# Unit I

# Introduction java


## Dr.G.Nagarajan

**PROFESSOR,**
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**
**SCHOOL OF COMPUTING**
**SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY**
**DEEMED TO BE UNIVERSITY**
**CHENNAI- 119, TAMIL NADU,**
**INDIA**

# What is Java?

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

# Introduction to Java programming language

- Today Java programming language is one of the most popular programming language which is used in critical applications like stock market trading system on BSE, banking systems or android mobile application.

- Java was developed by James Gosling from Sun Microsystems in 1995 as an object-oriented language for general-purpose business applications and for interactive, Web-based Internet applications. The goal was to provide platform-independent alternative to C++.

# Introduction to Java programming language

- In other terms it is architecturally neutral, which means that you can use Java to write a program that will run on any platform or device (operating system).

- Java program can run on a wide variety of computers because it does not execute instructions on a computer directly. Instead, Java runs on a Java Virtual Machine (JVM).

# Why Java?

- Java has been tested, refined, extended, and proven by a dedicated community of Java developers, architects and enthusiasts.
- **Simple Grammar** – Java has a very simple grammar familiar to anyone with experience in C and C++
- **Portability** – These days Java really does run well on all the popular platforms
- **Speed** – The latest JIT compilers for Suns JVM approach the speed of C/C++ code, and in some memory allocation intensive circumstances, exceed it.
- **Garbage Collection** – the programmer doesn't have to worry about memory (most of the time)
- **Huge library** and developer community support available on Internet.

# Where it is used?

- According to Sun, 3 billion devices run java. There are many devices where java is currently used. Some of them are as follows:
  - Desktop Applications such as acrobat reader, media player, antivirus etc.
  - Web Applications such as irctc.co.in, javatpoint.com etc.
  - Enterprise Applications such as banking applications.
  - Mobile
  - Embedded System
  - Smart Card
  - Robotics
  - Games
  - etc.

# History of Java

- Java language developed by company **Sun Microsystems** and creator is **Jamesh Gosling**.
- Java was developed by James Gosling, Patrick Naughton, Mike Sheridan at Sun Microsystems Inc. in 1991. It took 18 months to develop the first working version.
- The initial name was **Oak** but it was renamed to **Java** in 1995 as OAK was a registered trademark of another Tech company.

# Overview of Java

- **J2SE**
  - J2SE is used for developing client side applications.
- **J2EE**
  - J2EE is used for developing server side applications.
- **J2ME**
  - J2ME is used for developing mobile or wireless application by making use of a predefined protocol called WAP(wireless Access / Application protocol).

# Application of Java

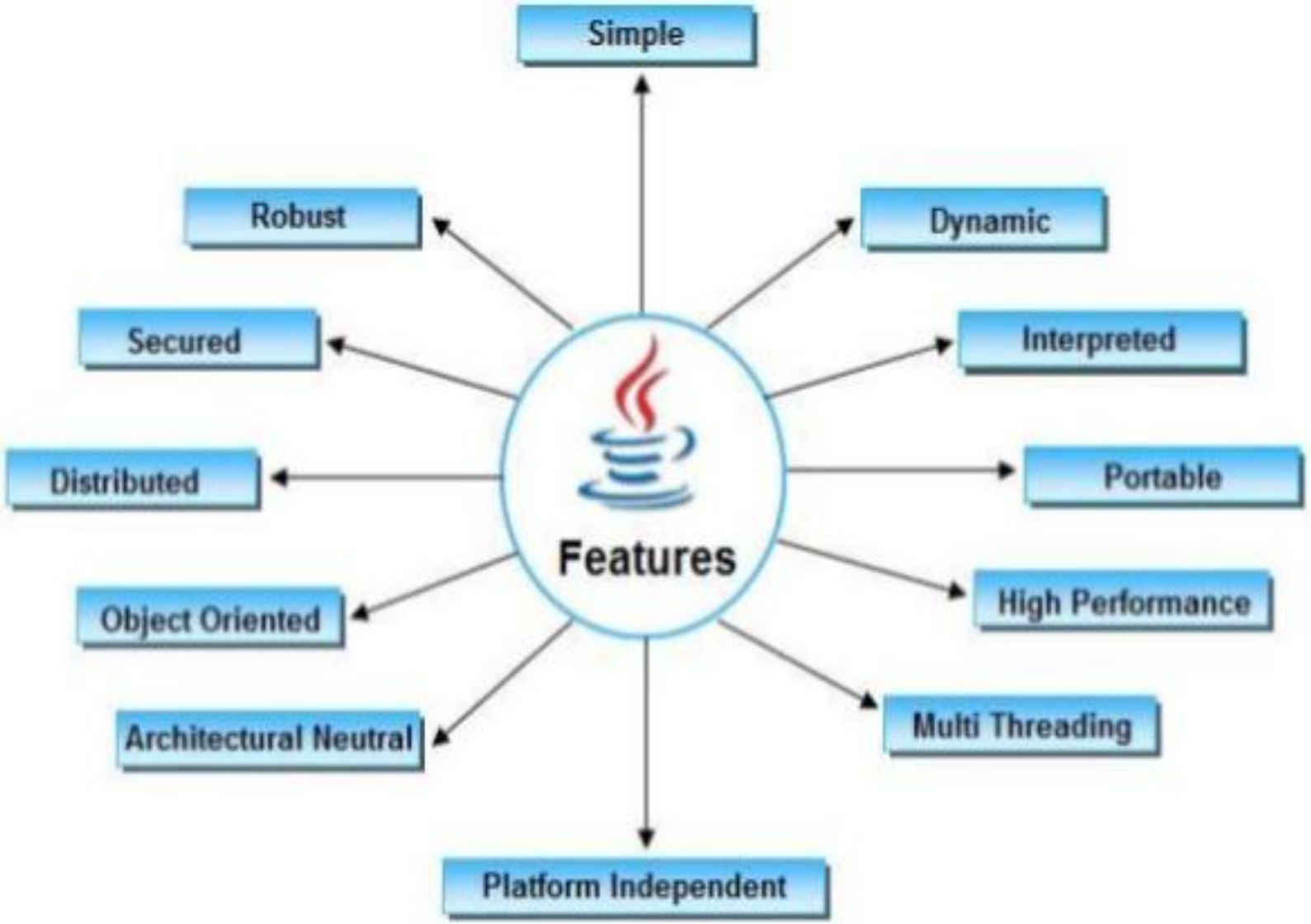
- Java is widely used in every corner of world and of human life. Java is not only used in software's but is also widely used in designing hardware controlling software components. There are more than 930 million JRE downloads each year and 3 billion mobile phones run java.
- **Following are some other usage of Java :**
  - Developing Desktop Applications
  - Web Applications like Linkedin.com, Snapdeal.com etc.
  - Mobile Operating System like Android
  - Embedded Systems
  - Robotics and games etc.

