++ or any other language therefore Java is often a default choice for scientific applications, including natural language processing.

Use Of Java

7. J2ME Apps : There was time when Nokia and Samsung handsets had a large market which uses J2ME. At that time almost all games, applications, which is available in Android are written using MIDP and CLDC, part of J2ME platform.

8. Embedded Systems : Java is also used in embedded systems. Computers, Printers, Routers, ATM's, Home Security Systems etc. all uses java a lot.

Why Java is Secured

1. JVM : JVM is Java Virtual Machine. Its basic role is to verify the bytecode before it is run. This makes sure that the program isn't making any unsafe operations. There are various unsafe operations that the programs can generally perform; for example, a program might branch to incorrect locations that can contain data rather than instruction. The JVM ensures that such unsafe operations aren't being performed.

Over and above this task, the JVM also enforces runtime constraints. This can include array bounds checking and others like it reduces the chances of the developers suffering from memory safety flaws. Hence, they can avoid such flaws as buffer overflow or others; which makes java more secured language as compared to other languages

Why Java is Secured

2. Security Manager : Security manager layer is present in JVM. It makes sure that the untrusted code doesn't manage to access some APIs and features of the platform.

3. No Pointers : Java does not support pointers (C & C++ supports). Some of the arbitrary memory locations can be addressed with the help of pointers for doing read and write operations which are unauthorized. This does not serve the purpose of being secured. That is why users do not use the concept of pointers.

Why Java is Secured

4. Access Modifiers : There are four access modifiers in java i.e. public, protected, private and default. So as developer if we want to hide some code then we can use access modifiers.
If we want that our code or data should not be override, then we can use 'final' keyword; which again adds security to java.

5. Exception Handling : Exception handling catch the results which are unexpected through exception handling and report the error to the programmer. Until the rectification of the error by the programmer this concept will not allow us to run the code. Thereby proving Java's security.

Why Java is Secured

6. Own Memory Management : The memory management mechanism is unique and is owned by Java. There is no need for manual intervention for garbage collection and everything is handled automatically.
There is no need for a headache to free the memories.
It drastically reduces the programmer overhead.
Therefore the programmer's hand must be free from memory management. Relieving the memory in Java is the job of JVM.

JVM(Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

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

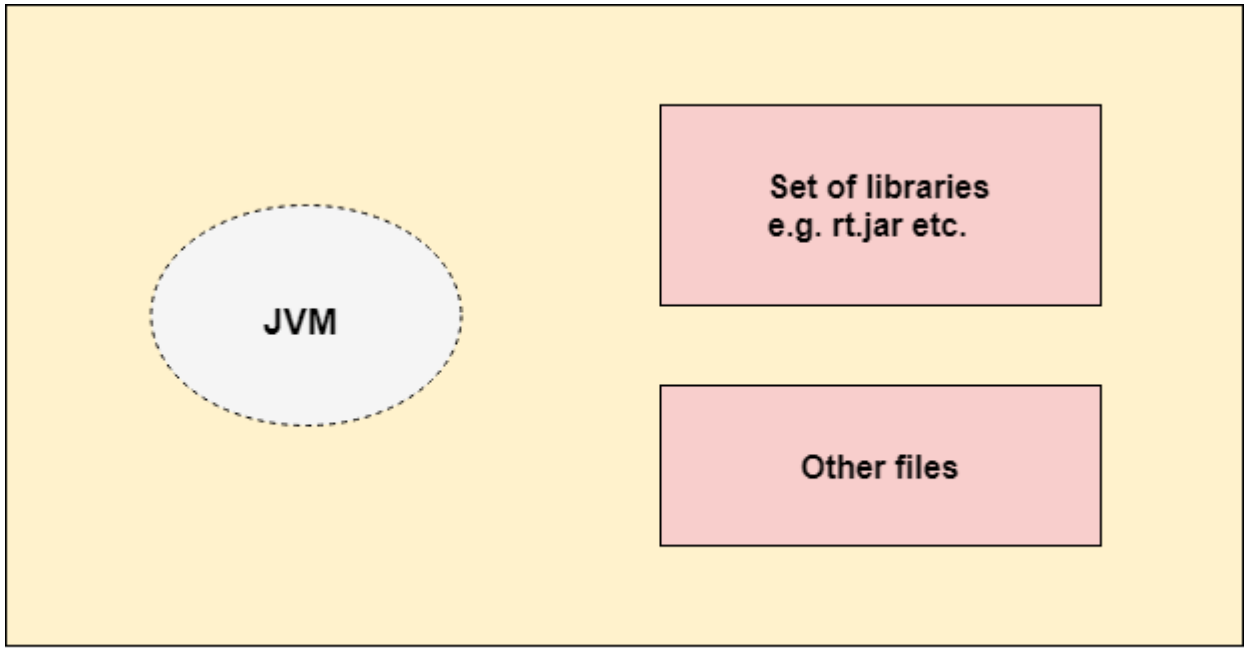
The JVM performs the following main tasks:

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

JAVA Runtime Environment(JRE)

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

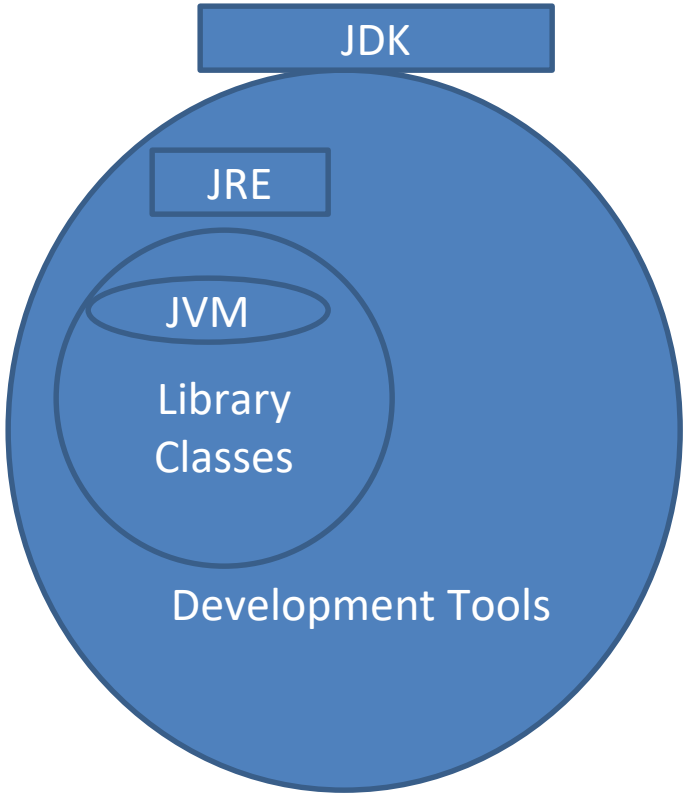
The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

Java Development Kit (JDK)

- TO Develop and Run JAVA applications JDK is used.
- JVM is responsible to Run Java program Line by Line
- JRE provides Environment just to Run Java program.



Java Source File Structure

Java source file structure describes that the Java source code file must follow a schema or structure. In this article, we will see some of the important guidelines that a Java program must follow.

A Java program has the following structure:

1. package statements: A package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces.

2. import statements: The import statement is used to import a package, class, or interface.

3. class definition: A class is a user-defined blueprint or prototype from which objects are created, and it is a passive entity.

Java Source File Structure

```
package mygfg;
import java.util.*;

class GFG{

int x;

}
```

Package

Import statement

Class Definition

Java Source File Structure

Working with Java Source File

1. Number of classes in a Java source file:

A Java program can contain any number of classes, and at most, one of them can be declared as a public class.

Explanation: Java allows us to create any number of classes in a program. But out of all the classes, at most, one of them can be declared as a public class. In simple words, the program can contain either zero public class, or if there is a public class present, it cannot be more than one.

2. Name of the Java source file:

The name of the Java source file can be anything provided that no class is declared as public.

Explanation: Java allows us to name the Java source file with anything if there is not a single class that is declared as public. But if there is a class that is declared as public, the name of the Java source file must be the same as the public class name. The Java source file extension must be .java.

3. Number of .class files in a Java source file:

The number of .class files generated is equal to the number of classes declared in the Java program.

Explanation: While compiling a program, the javac compiler generates as many .class files as there are classes declared in the Java source file.

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

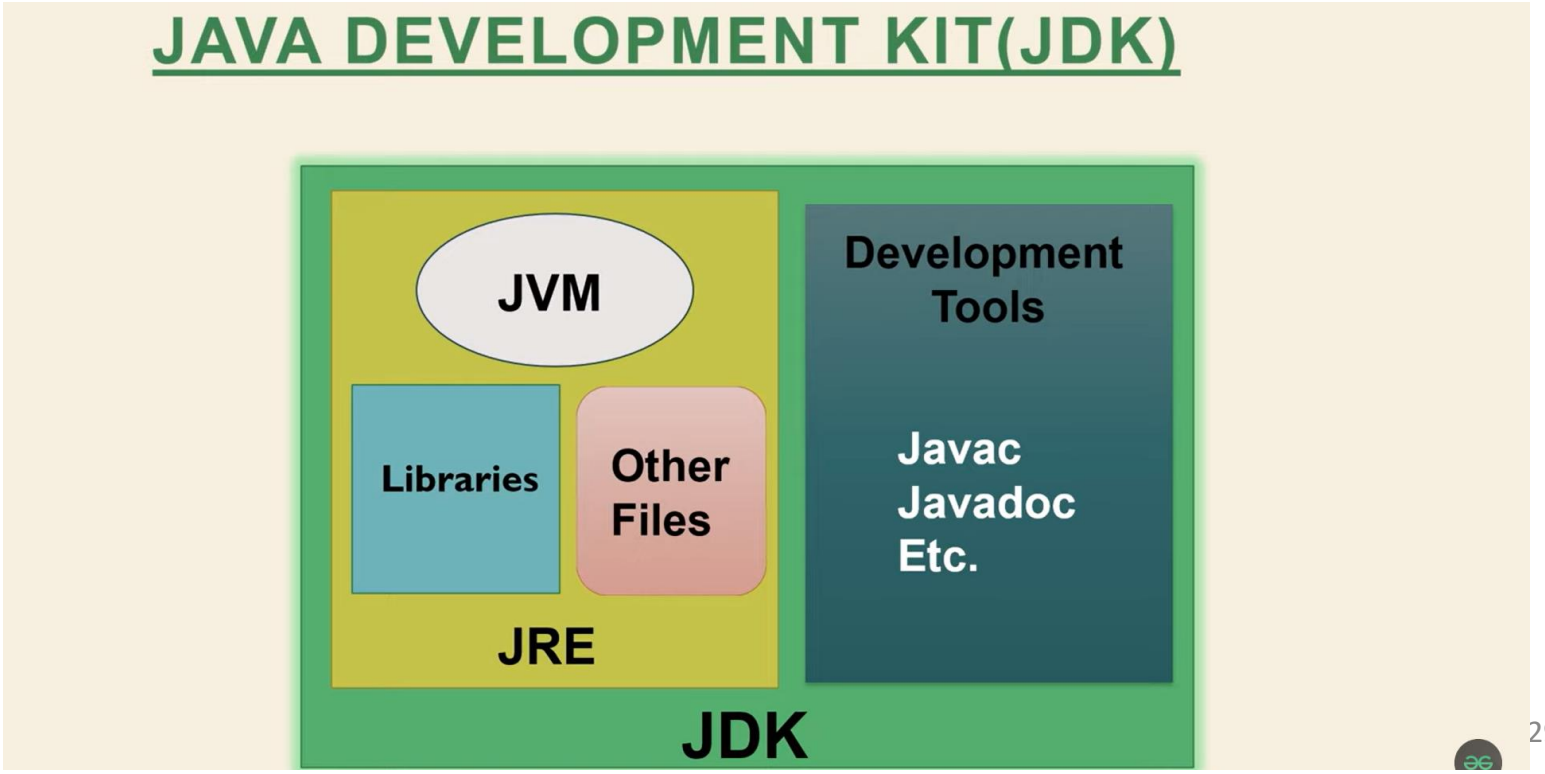
Compilation and Execution of a Java Program

Java, being a platform-independent programming language, doesn't work on the one-step compilation. Instead, it involves a two-step execution, first through an OS-independent compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system.

The two principal stages are explained below:

Principle 1: Compilation

Principle 2: Execution



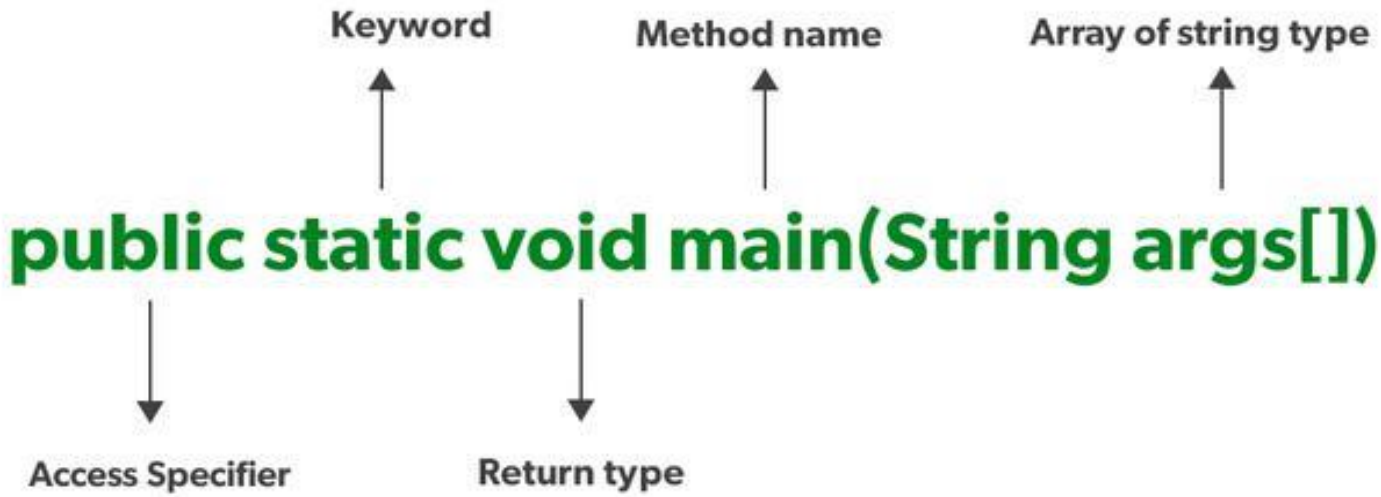
Java Fundamentals

Java main() Method – public static void main(String[] args)

Java's main() method is the starting point from where the JVM starts the execution of a Java program. JVM will not execute the code, if the program is missing the main method. Hence, it is one of the most important methods of Java, and having a proper understanding of it is very important.
The Java compiler or JVM looks for the main method when it starts executing a Java program. The signature of the main method needs to be in a specific way for the JVM to recognize that method as its entry point. If we change the signature of the method, the program compiles but does not execute.
The execution of the Java program, the **java.exe** is called. The Java.exe in turn makes Java Native Interface or JNI calls, and they load the JVM. The java.exe parses the command line, generates a new String array, and invokes the main() method. By default, the main thread is always a non-daemon thread.

Java Fundamentals

Java main() Method – public static void main(String[] args)



Java Fundamentals

Java main() Method – public static void main(String[] args)

1. Public

It is an **Access modifier**, which specifies from where and who can access the method. Making the main() method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

2. Static

It is a **keyword** that is when associated with a method, making it a **class-related method**. The *main()* method is static so that JVM can invoke it without **instantiating the class**. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the *main()* method by the JVM.

Java main() Method – public static void main(String[] args)

3. Void

It is a **keyword** and is used to **specify that a method doesn't return anything**. As the *main()* method doesn't return anything, its return type is **void**. As soon as the *main()* method terminates, the Java program terminates too. Hence, it doesn't make any sense to return from the *main()* method as JVM can't do anything with its return value of it.

4. main

It is the **name of the Java main method**. It is the **identifier** that the JVM looks for as the **starting point of the Java program**. It's not a keyword.

If we change the name while initiating main method, we will get an error.

Java main() Method – public static void main(String[] args)

5. String[] args

It **stores Java command-line arguments** and is an array of type ***java.lang.String*** class. Here, the name of the String array is *args* but it is not fixed and the user can use any name in place of it.

Java Fundamentals

System.out.println in Java

Java **System.out.println()** is used to print an argument that is passed to it.