# Introduction of Java

- **Principles of Java Programming language:**
- There were five primary goals in the creation of the Java language.
- It should be “simple, object-oriented and familiar”
- It should be “robust and secure”
- It should be “architecture-neutral and portable”
- It should execute with “high performance”
- It should be “interpreted, threaded, and dynamic”

# Features of Java

- There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.
  - Simple
  - Object-Oriented
  - Platform independent
  - Secured
  - Robust
  - Architecture neutral
  - Portable
  - Dynamic
  - Interpreted
  - High Performance
  - Multithreaded
  - Distributed


Simple
Robust
Dynamic
Secured
Interpreted
Distributed
Portable
Features
Object Oriented
High Performance
Architectural Neutral
Multi Threading
Platform Independent

# Features of Java

- **Simple**
- According to Sun, Java language is simple because:
  - Syntax is based on C++ (so easier for programmers to learn it after C++).
  - Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.
  - No need to remove unreferenced objects because there is Automatic Garbage Collection in java.
  - And also removed many other confusing and/or rarely used features like explicit pointer, operator overloading etc.

# Features of Java

- **Object-oriented**
- Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.
- Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules.
- Basic concepts of OOPs are:
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation

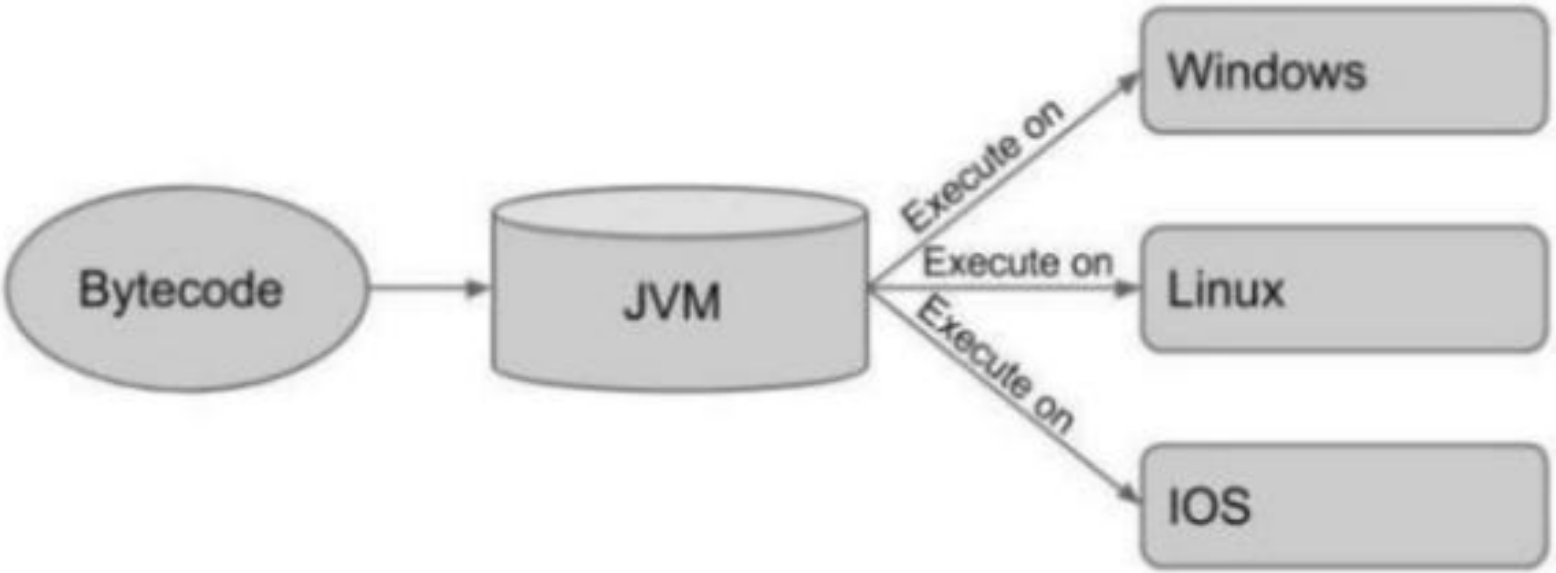
# Features of Java

- **Platform Independent**
- A platform is the hardware or software environment in which a program runs.
- There are two types of platforms software-based and hardware-based. Java provides software-based platform.
- The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:
  - Runtime Environment
  - API(Application Programming Interface)

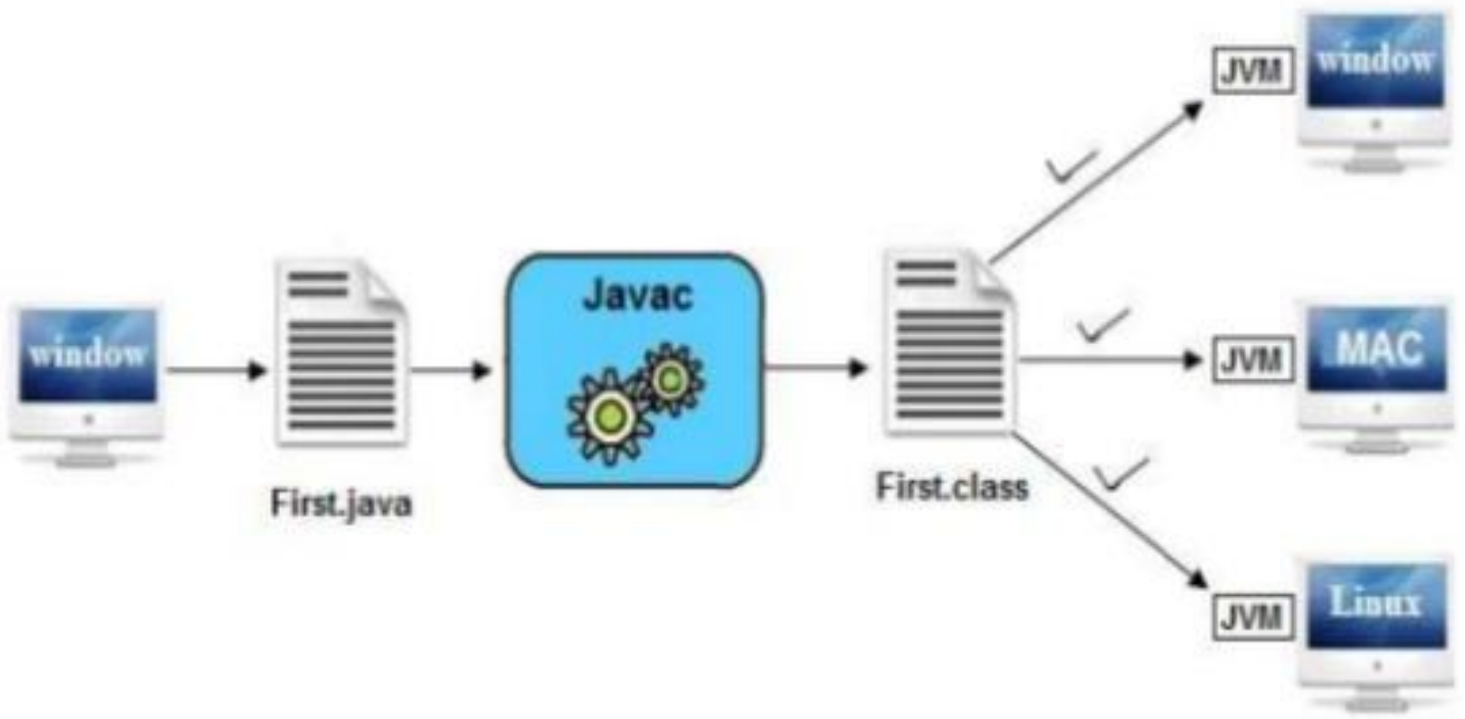
# Features of Java

- **Platform Independent**
- Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.
- This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).
- Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler does. Instead it produces a special format called byte code. Java byte code written in hexadecimal, byte by byte, looks like this:
  CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20

# Platform Independent



# Platform Independent

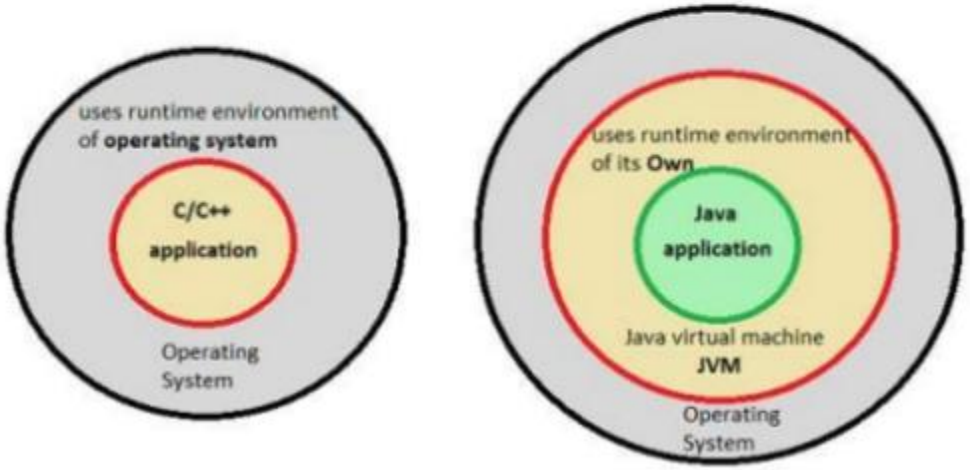


# Features of Java

- **Secured**
- Java is a secure programming language because:
- No explicit pointer
- Program run inside virtual machine sandbox.
- Array index limit checking
- Code pathologies reduced by :
  - **Bytecode verifier** – Checks classes after loading (code fragments for illegal code that can violate access right to object.)
  - **Classloader** – confines objects to unique namespaces. Prevents loading a hacked "java.lang.SecurityManager" class.
  - **Security manager** – determines what resource a class can access such as reading and writing to the local disk.

# Secured

Program run inside virtual machine sandbox



# Features of Java

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

# Features of Java

- **Architecture-Neutral**
- There is no implementation dependent features e.g. size of primitive types is fixed.
- In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.
- A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering there architecture and vendor (providers) irrespective to its development and compilation.

# Features of Java

- **High performance**
- It have high performance because of following reasons;
- This language **uses Bytecode** which is more faster than ordinary pointer code so Performance of this language is high.
- **Garbage collector**, collect the unused memory space and improve the performance of application.
- It have **no pointers** so that using this language we can develop an application very easily.
- It **support multithreading**, because of this time consuming process can be reduced to execute the program.

# Features of Java

- **Networked**
- It is mainly design for web based applications, J2EE is used for developing network based applications.
- **Dynamic**
- It support Dynamic memory allocation due to this memory wastage is reduce and improve performance of application. The process of allocating the memory space to the input of the program at a run-time is known as dynamic memory allocation, To programming to allocate memory space by dynamically we use an operator called 'new' 'new' operator is known as dynamic memory allocation operator.

# Features of Java

- **Distributed**
- We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.
- **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

- **What is Byte Code in java**
- The code which is converted by java compiler(**javac**) is called **byte code** in java.
- **Why Java is Portable?**
- The code which is converted by java compiler(**javac**) is non executable.
- **Byte code** is a highly optimized set of instructions
- **Byte code** is executed by Java run-time system, which is called the **java virtual machine (JVM)**.
- **Byte code** is **intermediate language** of source code and **executable** code.
- JVM converts byte code of java into machine language to execute **microprocessor** of OS.
- However internal details of **JVM** will differ from platform to platform but still all **JVM** understands the same java **bytecode. Or** in other word we can say **JVM** is platform dependent and **java byte code is platform independent**.

# Applications of Java

# How is Java different from C...

- **C Language:**
  - Major difference is that C is a **structure oriented language** and Java is an **object oriented language** and has mechanism to define classes and objects.
  - Java does not support an explicit **pointer** type
  - Java does not have **preprocessor**, so we cant use #define, #include and #ifdef statements.
  - Java does not include structures, unions and enum data types.
  - Java does not include keywords like goto, sizeof and typedef.
  - Java adds labeled break and continue statements.
  - Java adds many features required for object oriented programming.

# How is Java different from C++...

- **C++ language**

  Features removed in java:

  - Java doesn't support **pointers** to avoid **unauthorized** access of **memory locations**.
  - Java does not include structures, unions and enum data types.
  - Java does not support **operator over loading**.
  - Preprocessor plays less important role in C++ and so **eliminated** entirely in java.
  - Java does not perform **automatic** type conversions that result in loss of **precision**.

# How is Java different from C++...

## New features added in Java:

- **Multithreading**, that allows two or more pieces of the same program to execute concurrently.
- C++ has a set of library functions that use a common header file. But java replaces it with its own set of **API classes.**
- It adds **packages** and **interfaces**.
- Java supports automatic **garbage collection**.
- **break** and **continue** statements have been enhanced in java to accept labels as targets.
- The use of **unicode** characters ensures portability.

# How is Java different from C++...

- Java does not support **global variables**. Every method and variable is declared within a **class** and forms part of that class.
- Java does not allow **default arguments**.
- Java does not support inheritance of **multiple** super classes by a sub class (i.e., **multiple inheritance**). This is accomplished by using '**interface**' concept.
- It is not possible to declare **unsigned integers** in java.
- In java objects are passed by **reference** only. In C++ objects may be passed by **value** or **reference**.