Parts of System.out.println()

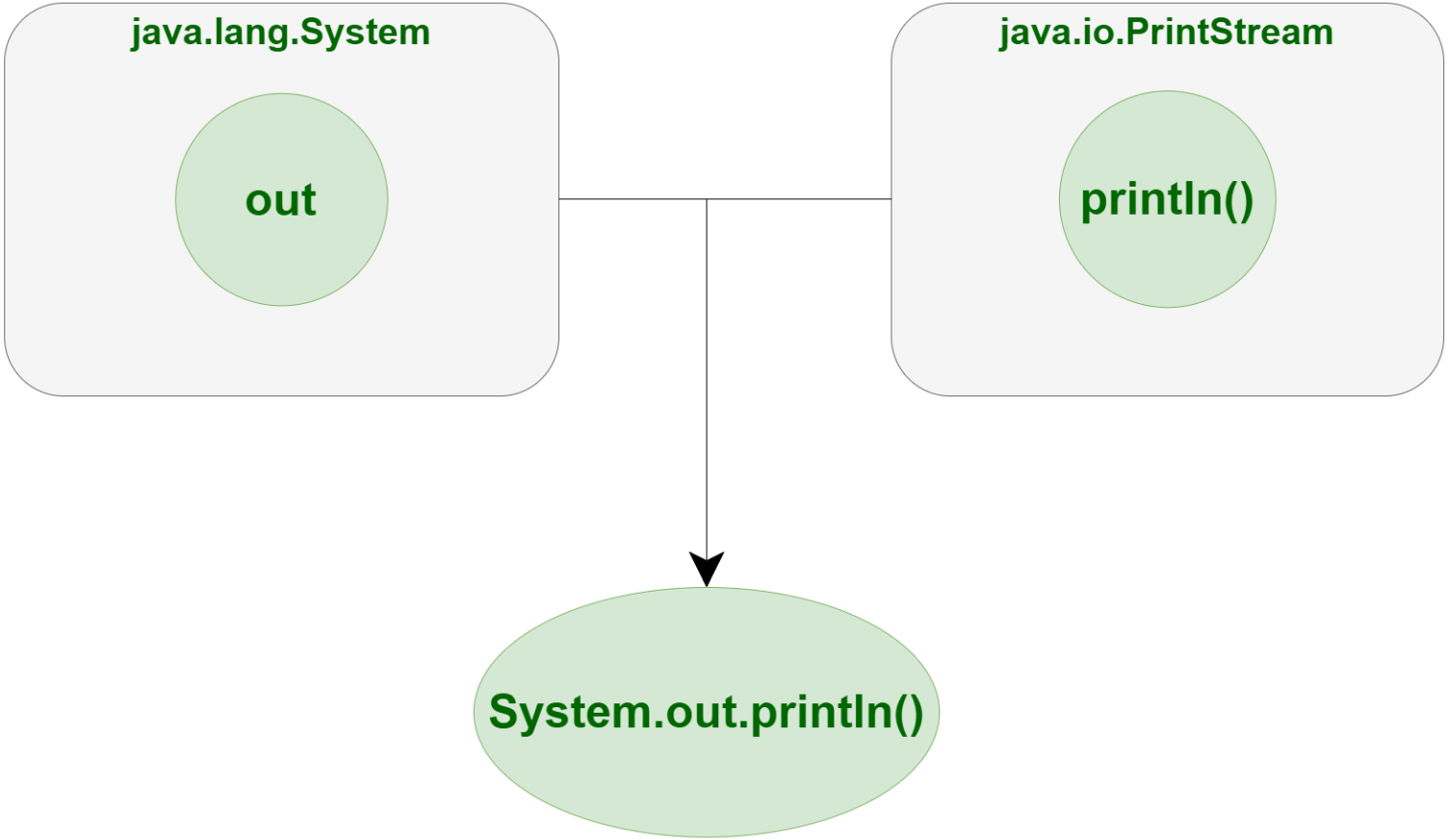
The statement can be broken into 3 parts which can be understood separately:

1.System: It is a final class defined in the java.lang package.

2.out: This is an instance of PrintStream type, which is a public and static member field of the System class.

3.println(): As all instances of the PrintStream class have a public method println(), we can invoke the same on out as well. This is an upgraded version of print(). It prints any argument passed to it and adds a new line to the output. We can assume that System.out represents the Standard Output Stream.

Java Fundamentals



Working of System.out.println()

History of Java

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

History of Java

History of Java

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s. Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

History of Java

History of Java

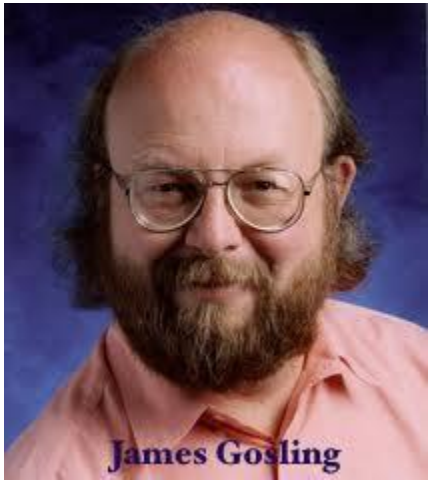
1) James Gosling, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.



History of Java

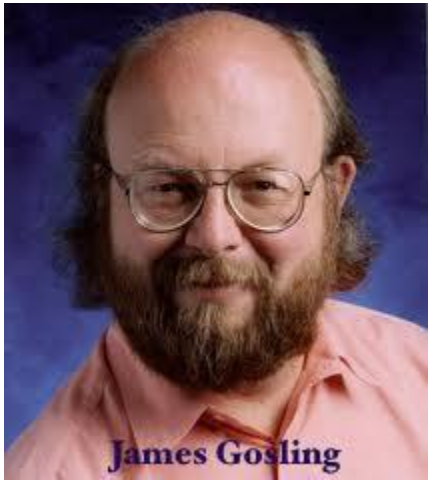
History of Java

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

7) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

8) In 1995, Time magazine called Java one of the Ten Best Products of 1995.

9) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.



History of Java

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

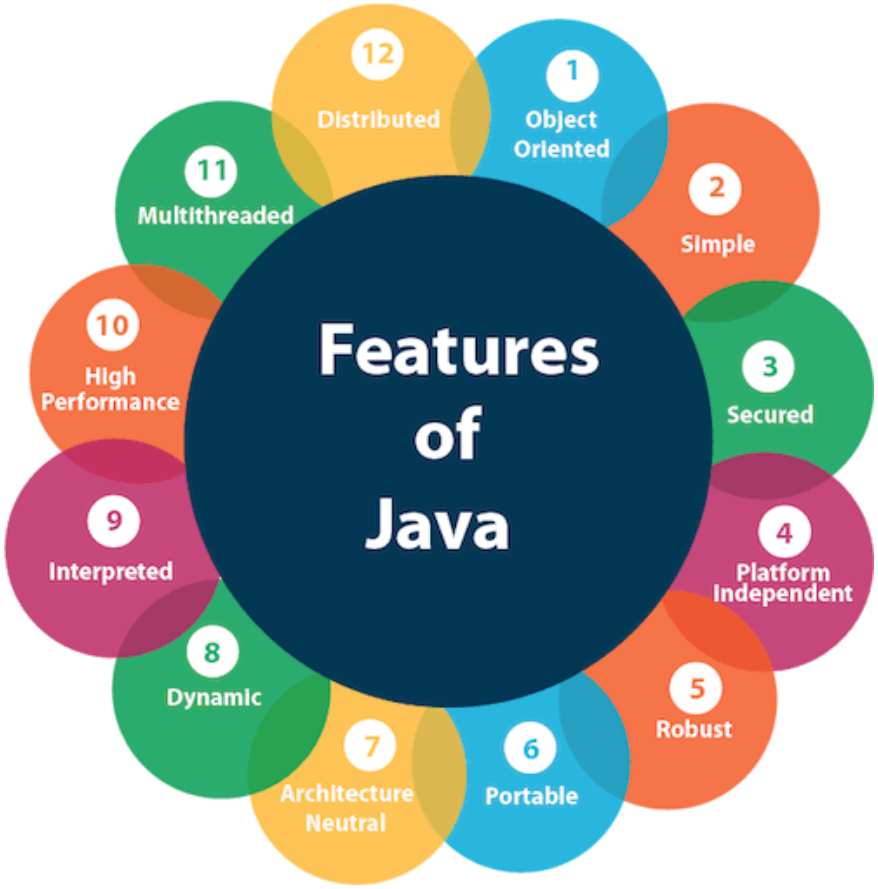
1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)

History of Java

Java Version History

11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (to be released by March 2022)

Features of Java



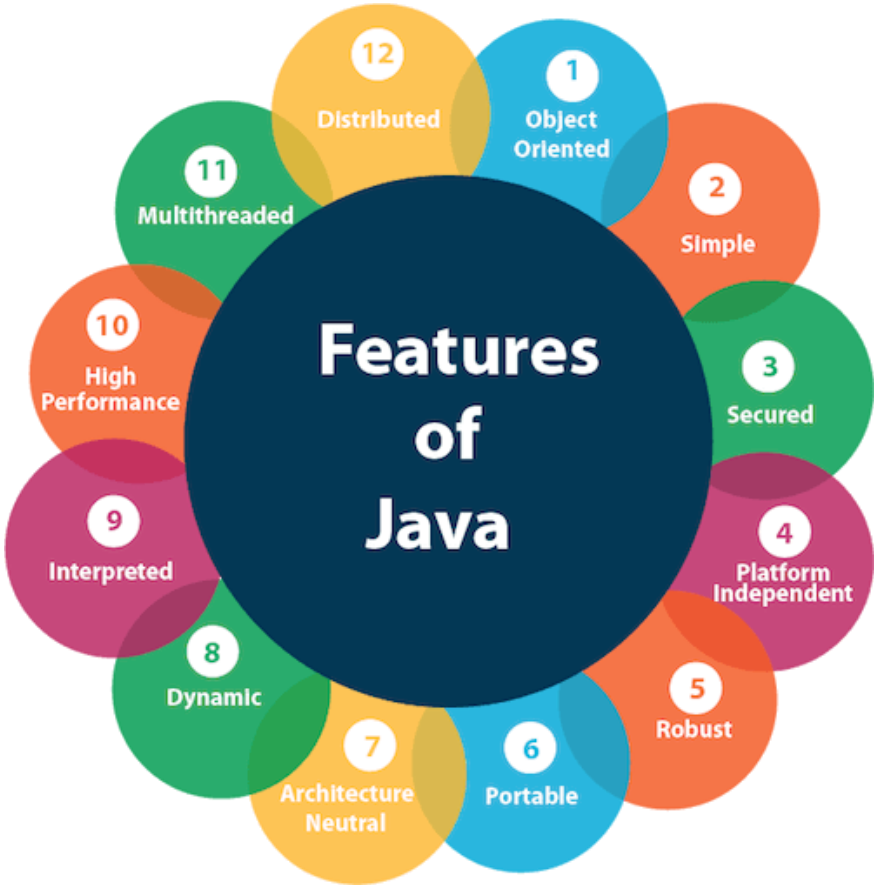
Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

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

Basic concepts of OOPs are:

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

Features of Java



Simple

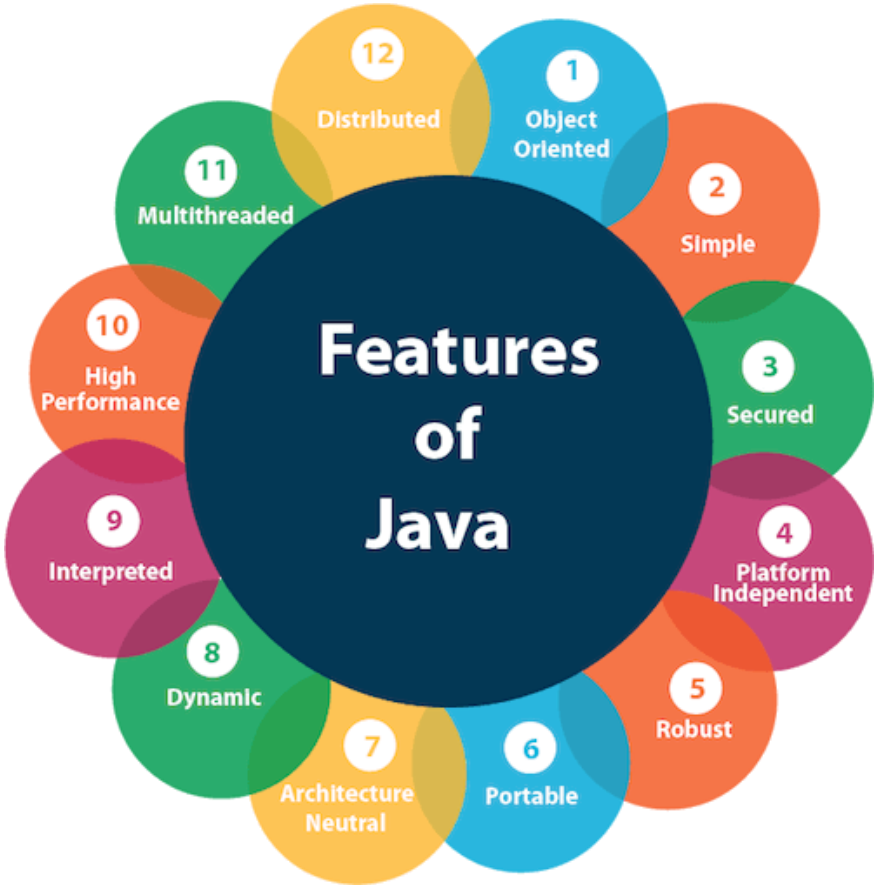
Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

•Java syntax is based on C++ (so easier for programmers to learn it after C++).

•Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

•There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Features of Java



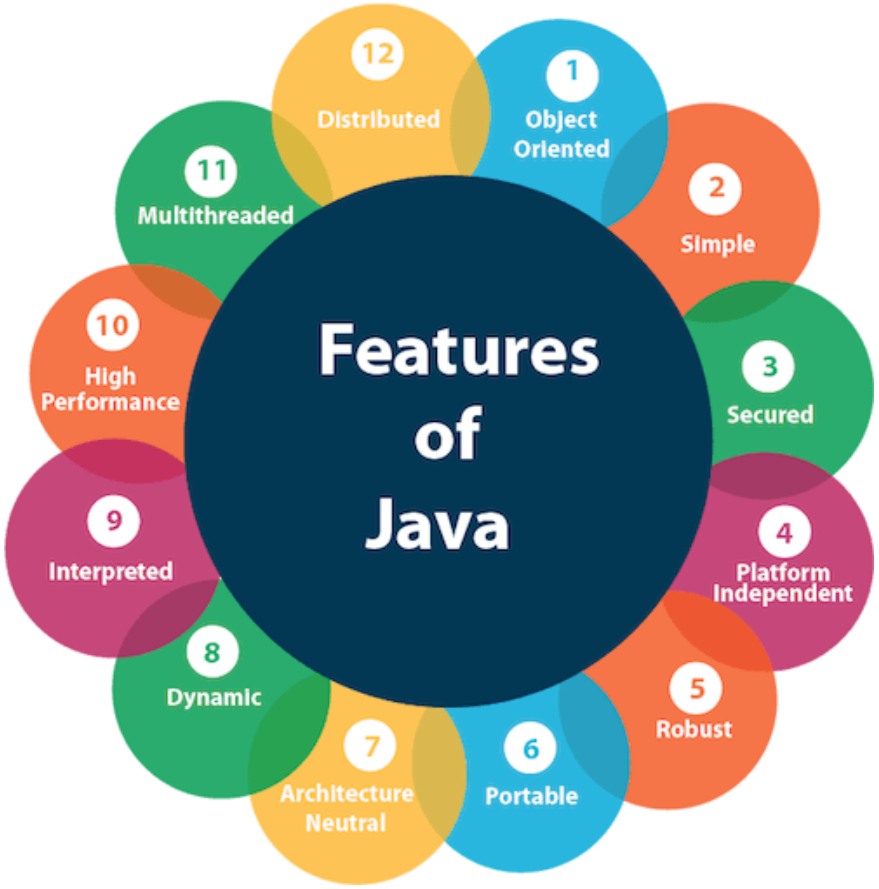
Secure

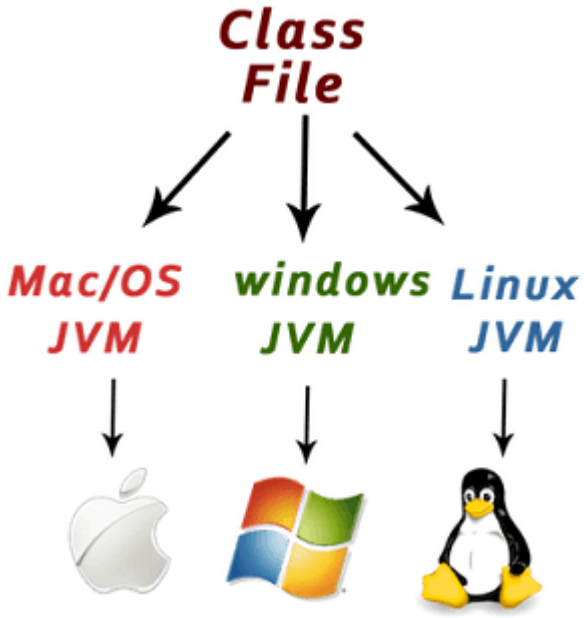
Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**