# How is Java different from C++...

## Features that differ:

- Though **C++** and **java** supports Boolean data type, C++ takes any **nonzero value** as true and **zero as** false. True and false in java are predefined literals that are values for a Boolean expression.
- Java has replaced the **destructor** function with a **finalize()** function.
- C++ supports exception handling that is similar to java's. However, in C++ there is no requirement that a thrown exception be caught.

# C++ vs Java

| Comparison Index | C++ | Java |
|---|---|---|
| Platform-independent | C++ is platform-dependent. | Java is platform-independent. |
| Mainly used for | C++ is mainly used for system programming. | Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications. |
| Goto | C++ supports goto statement. | Java doesn't support goto statement. |

# C++ vs Java

| Comparison Index | C++ | Java |
|---|---|---|
| Compiler and Interpreter | C++ uses compiler only. | Java uses compiler and interpreter both. |
| Call by Value and Call by reference | C++ supports both call by value and call by reference. | Java supports call by value only. There is no call by reference in java. |
| Structure and Union | C++ supports structures and unions. | Java doesn't support structures and unions. |

# C++ vs Java

| Comparison Index | C++ | Java |
|---|---|---|
| Thread Support | C++ doesn't have built-in support for threads. | Java has built-in thread support. |
| Documentation comment | C++ doesn't support documentation comment. | Java supports documentation comment (/** ... */) to create documentation for java source code. |
| unsigned right shift >>> | C++ doesn't support >>> operator | Java supports unsigned right shift >>> operator |

# Basic of Java

- Java is a platform independent, more powerful, secure, high performance, multithreaded programming language. Here we discuss some points related to java.
- **Define byte**
- **Byte code** is the set of optimized instructions generated during compilation phase and it is more powerful than ordinary pointer code.
- **Define JRE**
- The **Java Runtime Environment (JRE)** is part of the Java Development Kit (JDK). It contains set of libraries and tools for developing java application. The Java Runtime Environment provides the minimum requirements for executing a Java application.

# Basic of Java

- **Define JVM**
- **JVM** is set of programs developed by sun Micro System and supplied as a part of jdk for reading line by line of byte code and it converts into native understanding form of operating system. Java language is one of the compiled and interpreted programming language.
- **Garbage Collector**
- **Garbage Collector** is the system Java program which runs in the background along with regular Java program to collect un-Referenced (unused) memory space for improving the performance of our applications.

# Basic of Java

- **What is JVM?**
- **It is:**
- **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- **An implementation** Its implementation is known as JRE (Java Runtime Environment).
- **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

# Basic of Java

- **What it does**
- The JVM performs following operation:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment
- JVM provides definitions for the:
  - Memory area
  - Class file format
  - Register set
  - Garbage-collected heap
  - Fatal error reporting etc.

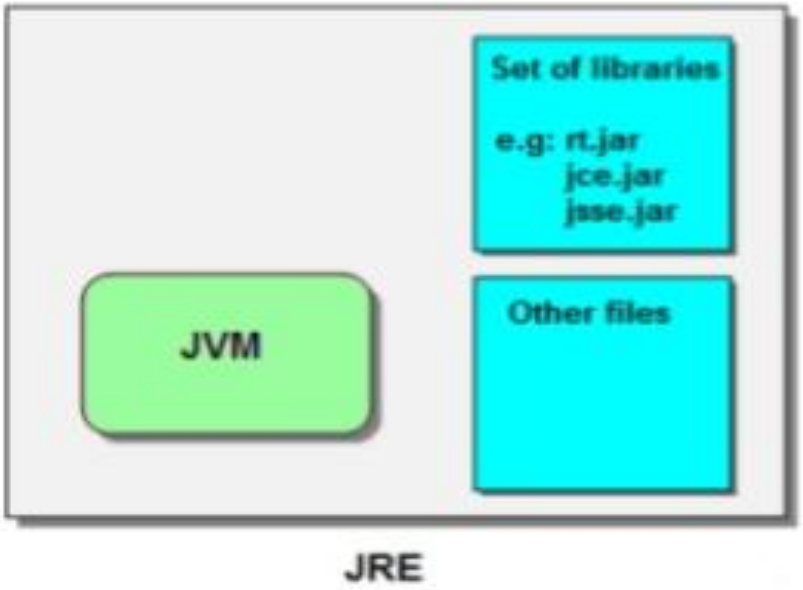
# Difference between JDK, JRE and JVM

- **JVM**

- JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.
- JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.
- The JVM performs following main tasks:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment

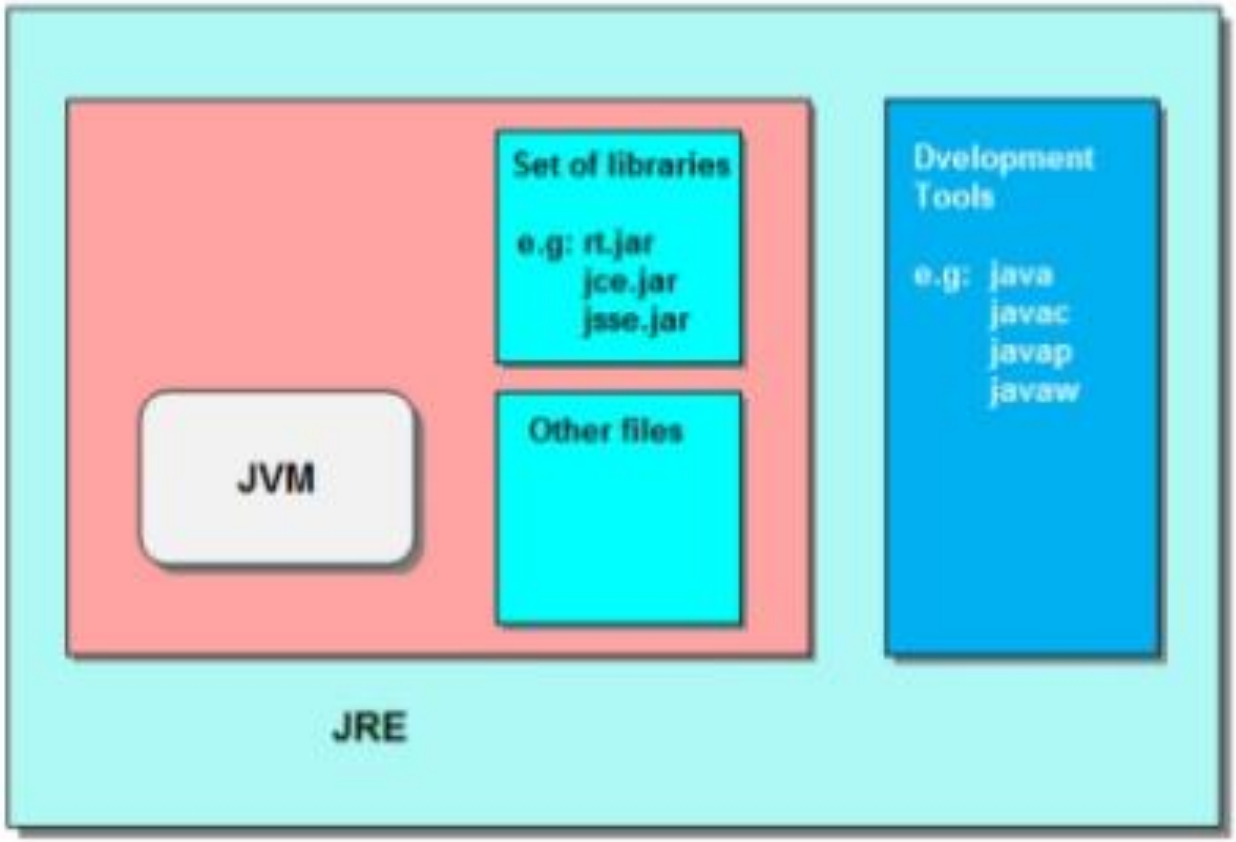
# Basic of Java

- **JRE**
- JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.



# Basic of Java

- The Java Development Kit (JDK) is primary components. It physically exists. It is collection of programming tools and JRE, JVM.

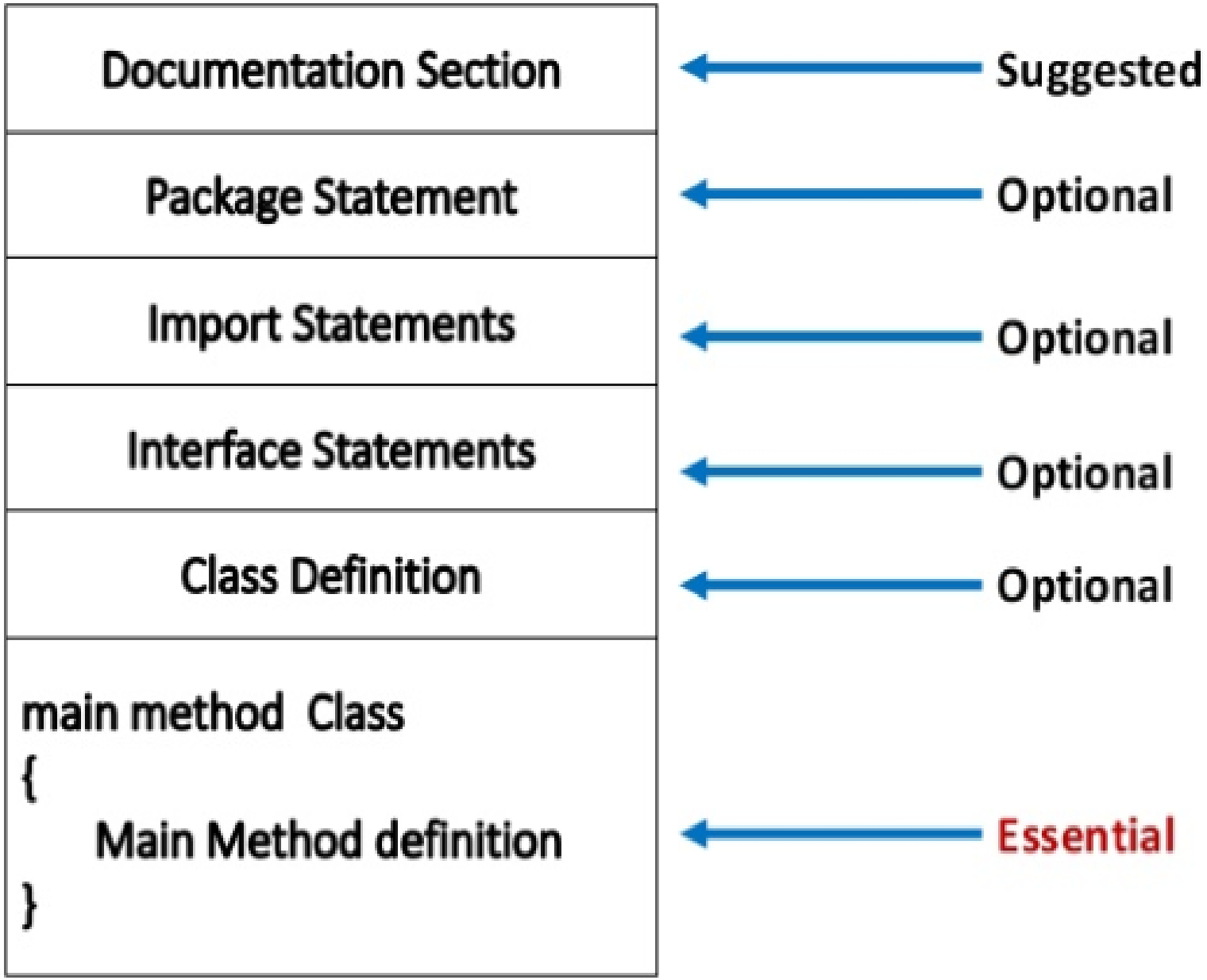


| Aspects | C | C++ | Java |
|---|---|---|---|
| Developed Year | 1972 | 1979 | 1991 |
| Developed By | Dennis Ritchie | Bjarne Stroustrup | James Gosling |
| Successor of | BCPL | C | C(Syntax) & C++ (Structure) |
| Paradigms | Procedural | Object Oriented | Object Oriented |
| Platform Dependency |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

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

# Java Program Structure

- As we seen in the previous slides first program example, a Java program may contain many classes of which only one class define a main method.
- Classes contain data members and methods that operate on the data members of the class. Method may contain data type declarations and executable statements.
- To wire a java program we first define classes and put them together.
- A java Program may contain one or more sections as show in fig in next slide.


Documentation Section
Suggested
Package Statement
Optional
Import Statements
Optional
Interface Statements
Optional
Class Definition
Optional
main method Class
{
Main Method definition
}
Essential

# Java Program Structure

- **Documentation Section**
- The documentation section comprises a set of comment lines giving the name of the program, the author and detail, which the programmer would like to refer to at a later stage.
- Comments must explain why and what a classes and how algorithms.
- This would greatly help in maintaining the program.
- /**……*/ know as documentation comments.
- This form of comment is used to generating documentation.

# Java Program Structure

- **Package Statement**
- The package statement allowed in a Java file is a package statement .
- This statement declares a package name and informs the compiler that the classes defined here belong to this package.
- **Example** **package** myfirstprogram;
- The package statement is optional.
- That is, our classes do not have to be part of package.
- More about packages we learn later in detail .