Features of Java

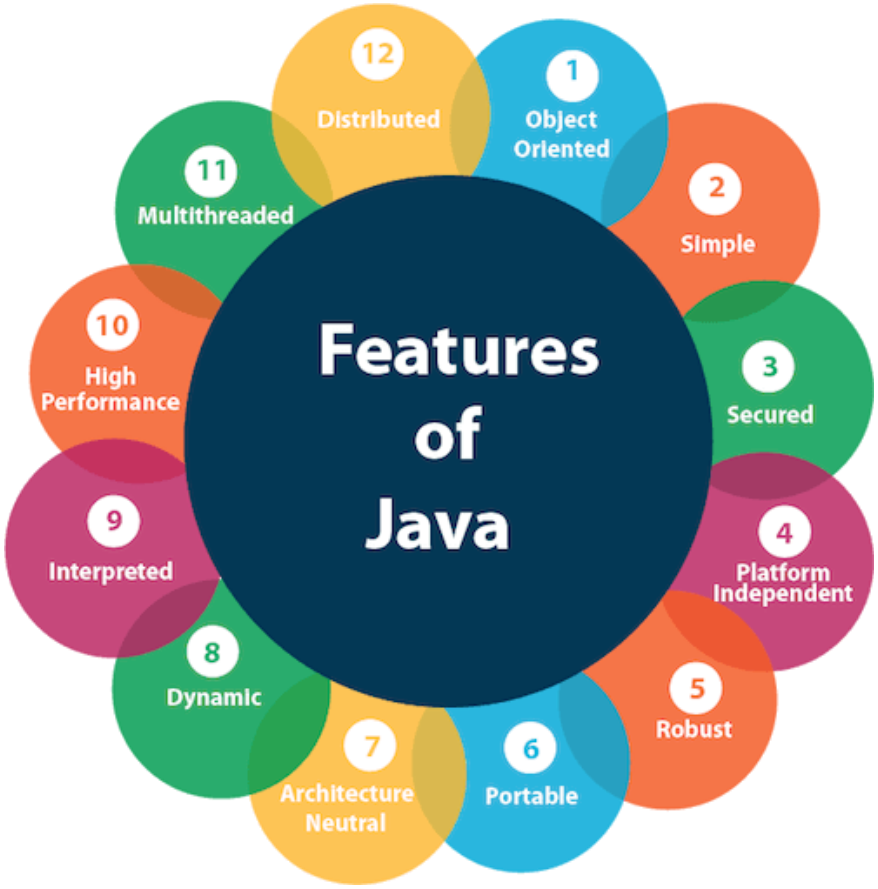
Platform Independent





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

Features of Java

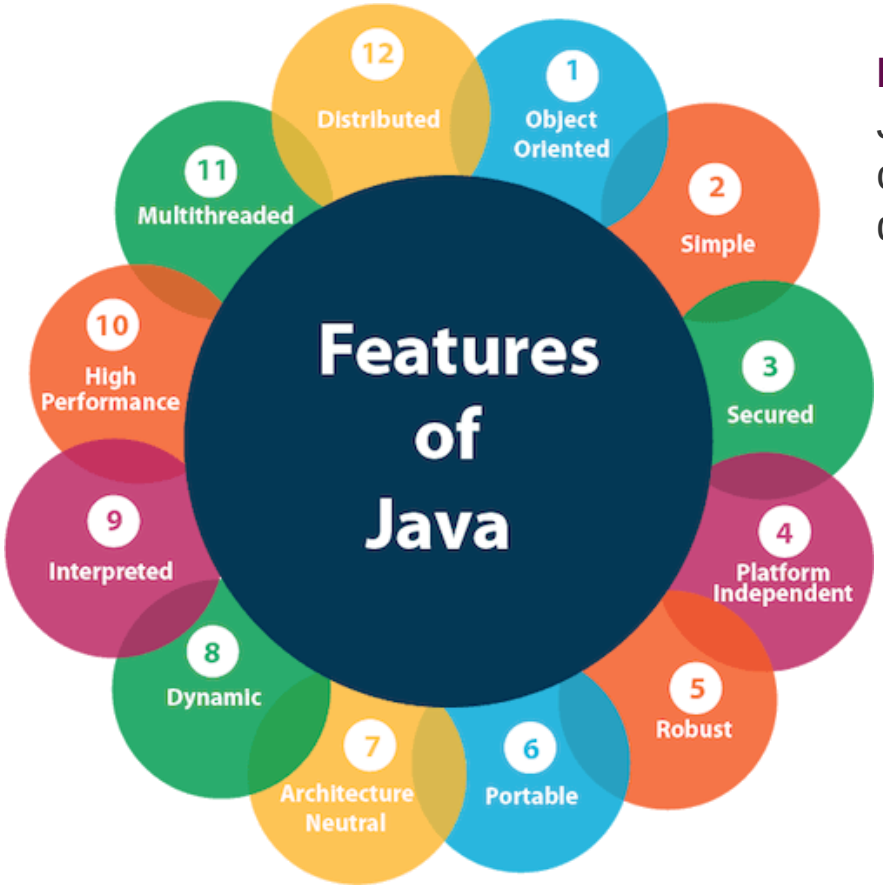


Robust

The English mining of Robust is strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

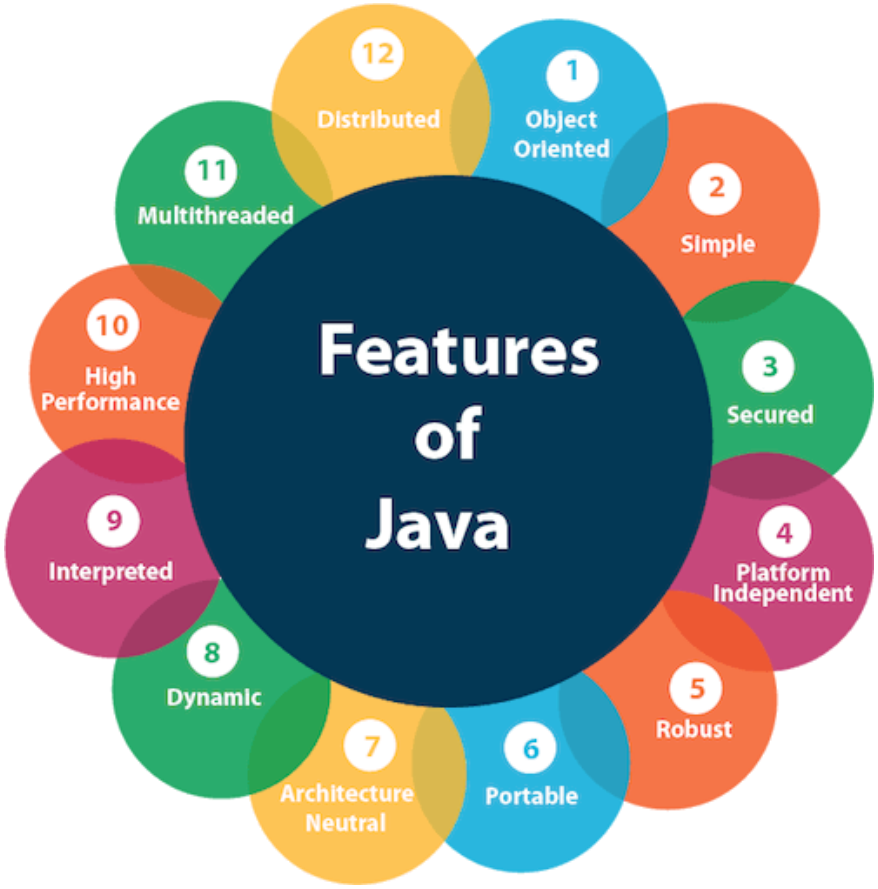
Features of Java



Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

Features of Java

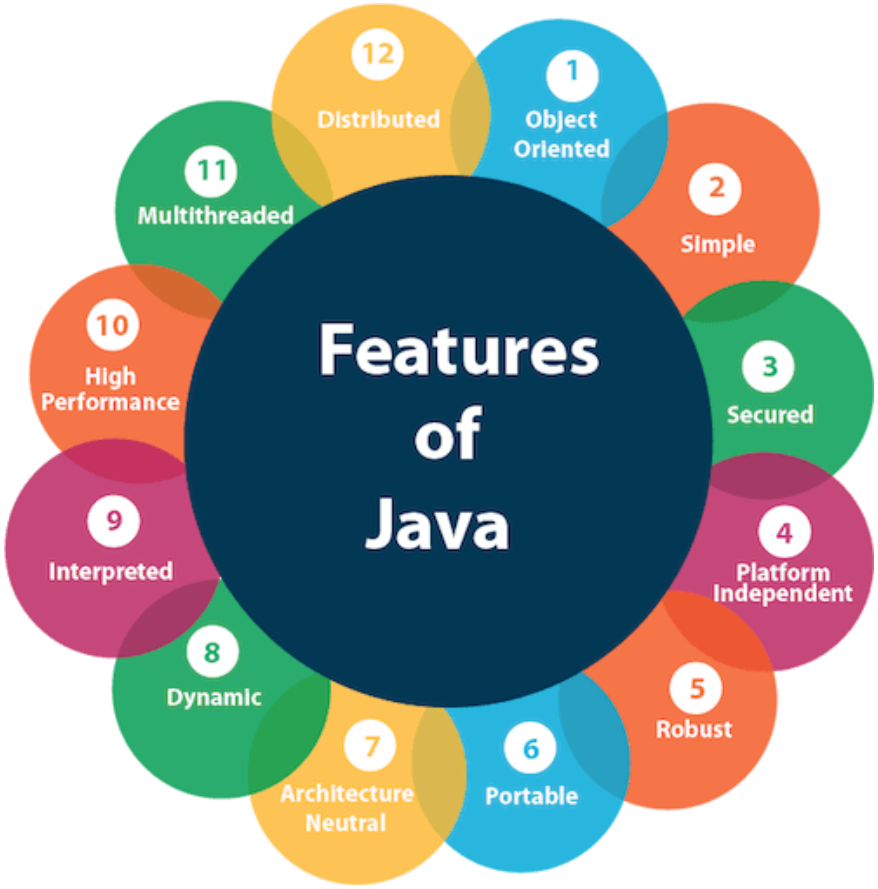


Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

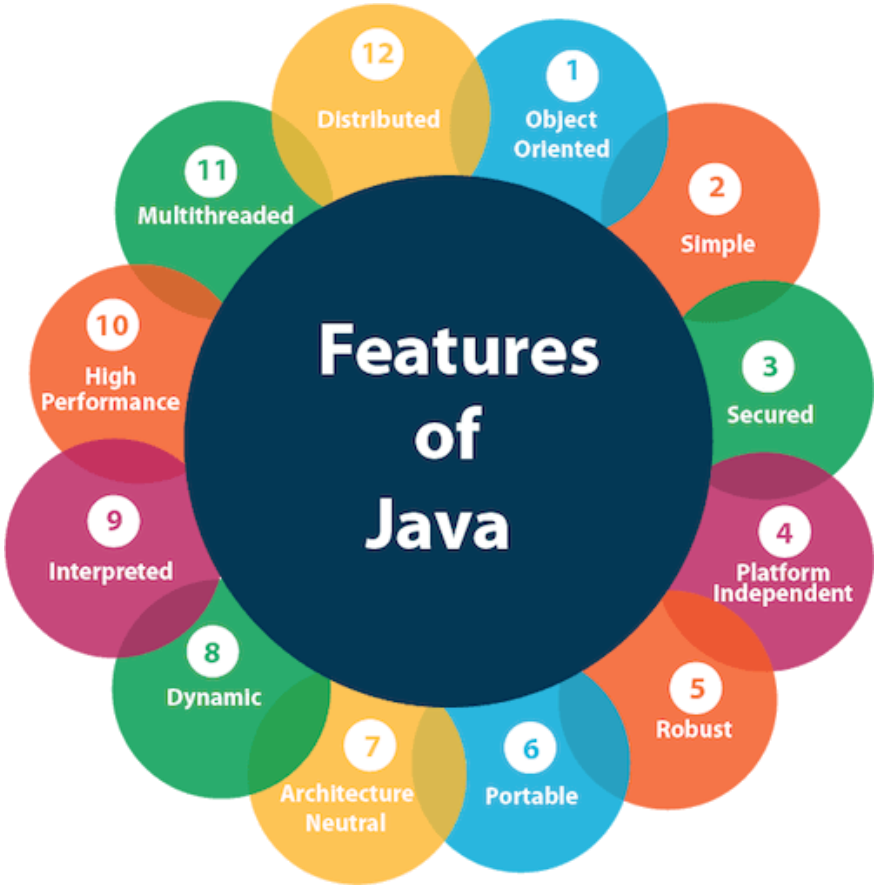
Features of Java



Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

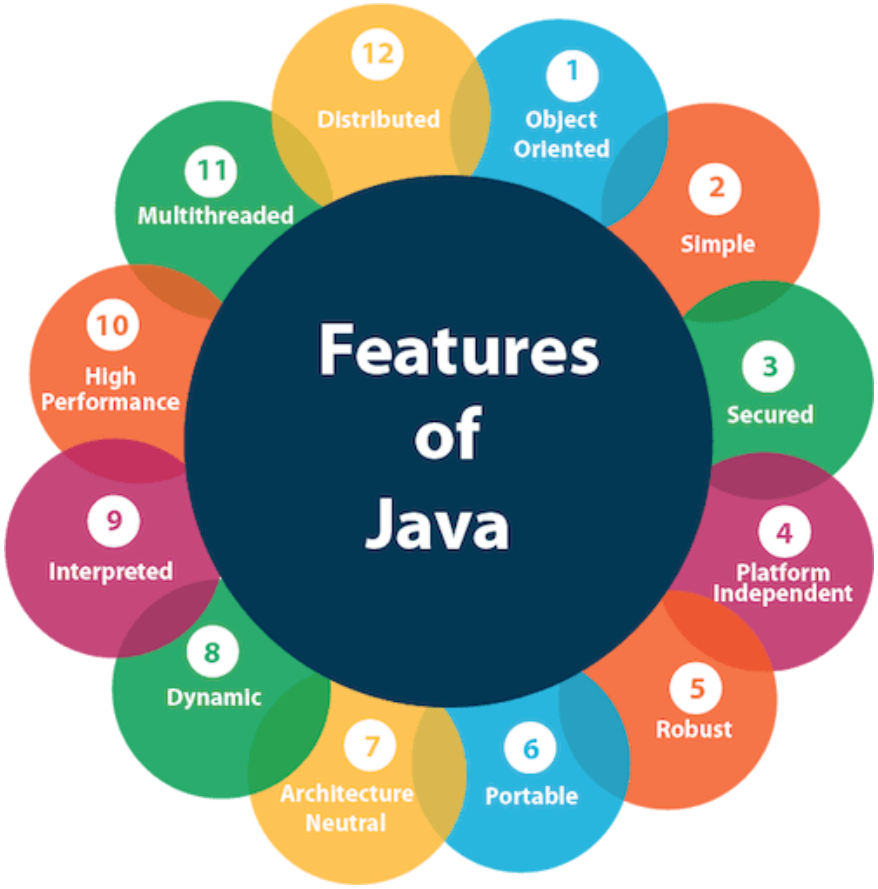
Features of Java



Interpreted

Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.