# Java Program Structure

- **Import Statement**
- The next thing after a package statement (but before any class definition) may be a number of import statements.
- This is similar to the #include statement in C/C++.
- **Example** **import** Math.min;
- This statement instruct the interpreter to load the test class contained in the package student.
- Using import statements, we can have access to classes that are part of other packages,
- More about import statement we learn in upcoming lecture

# Java Program Structure

- **Interface statements**
- An interface is like a class but includes group of method declaration.
- This is also an optional section and is used only when we wish to implement the multiple inheritance feature in the program.
- Interface is a new concept in java and we discussed in detail in upcoming lectures.

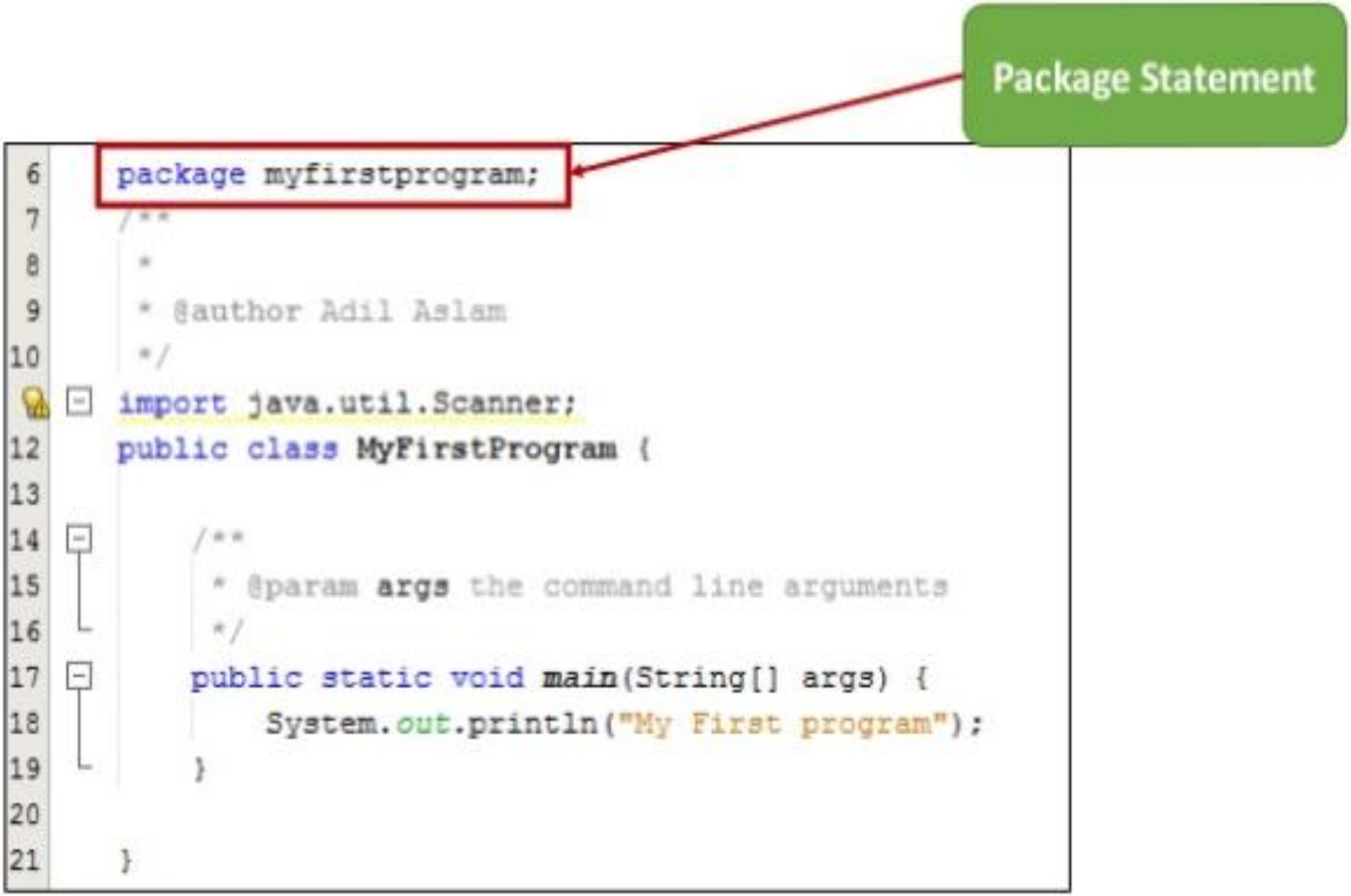
## Java Program Structure

- **Class Definition**
- A Java program may contain multiple class definition.
- Classes are the primary and essential part of a Java program.
- These classes are used to map the objects of real-world problem.
- The number of classes used depends on the complexity of the problem.