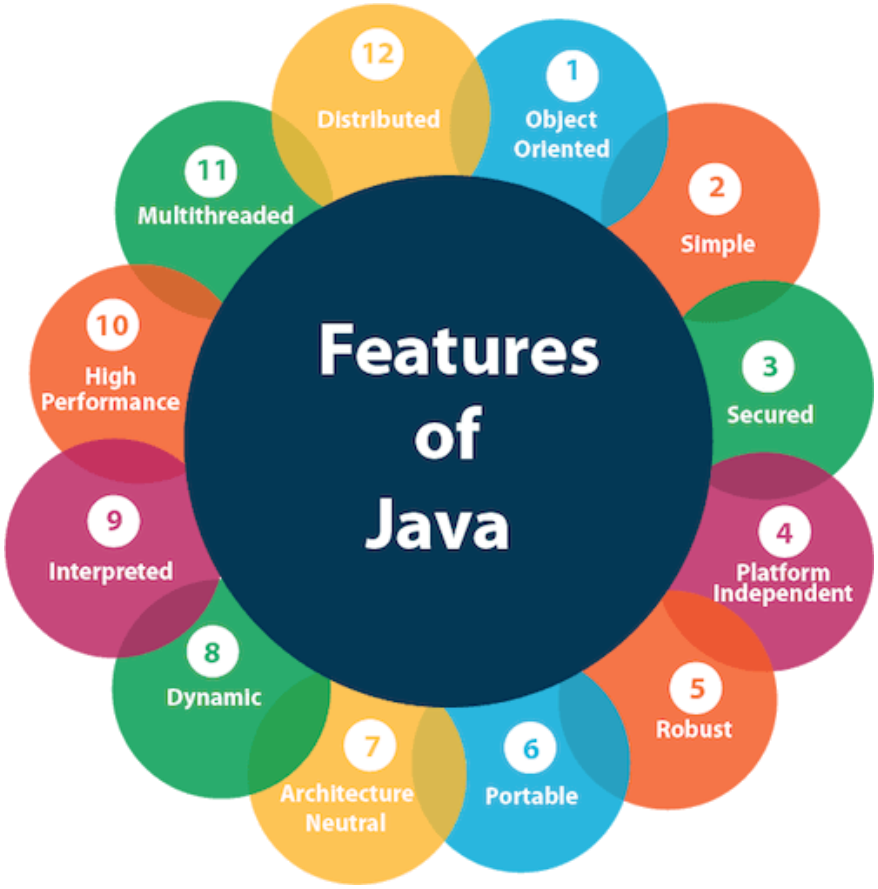
Features of Java



High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

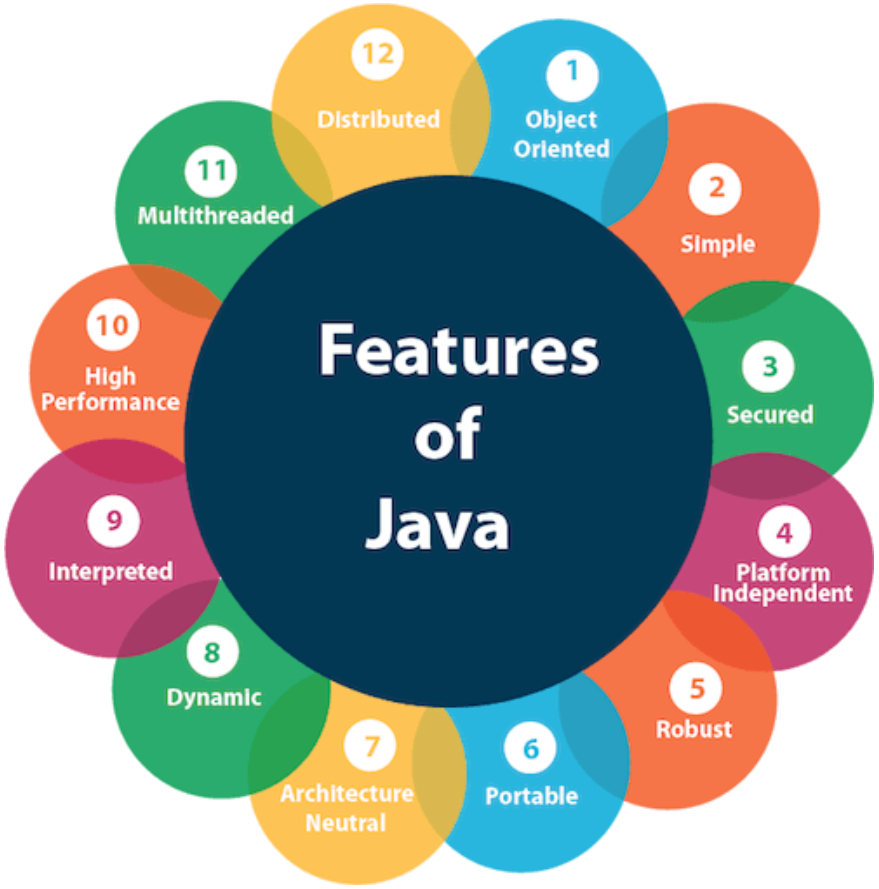
Features of Java



Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

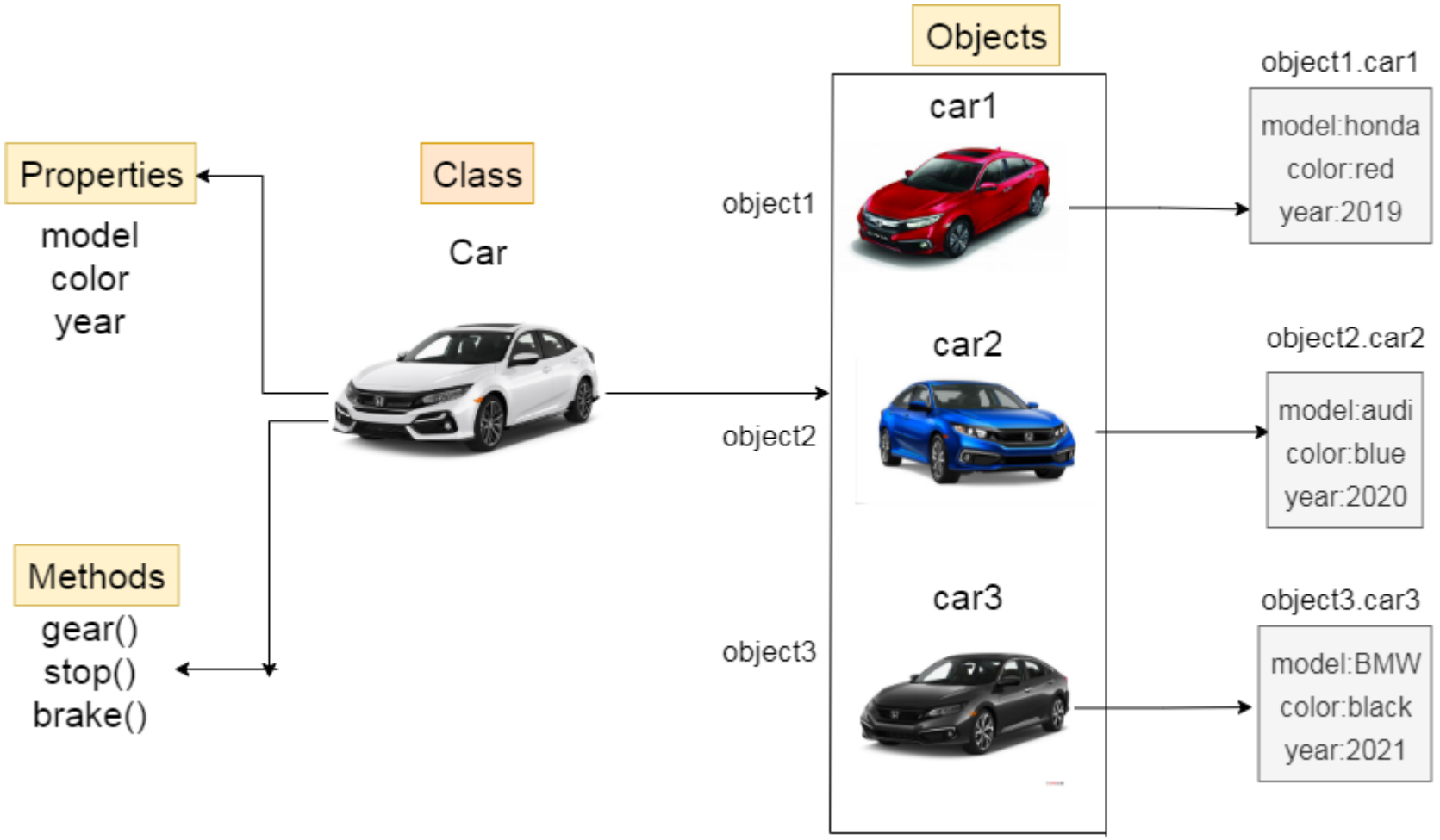
Features of Java



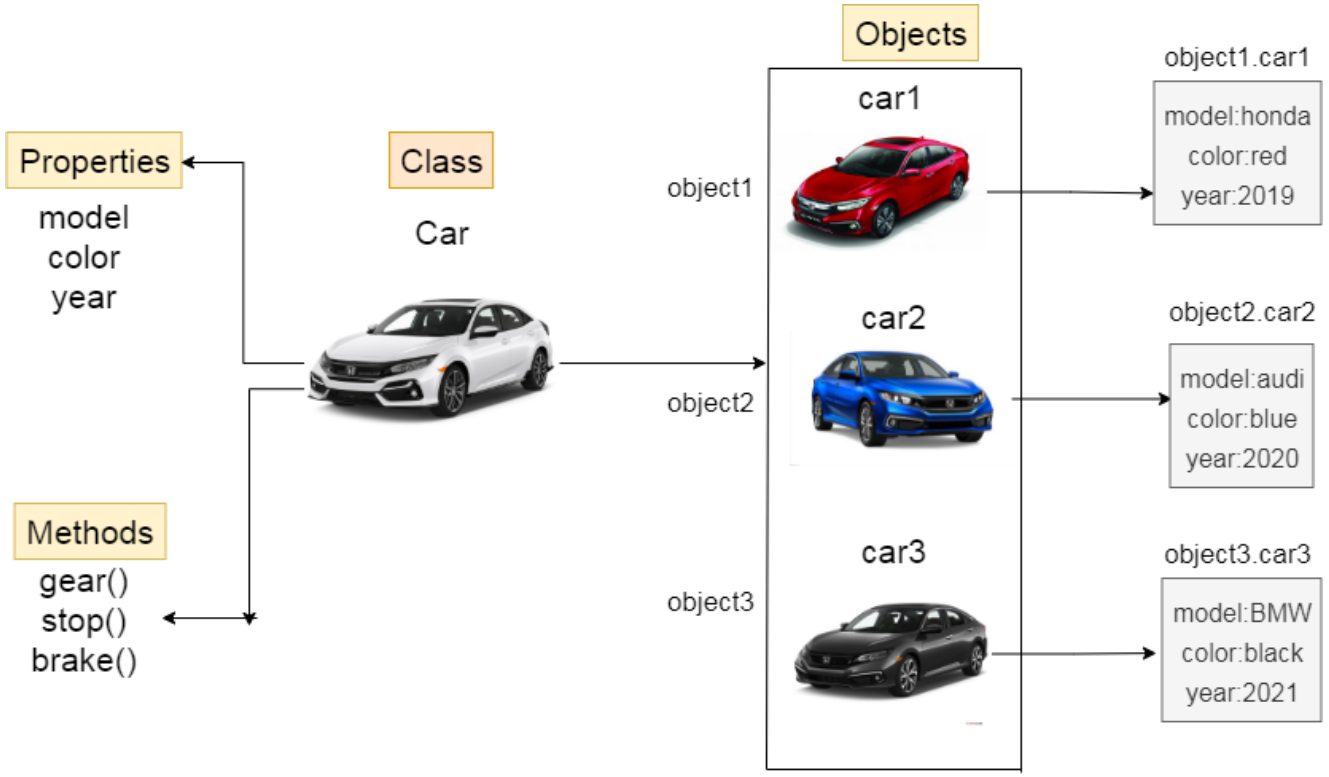
Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI (Remote Method Invocation) and EJB (Enterprise Server Bean) are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Defining Classes in Java



Defining Classes in Java



A class is a group of objects which have common properties. It is a user-defined data type with a template that serves to define its properties. To create a class, use the keyword class.

A Class is like an object constructor, or a "blueprint" for creating objects.

Defining Classes in Java

Create a Class

To create a class, use the keyword `class`:

class *MyClass*
{ // field, constructor, and
// method declarations }

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class:

- constructors for initializing new objects,
- fields provide the state of the class and its objects, and
- methods to implement the behavior of the class and its objects.

Defining Classes in Java

Create a Class

To create a class, use the keyword `class`:

class *MyClass*
{ // field, constructor, and
// method declarations }

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class:

- constructors for initializing new objects,
- fields provide the state of the class and its objects, and
- methods to implement the behavior of the class and its objects.

Defining Classes in Java

In general, class declarations can include these components, in order:

1.Modifiers such as *public*, *private*, and a number of others that you will encounter later. (However, note that the *private* modifier can only be applied to Nested Classes.)

2.The class name, with the initial letter **capitalized** by convention.

3.The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.

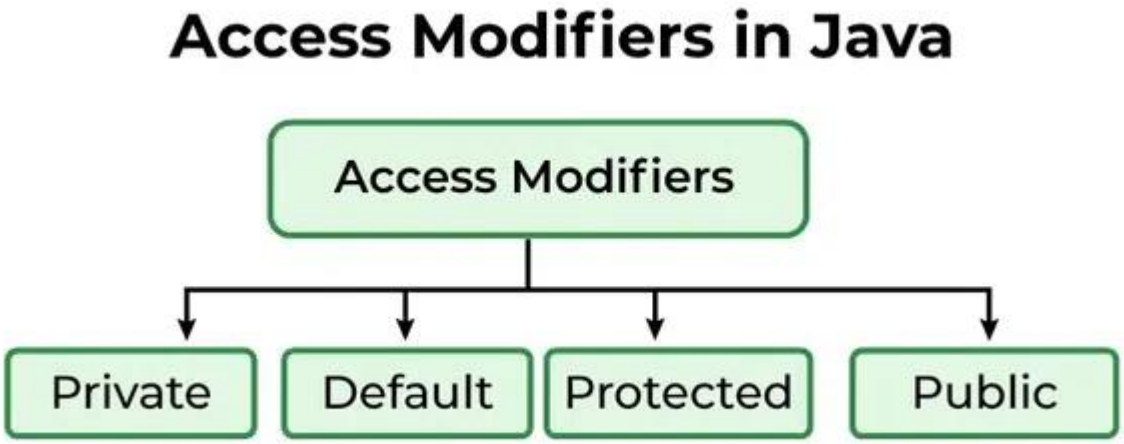
4.A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.

5.The class body, surrounded by braces, {}.

Declaring Member Variables in Java

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.



When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

Declaring Member Variables in Java

Field declarations are composed of three components, in order:

1.Zero or more modifiers, such as public or private.
2.The field's type.
3.The field's name.

The Bicycle class uses the following lines of code to define its fields:

```
public int model;
public int gear;
public int speed;
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Methods in Java

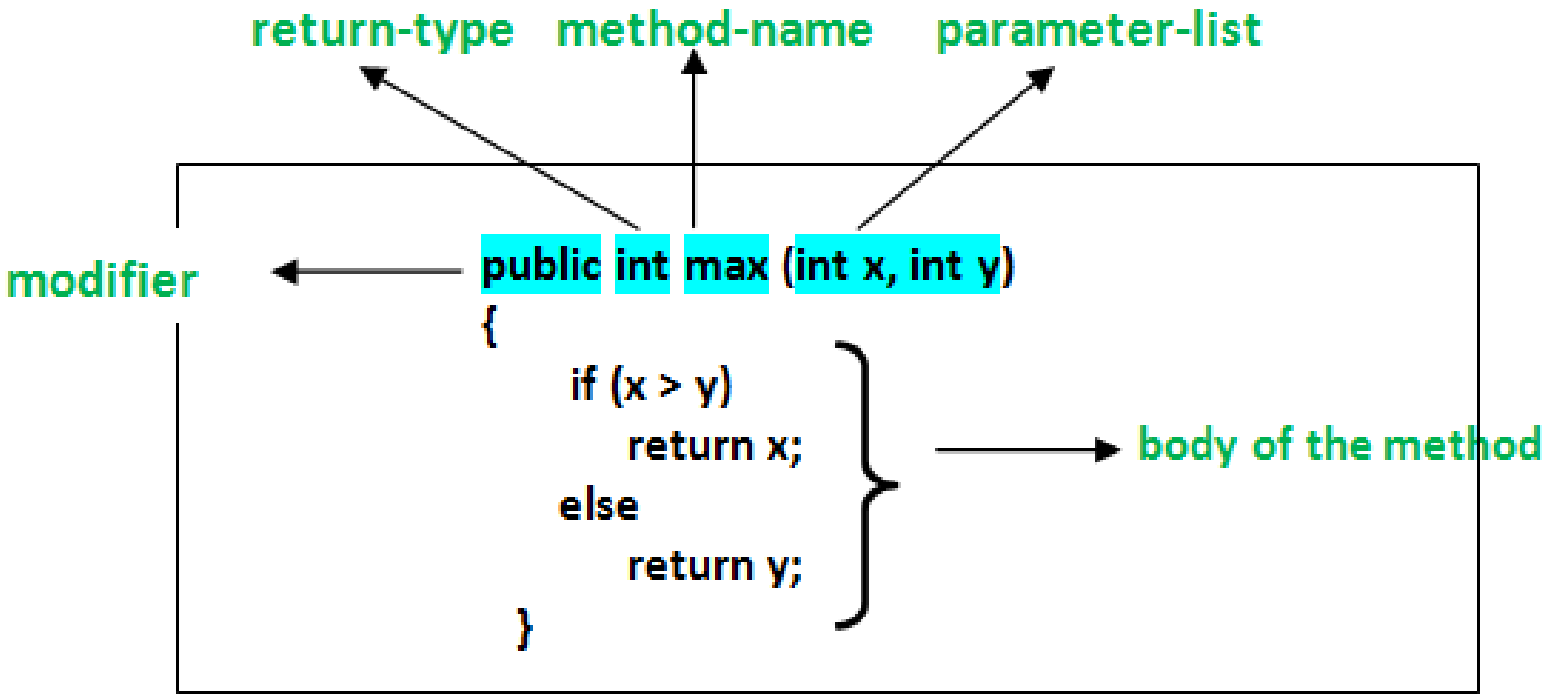
The **method in Java** or Methods of Java is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything. Java Methods allow us to **reuse** the code without retyping the code. In Java, every method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

1. A method is like a function i.e. used to expose the behavior of an object
2. It is a set of codes that perform a particular task.

Syntax of Method

```
<access_modifier> <return_type> <method_name>(
list_of_parameters)
{
    //body
}
```

Methods in Java



Methods in Java

More generally, method declarations have six components, in order:

1.Modifiers—such as public, private, and others you will learn about later.

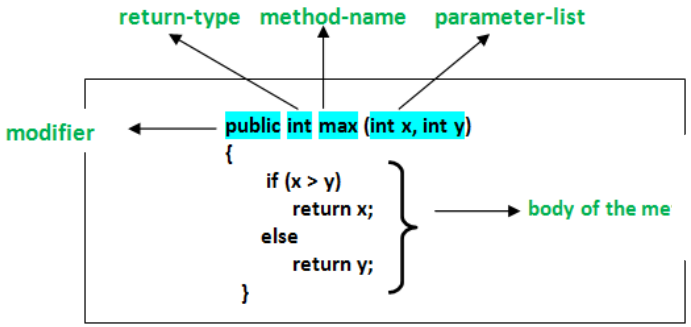
2.The return type—the data type of the value returned by the method, or void if the method does not return a value.

3.The method name—the rules for field names apply to method names as well, but the convention is a little different.

4.The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.

5.An exception list—to be discussed later.

6.The **method body**, enclosed between braces {} —the method's code, including the declaration of local variables, goes here.



Methods in Java

1. Modifier: It defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.

- **public:** It is accessible in all classes in your application.
- **protected:** It is accessible within the class in which it is defined and in its subclasses
- **private:** It is accessible only within the class in which it is defined.
- **default:** It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

Methods in Java

2. The return type: The data type of the value returned by the method or void if does not return a value. It is **Mandatory** in syntax.

3. Method Name: the rules for field names apply to method names as well, but the convention is a little different. It is **Mandatory** in syntax.

4. Parameter list: Comma-separated list of the input parameters is defined, preceded by their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses (). It is **Optional** in syntax.

5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s). It is **Optional** in syntax.

6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations. It is **Optional** in syntax.

Methods in Java

Access modifiers of class

Modifier	Class	Package	Subclass	Other Classes
Private	Yes	No	No	No
No modifier	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Methods in Java

Types of Methods in Java

There are two types of methods in Java:

1. 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.

2. 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.

Class in Java

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

a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.

To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box();        // create a Box object called mybox
```

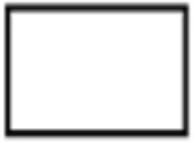
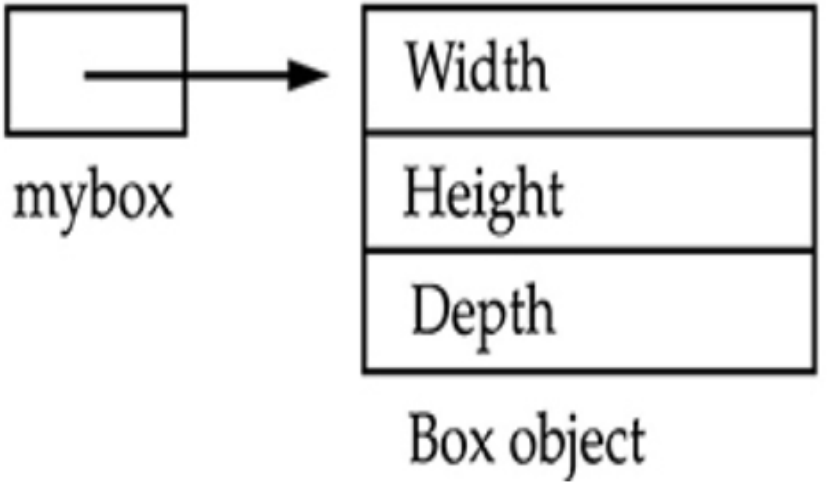
Class in Java

Obtaining objects of a class is a two-step process.

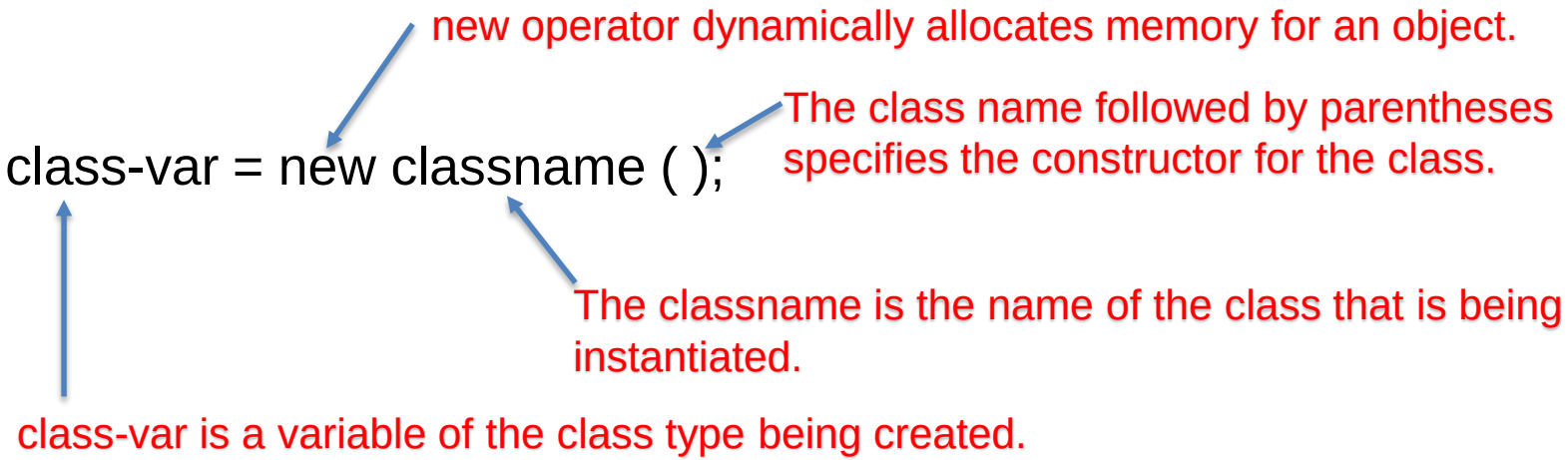
- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator.

The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.

Class in Java

Statement	Effect
Box mybox;	
mybox as a reference to an object of type Box	mybox
mybox = new Box();	
allocates an object and assigns a reference to it to mybox.	mybox Width Height Depth Box object

Constructor in Java



Constructors in Java

Java Constructors

Java constructors or constructors in Java is a terminology used to construct something in our programs. A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

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

Constructors in Java

How Java Constructors are Different From Java Methods?

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Constructors in Java

Types of Constructors in Java

Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

- Default Constructor
- Parameterized Constructor
- Copy Constructor

Constructors in Java

1. Default Constructor in Java

A constructor that has no parameters is known as default the constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can't change the default constructor.

```
import java.io.*;

// Driver class
class GFG {

        // Default Constructor
        GFG() { System.out.println("Default constructor"); }

        // Driver function
        public static void main(String[] args)
        {
                GFG hello = new GFG();
        }
}
```

Constructors in Java

2. Parameterized Constructor in Java

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

```
import java.io.*;
class Abc {
        String name;
        int id;
        Abc(String name, int id)
        {
                this.name = name;
                this.id = id;
        }
}
class GFG {
        public static void main(String[] args)
        {
                // This would invoke the parameterized constructor.
                Abc obj1 = new Abc("avinash", 68);
                System.out.println("Abc Name :" + obj1.name+ " and AbcId :" + obj1.id);
        }
}
```

Constructors in Java

3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

Note: *In Java,there is no such inbuilt copy constructor available like in other programming languages such as C++, instead we can create our own copy constructor by passing the object of the same class to the other instance(object) of the class.*

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Constructors in Java

```
// Java Program for Copy Constructor
import java.io.*;

class Geek {
    // data members of the class.
    String name;
    int id;

    // Parameterized Constructor
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    Geek(Geek obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
}
```

```
class GFG {
    public static void main(String[] args)
    {
     // This would invoke the parameterized constructor.
        System.out.println("First Object");
        Geek geek1 = new Geek("avinash", 68);
        System.out.println("GeekName :" + geek1.name
                                                  .id);
```

Output

```
First Object
GeekName :avinash and GeekId :68

Copy Constructor used Second Object
GeekName :avinash and GeekId :68
```

Access Specifiers in Java

1) Private

The private access modifier is accessible only within the class.

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

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

Access Specifiers in Java

1) Private

The private access modifier is accessible only within the class.

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

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

Access Specifiers in Java

2) Default

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

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

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

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Access Specifiers in Java

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

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

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

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

Access Specifiers in Java

3) Public

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

```
//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
```

static in Java

- It is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

static in Java

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods of their class.
- They can only directly access **static** variables of their class.
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

static in Java

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String[] args) {
    meth(42);
  }
}
```

```
Static block initialized.
x = 42
a = 3
b = 12
```

static in Java

For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

static in Java

```
class A{
    static void sum(int a,int b){
        System.out.println("a = "+a);
        System.out.println("b = "+b);
        System.out.println("sum = "+(a+b));
    }
    static{
        System.out.println("I am static block");

    }
}
public class Main{
    public static void main(String[] args){
    A.sum(10,20);
    }
}
```

```
I am static
a = 10
b = 20
sum = 30
```

final in Java

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is probably the most common.

final in Java

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

It is a common coding convention to choose all uppercase identifiers for **final** fields,

final in Java

Final member

```
public class A{
        public static void main(String args[]){
        final int num=10;
        System.out.println(num);
        num=40;
        System.out.println(num);
    }
}
```

```
A.java:5: error: cannot assign a value to final variable num
num=40;
    ^
1 error
```

final in Java

Final method

```
class  A{
    void mNumber(){
        System.out.println("123456789");
    }
    void pin(){
        System.out.println("4321");
    }
}
class B extends A{
    void mNumber(){
         System.out.println("1000000009");
    }
    void pin(){
        System.out.println("1111");
    }
}
public class staticmethod {
        public static void main(String[] args) {
                A obj=new A();
                obj.mNumber();
                obj.pin();
                B obj2=new B();
                obj2.mNumber();
                obj2.pin();
        }
}
```

```
123456789
4321
1000000009
1111
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

final in Java

Final method

```
class  A{
    void mNumber(){
        System.out.println("123456789");
    }
    final void pin(){
        System.out.println("4321");
    }
}
class B extends A{
    void mNumber(){
         System.out.println("1000000009");
    }
    void pin(){
        System.out.println("1111");
    }
}
public class staticmethod {
        public static void main(String[] args) {
                A obj=new A();
                obj.mNumber();
                obj.pin();
                B obj2=new B();
                obj2.mNumber();
                obj2.pin();
        }
}
```

```
staticmethod.java:13: error: pin() in B cannot
    override pin() in A
    void pin(){
         ^
overridden method is final
1 error
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Input from user in Java

To input integer value

```
import java.util.Scanner;
class userInput
{
    public static void main(String args[])
    {
        int a,b;
        Scanner obj=new Scanner(System.in);
        System.out.print("Enter value ");
        a=obj.nextInt();
        b=obj.nextInt();
        System.out.print(a+" "+b);
    }
}
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Input from user in Java

To input a character

```
import java.util.Scanner;
class charPrint
{
    public static void main(String[] args) {

        char ch;
        System.out.print("Please Enter Character ");
        Scanner r=new Scanner(System.in);
        ch=r.next().charAt(0);
        System.out.print(ch);

    }
}
```

Variables in Java

```
class A
{
    static int b=20;//Static
    int c=30;//Instance
    public static void main(String[] args)
    {
        int a=10;//Local
    }
}
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Variables in Java

```
class A
{
    static int b=20;//Static
    int c=30;//Instance
    public static void main(String[] args)
    {
        int a=10;//Local
        A ref=new A();
        System.out.println(a);
        System.out.println(A.b);
        System.out.println(ref.c);