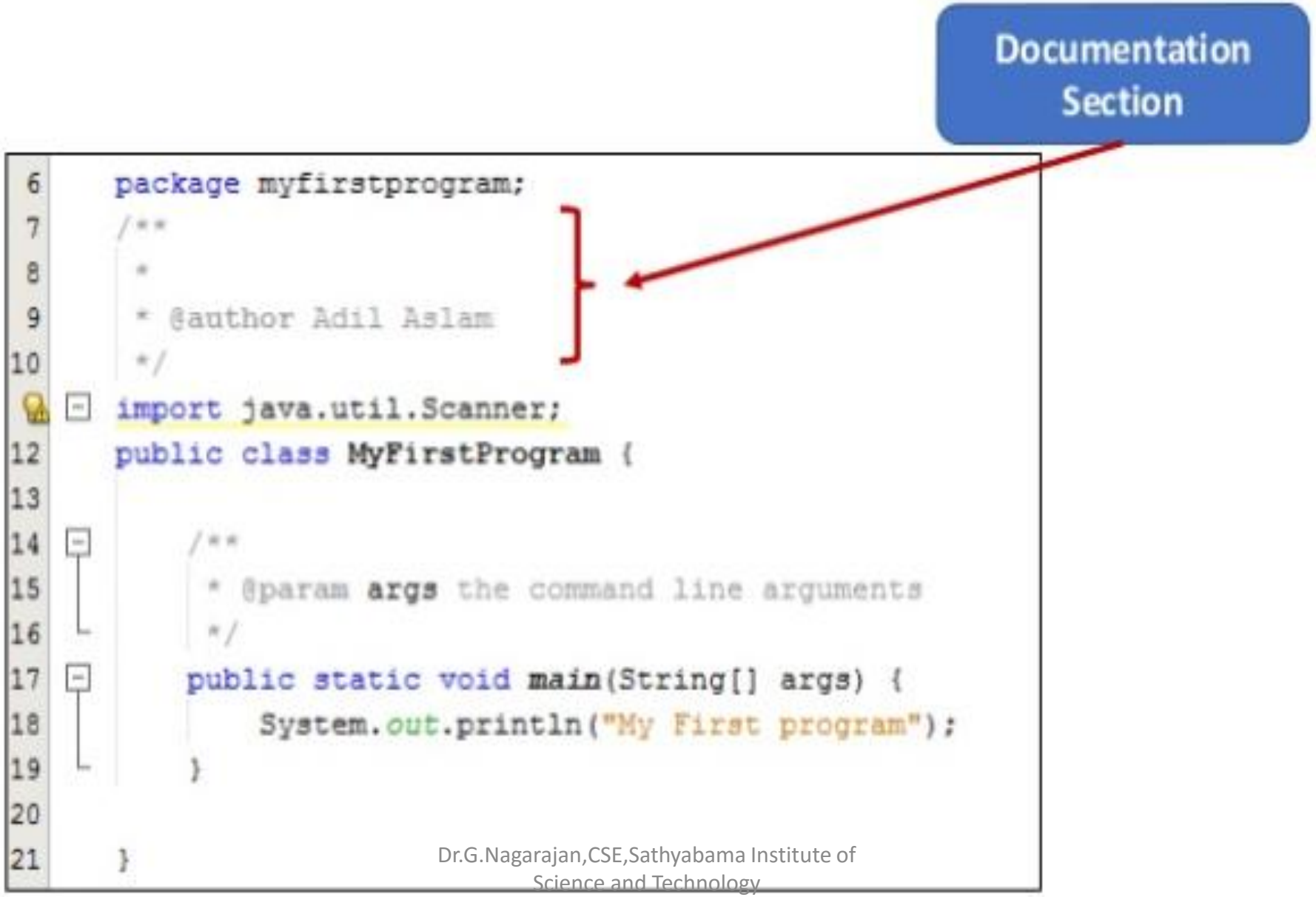
## Java Program Structure

- **Main Method Class**
- Since every Java stand-alone program require a main method as its starting point, this class is the essential part of a java program.
- A simple Java program may contain only this part.
- The main method create object of various classes and establish communication between them.
- On reaching end of main the program terminates and the control passes back to the operating system (OS).