    }
}
```

Variables in Java

```
class A
{
    static int a=10;
    void fun()
    {
        int b=10;
        System.out.println(a+" "+b);
        ++a;  ++b;//11
    }
    public static void main(String[] args)
    {
        A r=new A();
        r.fun();
        r.fun();
    }
}
```

Operators in Java

Operators constitute the basic building block to any programming language. Java too provides many types of operators which can be used according to the need to perform various calculations and functions, be it logical, arithmetic, relational, etc. They are classified based on the functionality they provide.

1.Arithmetic Operators
2.Unary Operators
3.Assignment Operator
4.Relational Operators
5.Logical Operators
6.Ternary Operator
7.Bitwise Operators
8.Shift Operators

Operators in Java

Arithmetic Operators

These operators involve the mathematical operators that can be used to perform various simple or advanced arithmetic operations on the primitive data types referred to as the operands. These operators consist of various unary and binary operators that can be applied on a single or two operands.

Operators	Result
+	Addition of two numbers
-	Subtraction of two numbers
*	Multiplication of two numbers
/	Division of two numbers
%	(Modulus Operator)Divides two numbers and returns the remainder

Operators in Java

Unary Operators

Operator 1: Unary minus(-)

Operator 2: 'NOT' Operator(!)

Operator 3: Increment(++)

- **3.1:** Post-increment operator
- **3.2:** Pre-increment operator

Operator 4: Decrement (--)

- **4.1:** Post-decrement operator
- **4.2:** Pre-decrement operator

Operator 5: Bitwise Complement(~)

Operators in Java

Assignment Operators

These operators are used to assign values to a variable. The left side operand of the assignment operator is a variable, and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type of the operand on the left side.

1. **Simple Assignment Operator: =**
2. **Compound Assignment Operator:**
   - **(+=) operator:**
   - **(-=) operator**
   - **(*=) operator**
   - **(/=) operator**
   - **(%=) operator**

Operators in Java

Java Relational Operators are a bunch of binary operators used to check for relations between two operands, including equality, greater than, less than, etc. They return a boolean result after the comparison and are extensively used in looping statements as well as conditional if-else statements and so on.

'Equal to' operator (==)
'Not equal to' Operator(!=)
'Greater than' operator(>)
'Less than' Operator(<)
Greater than or equal to (>=)
Less than or equal to (<=)

Operators in Java

Logical operators are used to perform logical “AND”, “OR” and “NOT” operations, i.e. the function similar to AND gate and OR gate in digital electronics.

1.AND Operator (**&&**) – if(a && b) [if true execute else don’t]

2.OR Operator (**||**) – if(a || b) [if one of them is true to execute else don’t]

3.NOT Operator (**!**) – !(a<b) [returns false if a is smaller than b]

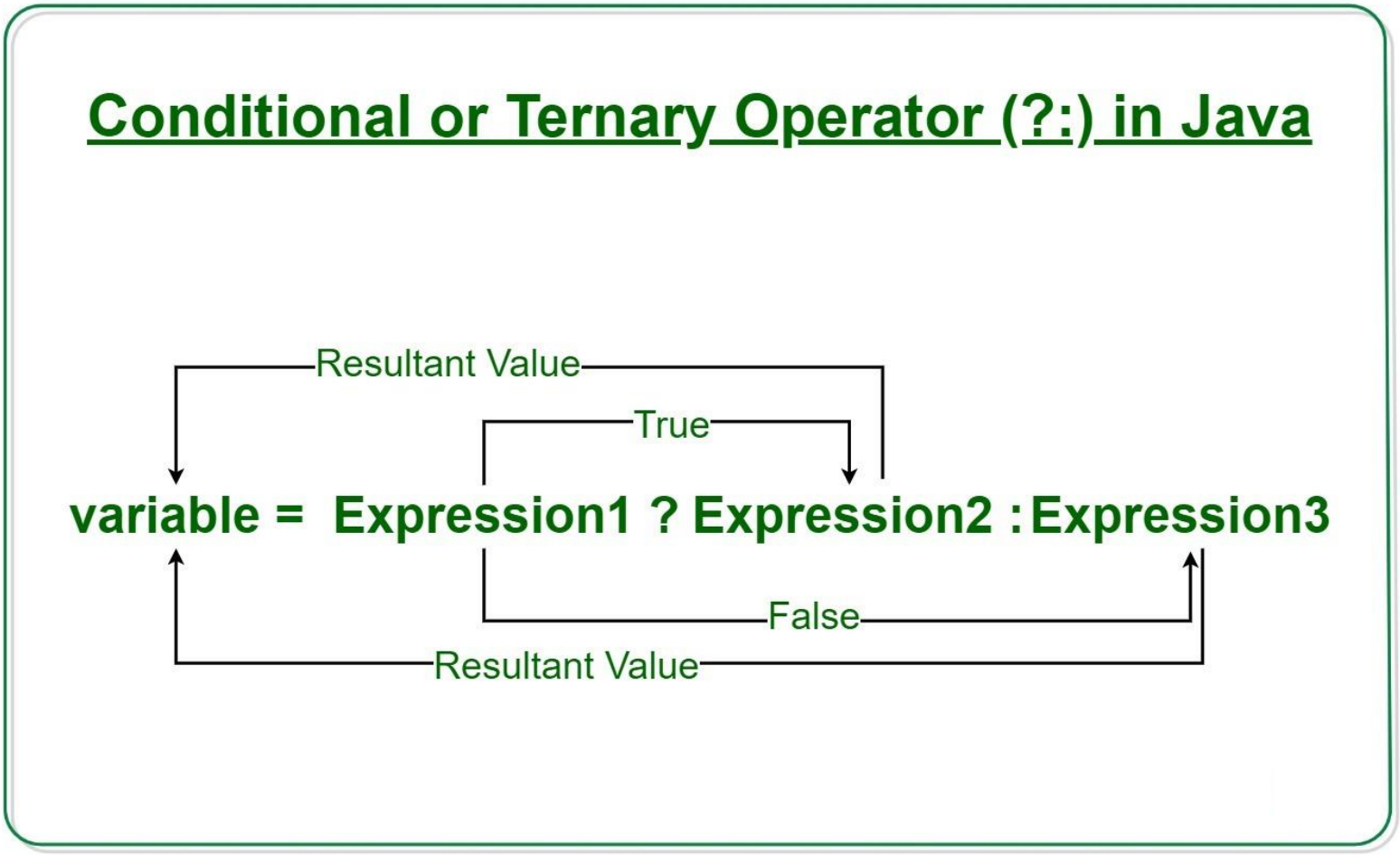
Operators in Java

Ternary Operator in Java

Java ternary operator is the only conditional operator that takes three operands. It's a one-liner replacement for the if-then-else statement and is used a lot in Java programming. We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators. Although it follows the same algorithm as of if-else statement, the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.

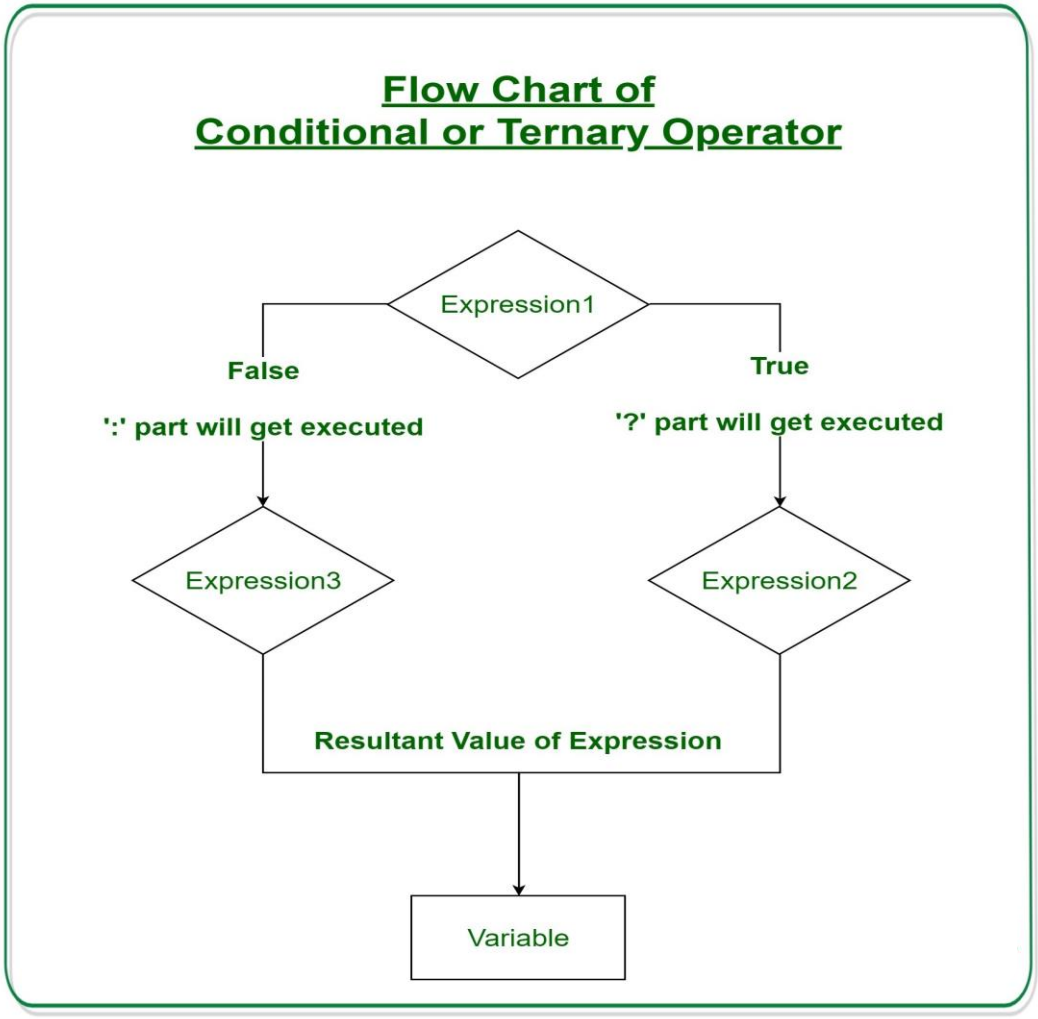
Operators in Java

Ternary Operator in Java



Operators in Java

Ternary Operator in Java



Operators in Java

Bitwise Operators

Bitwise operators are used to performing the manipulation of individual bits of a number. They can be used with any integral type (char, short, int, etc.). They are used when performing update and query operations of the Binary indexed trees.

Bitwise OR (|)

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7
  0101
| 0111
 ________
  0111  = 7 (In decimal)
```

Operators in Java

Bitwise Operators

Bitwise operators are used to performing the manipulation of individual bits of a number. They can be used with any integral type (char, short, int, etc.). They are used when performing update and query operations of the Binary indexed trees.

Bitwise OR (|)
Bitwise AND (&)
Bitwise XOR (^)
Bitwise Complement (~)

Operators in Java

Bitwise Operators

Bit-Shift Operators (Shift Operators)

Shift operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two.

Shift Operators are further divided into 4 types. These are:

1.Signed Right shift operator (>>)
2.Unsigned Right shift operator (>>>)
3.Left shift operator(<<)
4.Unsigned Left shift operator (<<<)

Control Flow in Java

Java provides three types of control flow statements.

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

Control Flow in Java

Decision-Making statements:

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

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

Control Flow in Java

Decision-Making statements:

```
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
}
```

Control Flow in Java

Decision-Making statements:

Nested if-statement

```
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
}
}
```

Control Flow in Java

Switch Statement:

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.

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

Control Flow in Java

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.

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

Control Flow in Java

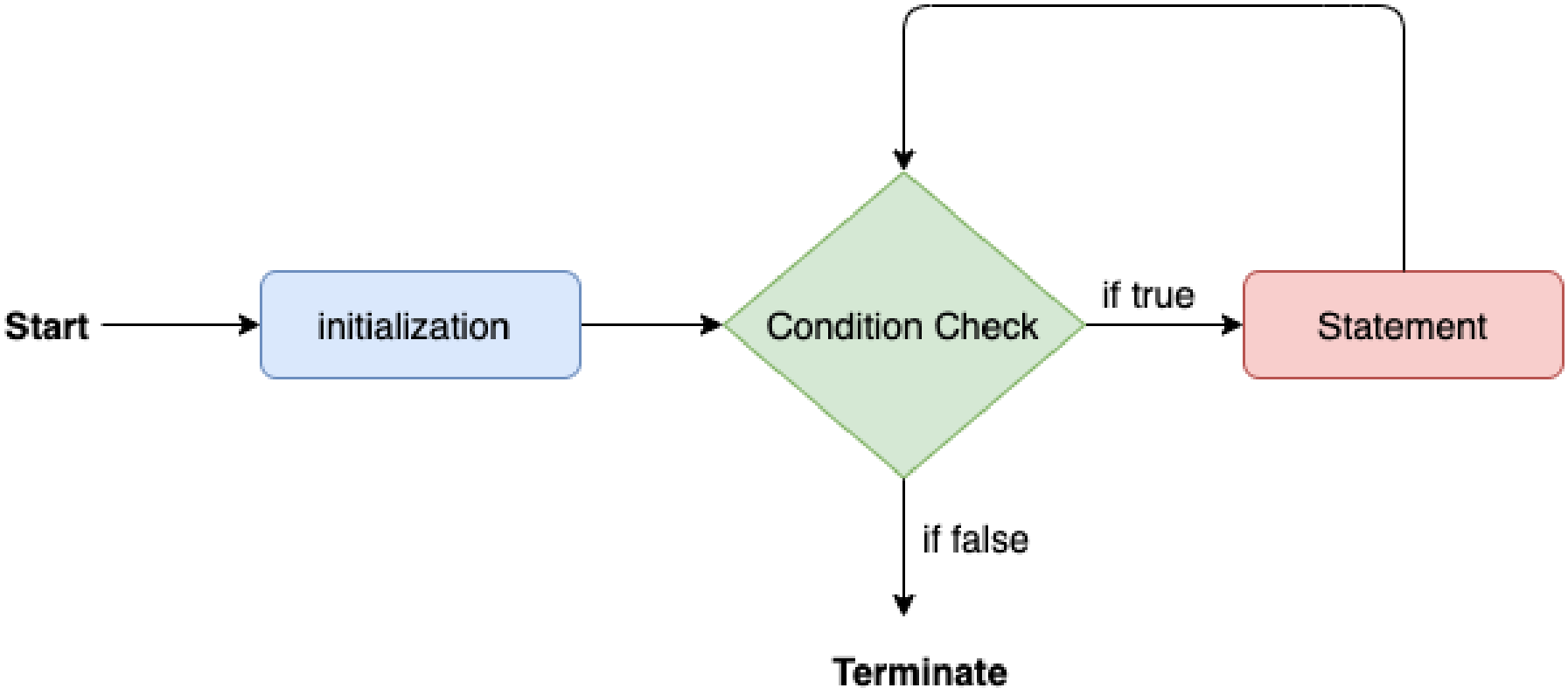
Java for loop

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
}
```

Control Flow in Java

Java for loop



Control Flow in Java

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.

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

Control Flow in Java

Java for-each loop

```
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
```

Control Flow in Java

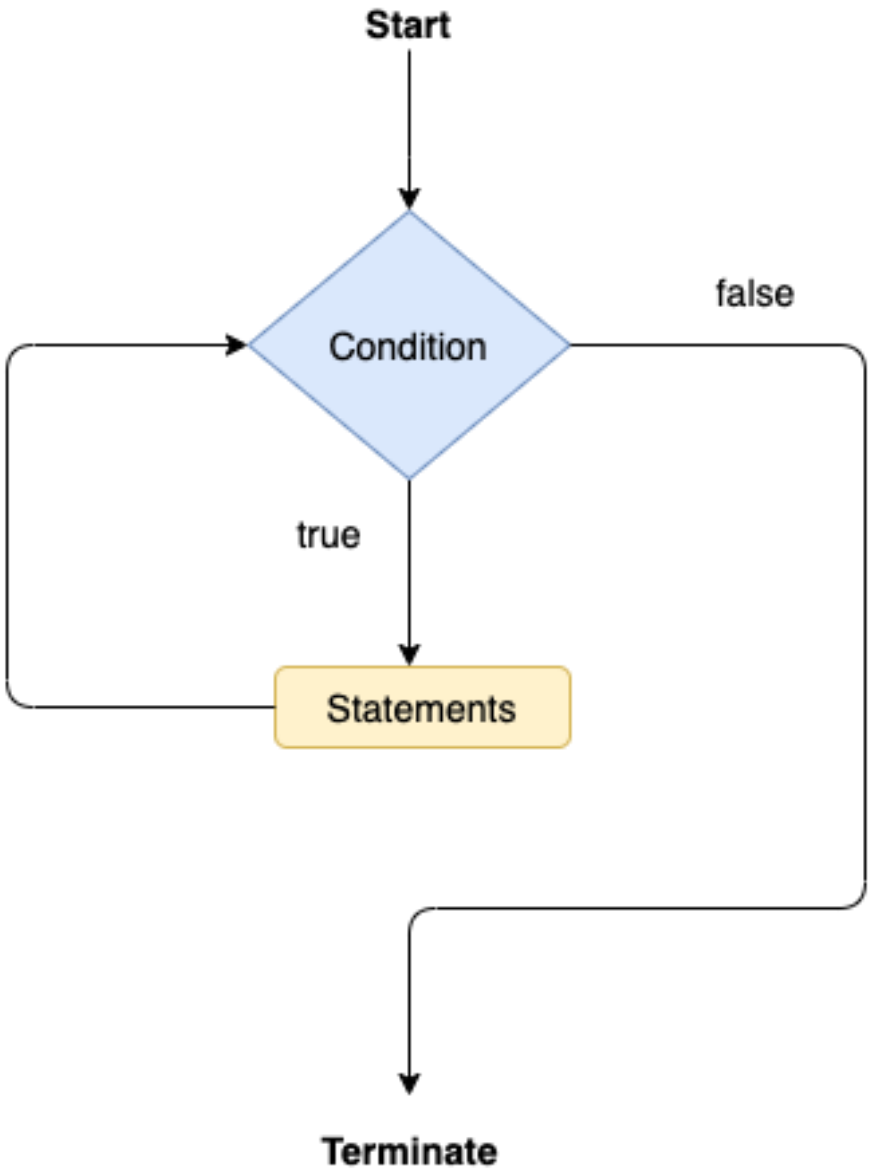
Java while loop

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.

```
while(condition){
//looping statements
}
```

Control Flow in Java

Java while loop



Control Flow in Java

Java do-while loop

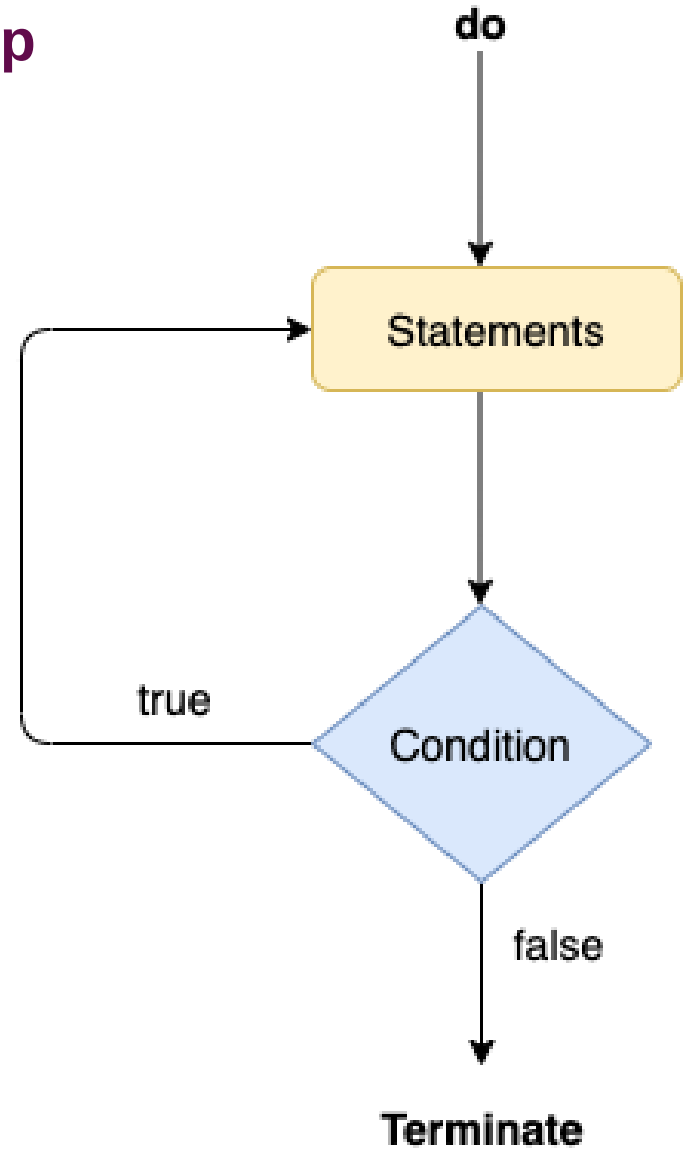
The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

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);
```

Control Flow in Java

Java do-while loop



Control Flow in Java

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.

Control Flow in 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;
}
}
}
}
```

Control Flow in Java

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.

```
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);
}
}
}

}
```

Arrays in Java

Java Arrays

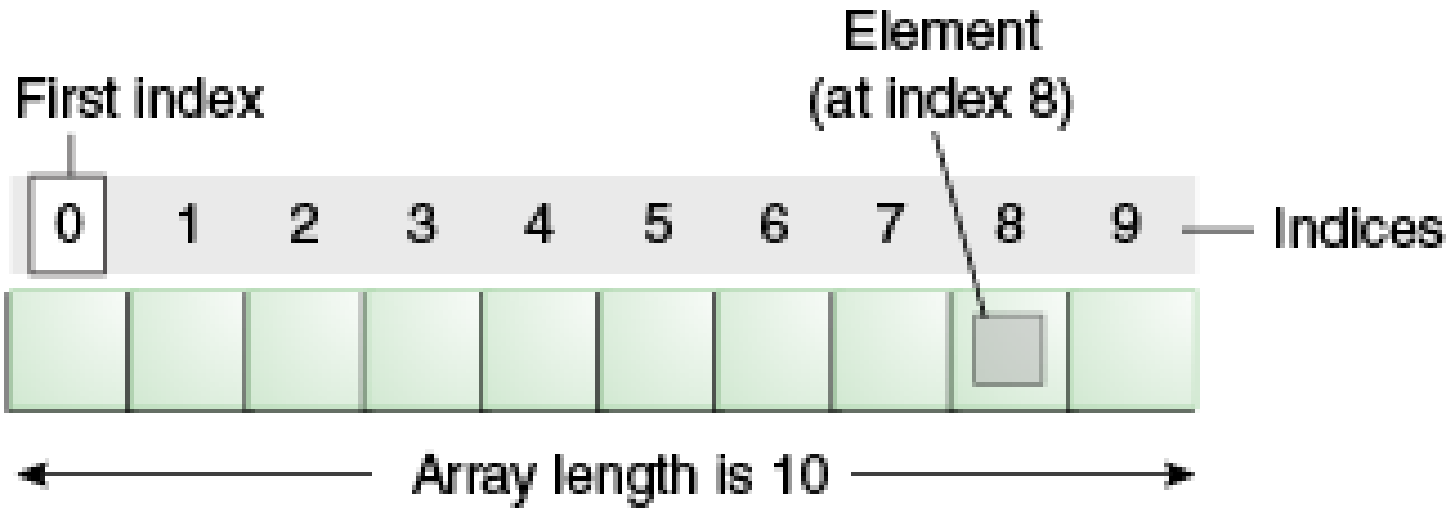
Normally, an array is a collection of similar type of elements which has contiguous memory location.

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.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

Arrays in Java



Advantages

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

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

Disadvantages

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

Arrays in Java

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Arrays in Java

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];
```

Arrays in Java

Single D
Java

```
//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]);
}}
```

Arrays in Java

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
```

```
//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]);
}}
```

Arrays in Java

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Arrays in Java

```
//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);
}}
```

Output:

```
33
3
4
5
```

Arrays in Java

Multidimensional Array in Java

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

Syntax to Declare Multidimensional Array in Java

1.dataType[][] arrayRefVar; (or)
2.dataType [][]arrayRefVar; (or)
3.dataType arrayRefVar[][]; (or)
4.dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

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

Arrays in Java

Multidimensional Array in Java

```
//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();
}
}}
```

Arrays in Java

Addition of 2 Matrices in Java

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
 }}
```

Arrays in Java

Multiplication of 2 Matrices in Java

```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices

int c[][]=new int[3][3];  //3 rows and 3 columns
```

Arrays in Java

Multiplication of 2 Matrices in Java

```
//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}}
```

Strings in Java

Java String

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

For example:

```
char[] ch={‘r',’k',’g',’i',’t',’g',’h',’a',’z’,’i’,’a’,’b’,’a’,’d’};
String s=new String(ch);
is same as:
String s=“rkgitghaziabad";
```

Strings in Java

Java String

Java String class provides a lot of methods to perform operations on strings such as toLowerCase(), toUpperCase(),compare(), concat(), equals(), length(), replace(), compareTo(), indexof(), substring() etc.

```
public class Main {
  public static void main(String[] args) {
    String str = "Hello how are you";
    System.out.println(str);
    System.out.println(str.toUpperCase());
    System.out.println(str.toLowerCase());
    System.out.println(str.indexOf("how"));
  }
}
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

Use of inheritance in java

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

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Inheritance in Java

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing

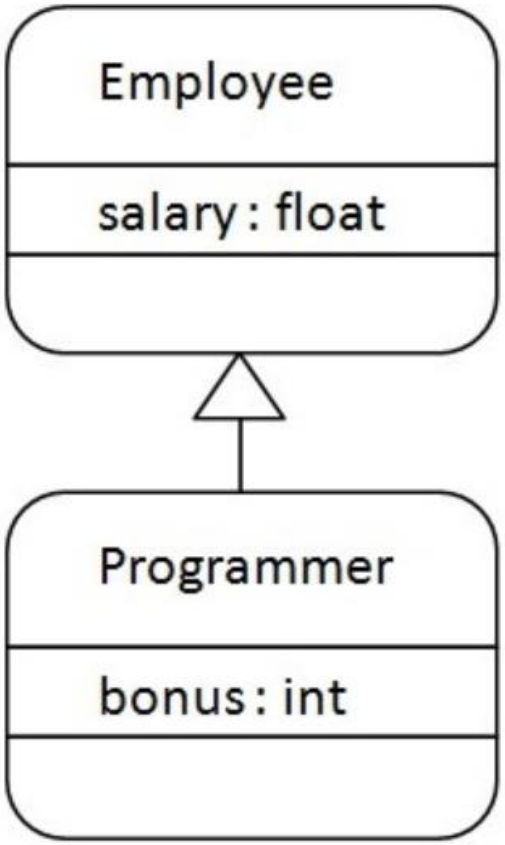
Inheritance in Java

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

Inheritance in Java

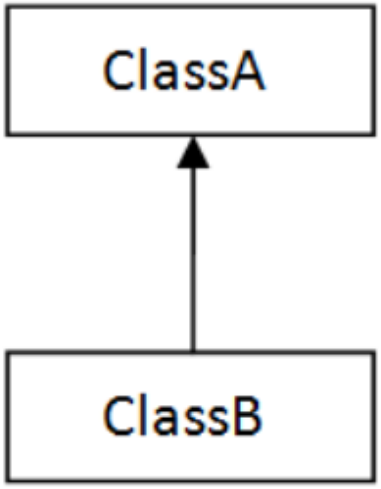
Java Inheritance Example



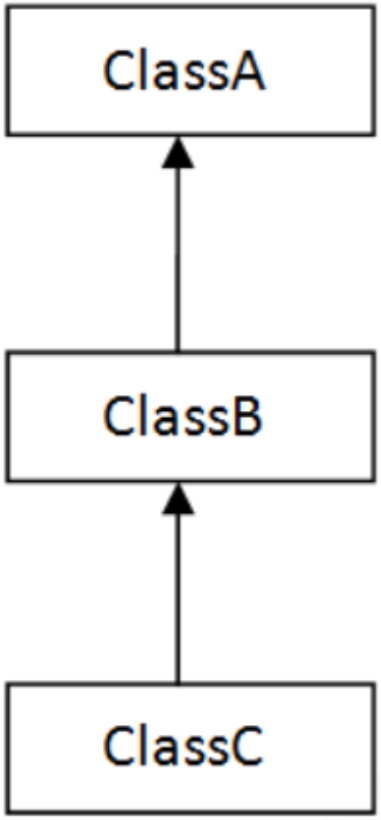

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Inheritance in Java

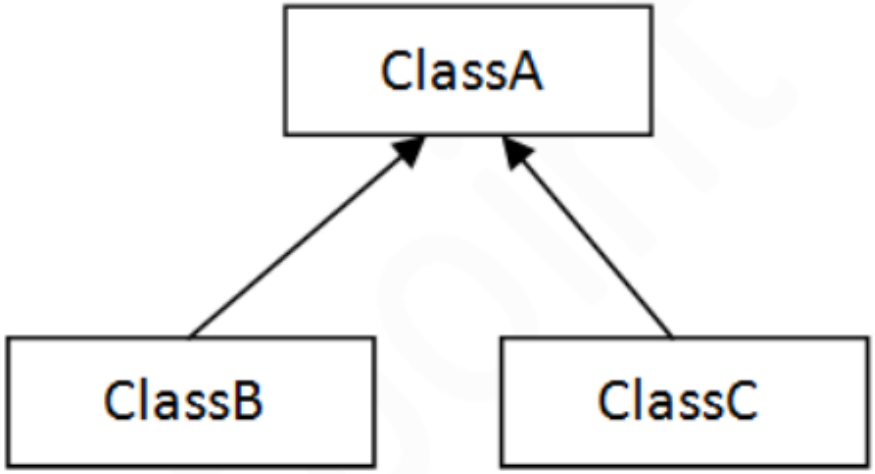
Types of inheritance in java



1) Single



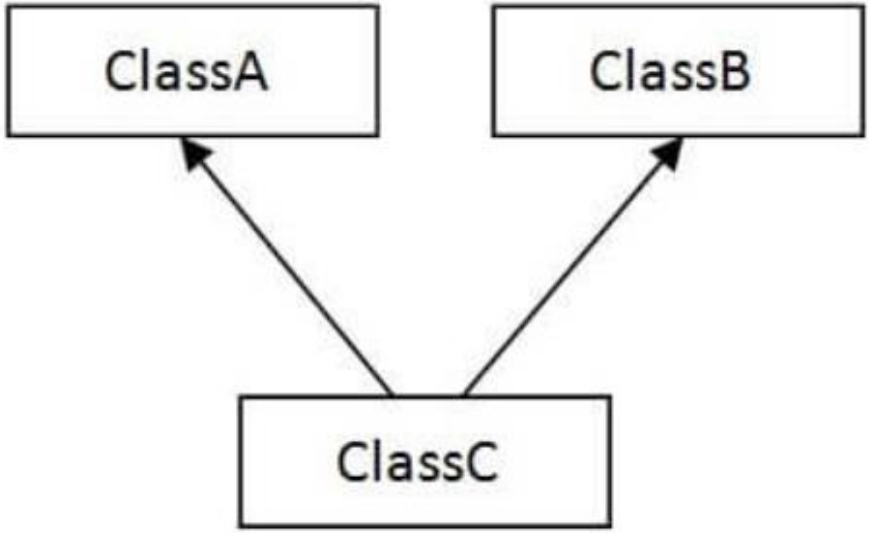
2) Multilevel



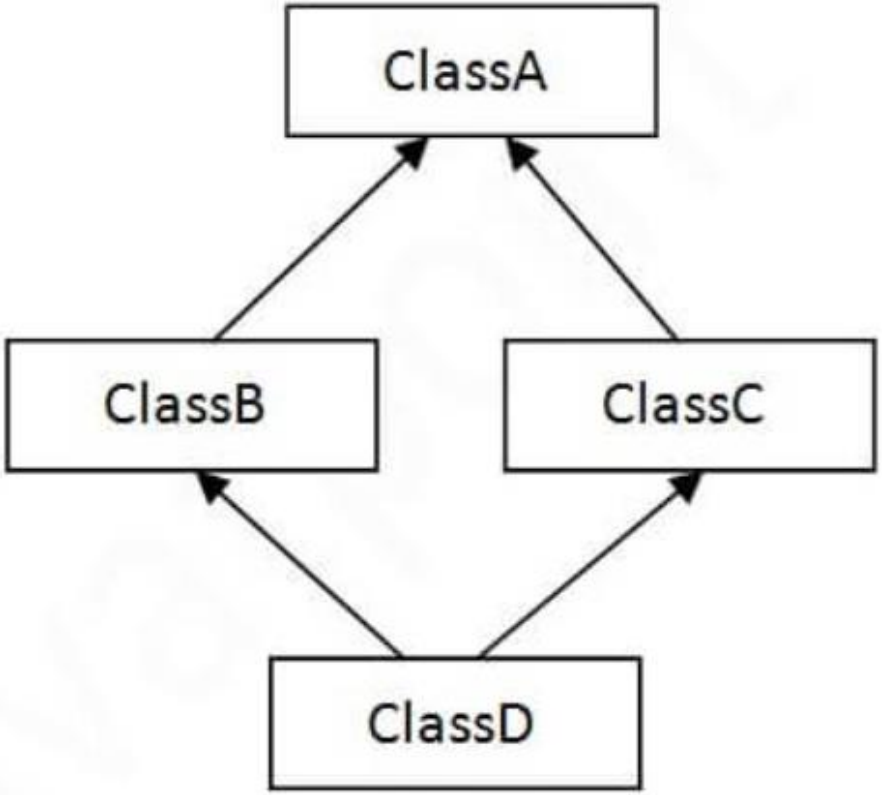
3) Hierarchical

Inheritance in Java

Types of inheritance in java



4) Multiple



5) Hybrid

Inheritance in Java

Single Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

Inheritance in Java

Multilevel Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Inheritance in Java

Hierarchical Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Inheritance in Java

Multiple inheritance is not supported in java?

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Compile Time Error

Method Overloading in Java

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading

Method overloading *increases the readability of the program.*

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

Different ways to overload the method

There are two ways to overload the method in java

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

Method Overloading in Java

1) Method Overloading: changing no. of arguments

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

Method Overloading in Java

2) Method Overloading: changing data type of arguments

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

Output:

```
22
24.9
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Method Overloading in Java

Method Overloading is not possible by changing the return type of method only

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

Output:

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

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

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

Rules for Java Method Overriding

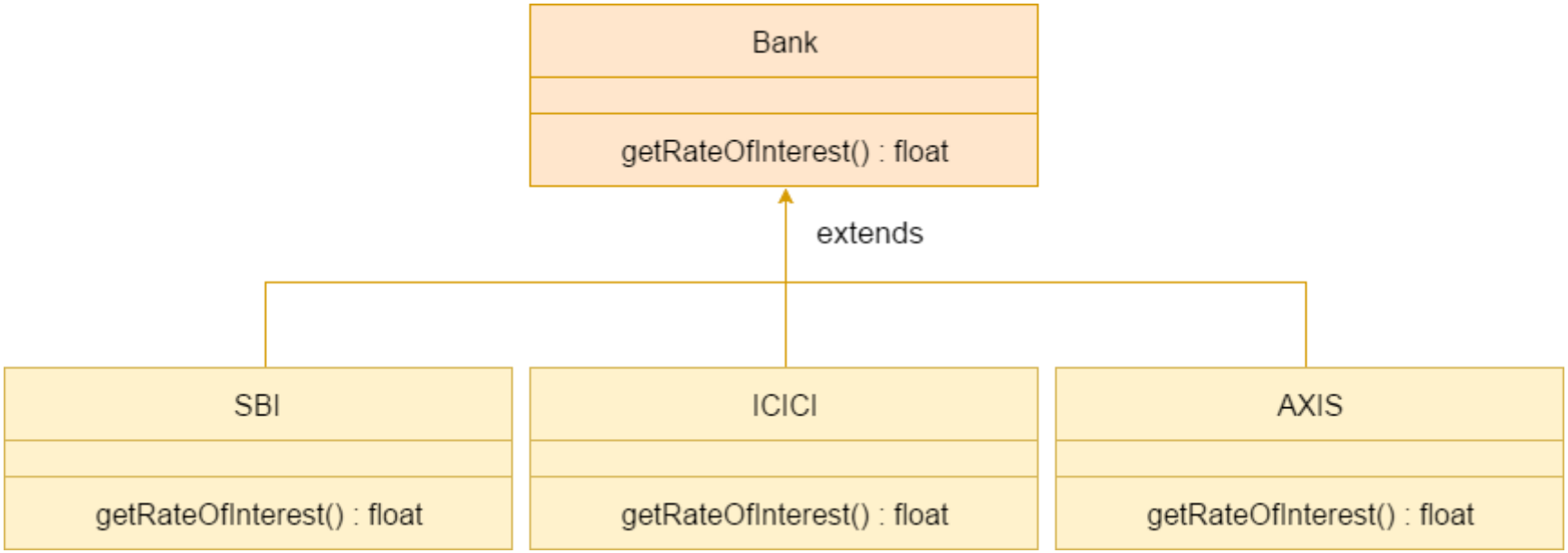
1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Method Overriding in Java

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Method Overriding in Java



Method Overriding in Java

```
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
```

Method Overriding in Java

```
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest()
);
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest(
));
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest
());
}
}
```

Method Overriding in Java

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Difference between Overlaoding & Overriding in Java

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods

Encapsulation in Java

Advantage of Encapsulation in Java

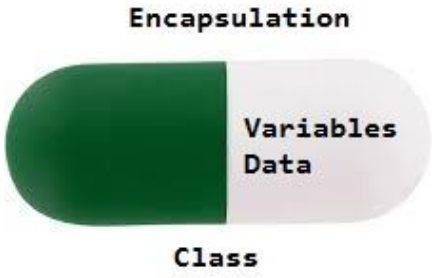
By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.



Encapsulation in Java

Advantage of Encapsulation in Java

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Encapsulation in Java

Achieving Encapsulation in Java

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Encapsulation in Java

```
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;    }

   public String getName() {
      return name;    }

   public String getIdNum() {
      return idNum;    }

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

   public void setName(String newName) {
      name = newName;   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

Encapsulation in Java

The public **`set---()`** and **`get---()`** methods are the access points of the instance variables of the **`EncapTest`** class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

Encapsulation in Java

```
//A Java class which is a fully encapsulated class.
//It has a private data member and getter and setter method
s.
package com.javatpoint;
public class Student{
//private data member
private String name;
//getter method for name
public String getName(){
return name;
}
//setter method for name
public void setName(String name){
this.name=name
}
}
```

Encapsulation in Java

```
package com.javatpoint;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Abstraction in Java

Abstraction is a feature of OOPs. The feature allows us to **hide** the implementation detail from the user and shows only the functionality of the programming to the user. Because the user is not interested to know the implementation. It is also safe from the security point of view.

The best example of abstraction is a car. When we derive a car, we do not know **how is the car moving** or **how internal components are working?** But we know **how to derive a car**. It means it is not necessary to know how the car is working, but it is important how to derive a car. The same is an abstraction.

Abstraction in Java

We can achieve abstraction in two ways:

- Using Abstract Class
- Using Interface

Abstraction in Java

Using Abstract Class

Abstract classes are the same as normal Java classes the difference is only that an abstract class uses abstract keyword while the normal Java class does not use. We use the abstract keyword before the class name to declare the class as abstract.

we cannot instantiate (create an object) an abstract class. An abstract class contains abstract methods as well as concrete methods. If we want to use an abstract class, we have to inherit it from the base class.

If the class does not have the implementation of all the methods of the interface, we should declare the class as abstract. It provides complete abstraction. It means that fields are public static and final by default and methods are empty.

Abstraction in Java

The syntax of abstract class is:

```
public abstract class ClassName
{
public abstract methodName();
}
```

Abstraction in Java

```
//abstract class
abstract class Demo
{
//abstract method
abstract void display();  }
//extends the abstract class
public class MainClass extends Demo
{
//defining the body of the method of the abstract class
void display()  {
System.out.println("Abstract method called.");
}
public static void main(String[] args)
{
MainClass obj = new MainClass ();
//invoking abstract method
obj.display();
}
}
```

Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

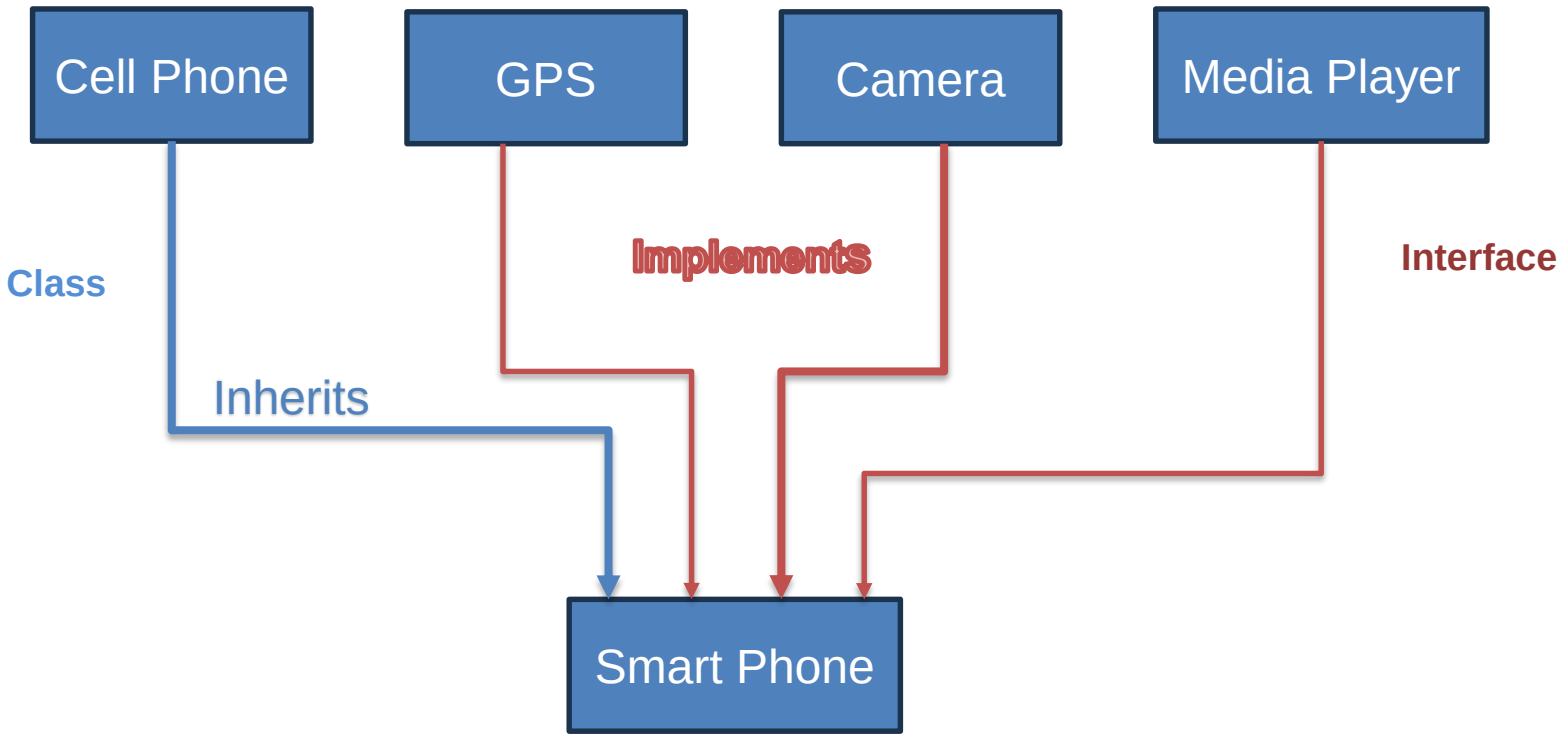
Why use Java interface?

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.

Interface in Java

The relationship between classes and interfaces



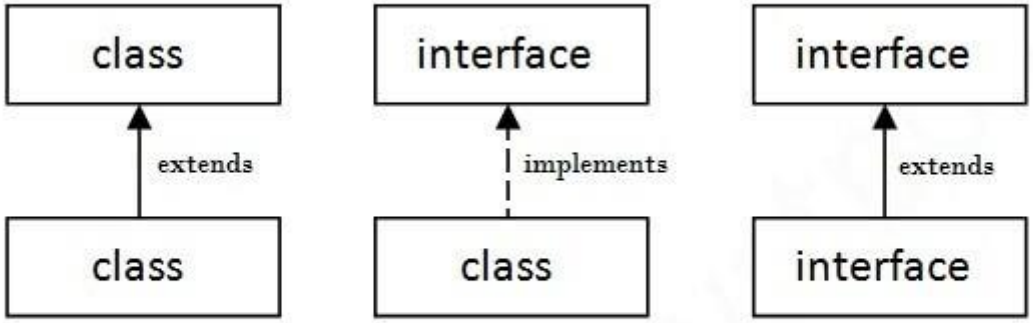
Interface in Java

Using Interface

In Java, an interface is similar to Java classes. The difference is only that an interface contains empty methods (methods that do not have method implementation) and variables. In other words, it is a collection of abstract methods (the method that does not have a method body) and static constants. The important point about an interface is that each method is **public** and **abstract** and does not contain any constructor. Along with the abstraction, it also helps to achieve multiple inheritance. The implementation of these methods provided by the clients when they implement the interface.

Interface in Java

The relationship between classes and interfaces



Interface in Java

Features of Interface:

- We can achieve total abstraction.
- We can use multiple interfaces in a class that leads to multiple inheritance.
- It also helps to achieve loose coupling.

To use an interface in a class, Java provides a keyword called **implements**. We provide the necessary implementation of the method that we have declared in the interface.

Interface in Java

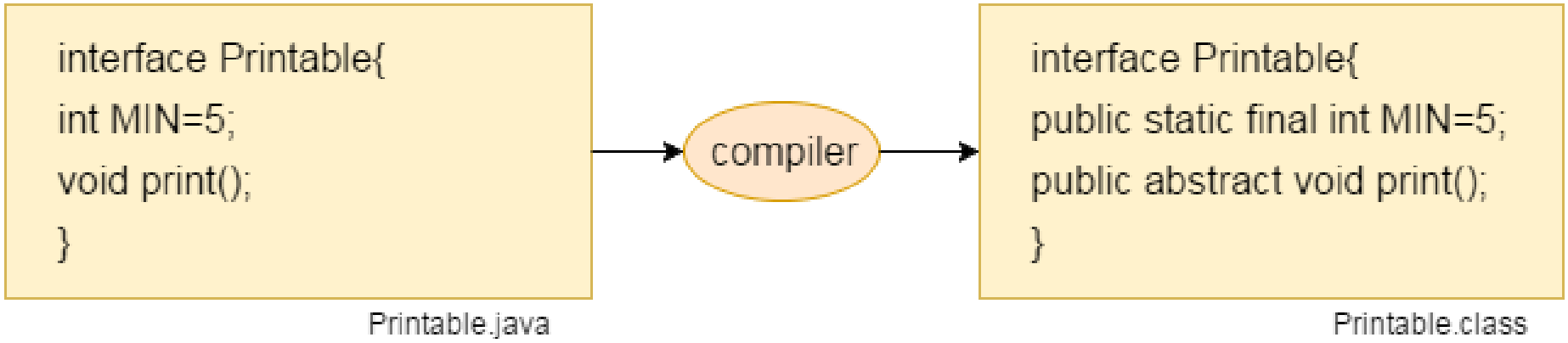
Syntax:

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Interface in Java

Interface fields are public, static and final by default, and the methods are public and abstract.



Interface in Java

Car.java

```
interface CarStart
{    void start();     }
interface CarStop
{    void stop();    }
public class Car implements CarStart, CarStop
{    public void start()
     {  System.out.println("The car engine has been started.");   }
public void stop()
     {  System.out.println("The car engine has been stopped.");   }
public static void main(String args[])
   {
    Car c = new Car();
    c.start();
    c.stop();    }
 }
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Interface in Java

Difference Between Class and Interface

Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (with implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

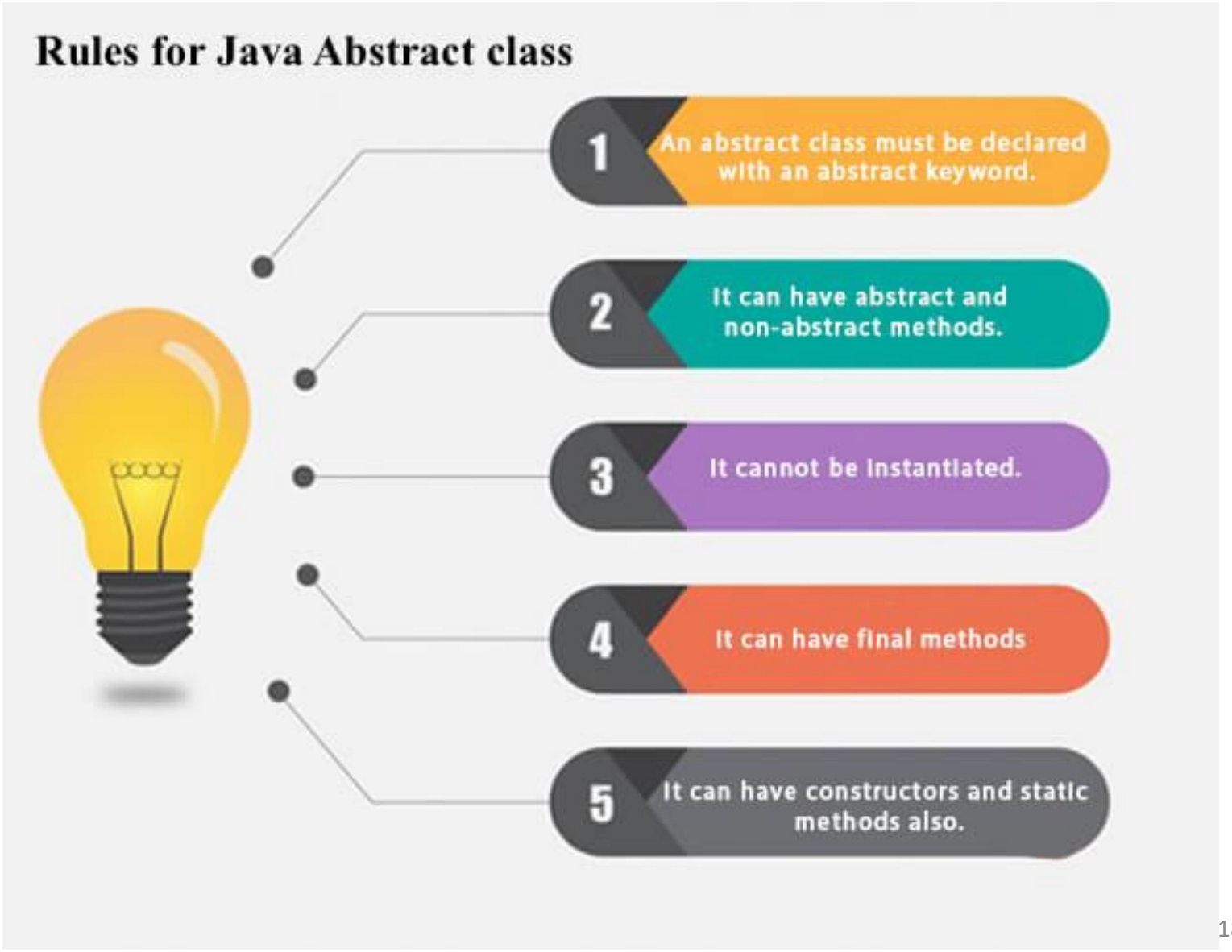
Abstract Class in Java

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter)

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Abstract Class in Java



Abstract Class in Java

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

Abstract Class in Java

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}
// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() { // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Packages in Java

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

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

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

Packages in Java

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:

```
javac -d directory javafilename
```

Packages in Java

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

Packages in Java

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

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

Packages in Java

How to access package from another package?

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

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

Packages in Java

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.

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

```
Output:Hello
```

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

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

Packages in Java

2)Using packagename.classname

```
//save by A.java

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

```
Output:Hello
```

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

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

Packages in Java

3) Using 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 ar
gs[]){
   pack.A obj = new pack.A();
//using fully qualified name
   obj.msg();
  }
}
```

```
Output:Hello
```

Static import in Java

In Java, static import concept is introduced in 1.5 version. With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly. According to SUN microSystem, it will improve the code readability and enhance coding. But according to the programming experts, it will lead to confusion and not good for programming. If there is no specific requirement then we should **not** go for static import.

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Static import in Java

```
// Java Program to illustrate calling of predefined
methods without static import
class Geeks {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }
}
```

Output:

```
2.0
4.0
6.3
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Static import in Java

```
// Java Program to illustrate calling of predefined
methods with static import
import static java.lang.Math.*;
class Test2 {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

Output:

```
2.0
4.0
6.3
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Static import in Java

```
// Java to illustrate calling of static member of
// System class without Class name
import static java.lang.Math.*;
import static java.lang.System.*;
class Geeks {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }
}
```

Output:

```
2.0
4.0
6.3
```

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Static import in Java

Ambiguity in static import:

If two static members of the same name are imported from multiple different classes, the compiler will throw an error, as it will not be able to determine which member to use in the absence of class name qualification.

```
/ Java program to illustrate ambiguity in case of static import
import static java.lang.Integer.*;
import static java.lang.Byte.*;
public class Geeks {
    public static void main(String[] args)
    {
        system.out.println(MAX_VALUE);
    }
}
```

Here compiler will be confused by seeing two import statements because both Integer and Byte class contains a static variable MAX_VALUE .

Output:

```
Error:Reference to MAX_VALUE is ambiguous
```