# Java Program Structure



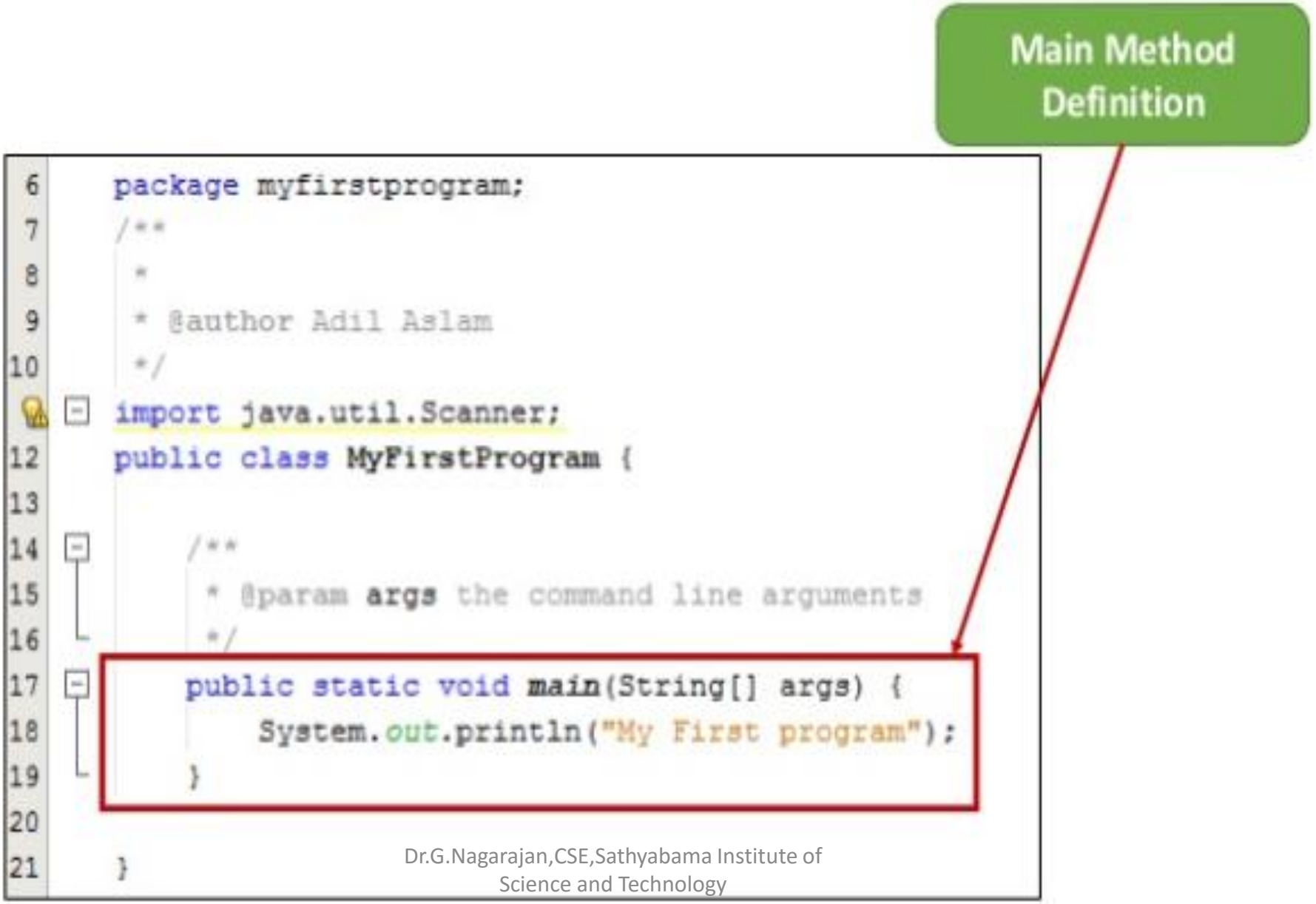
# Java Program Structure



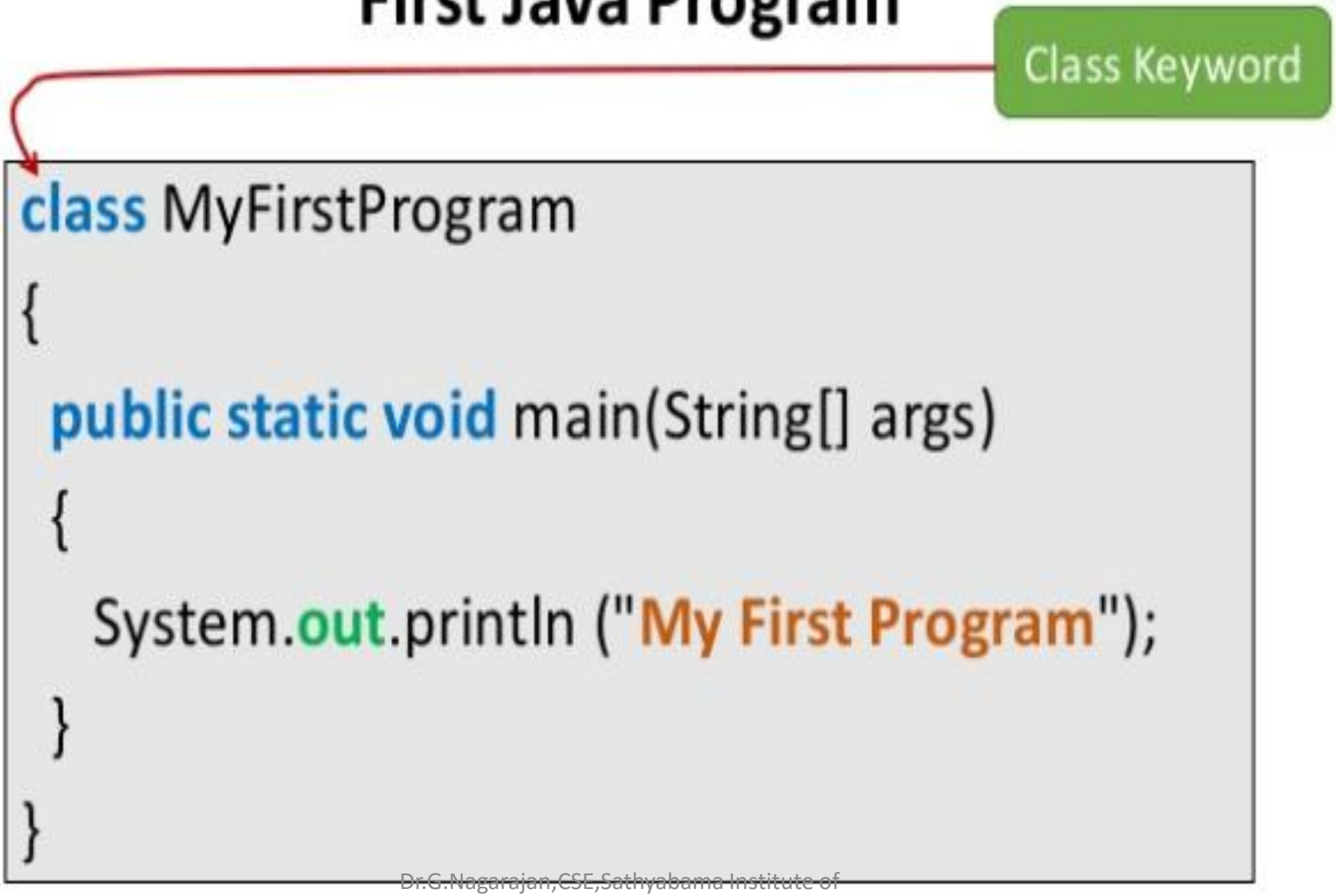
# Java Program Structure

# Java Program Structure

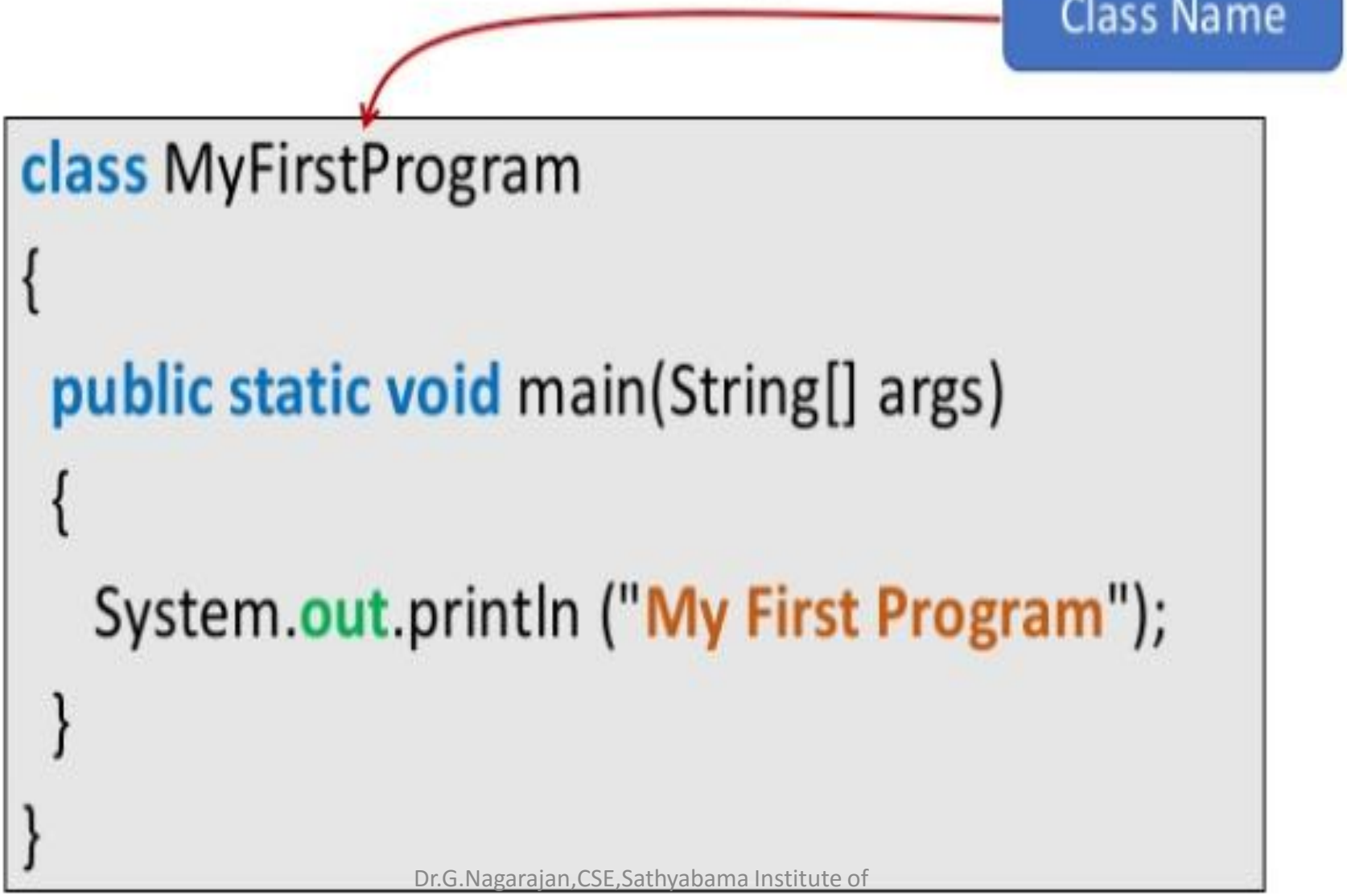
# Java Program Structure



# First Java Program



# First Java Program

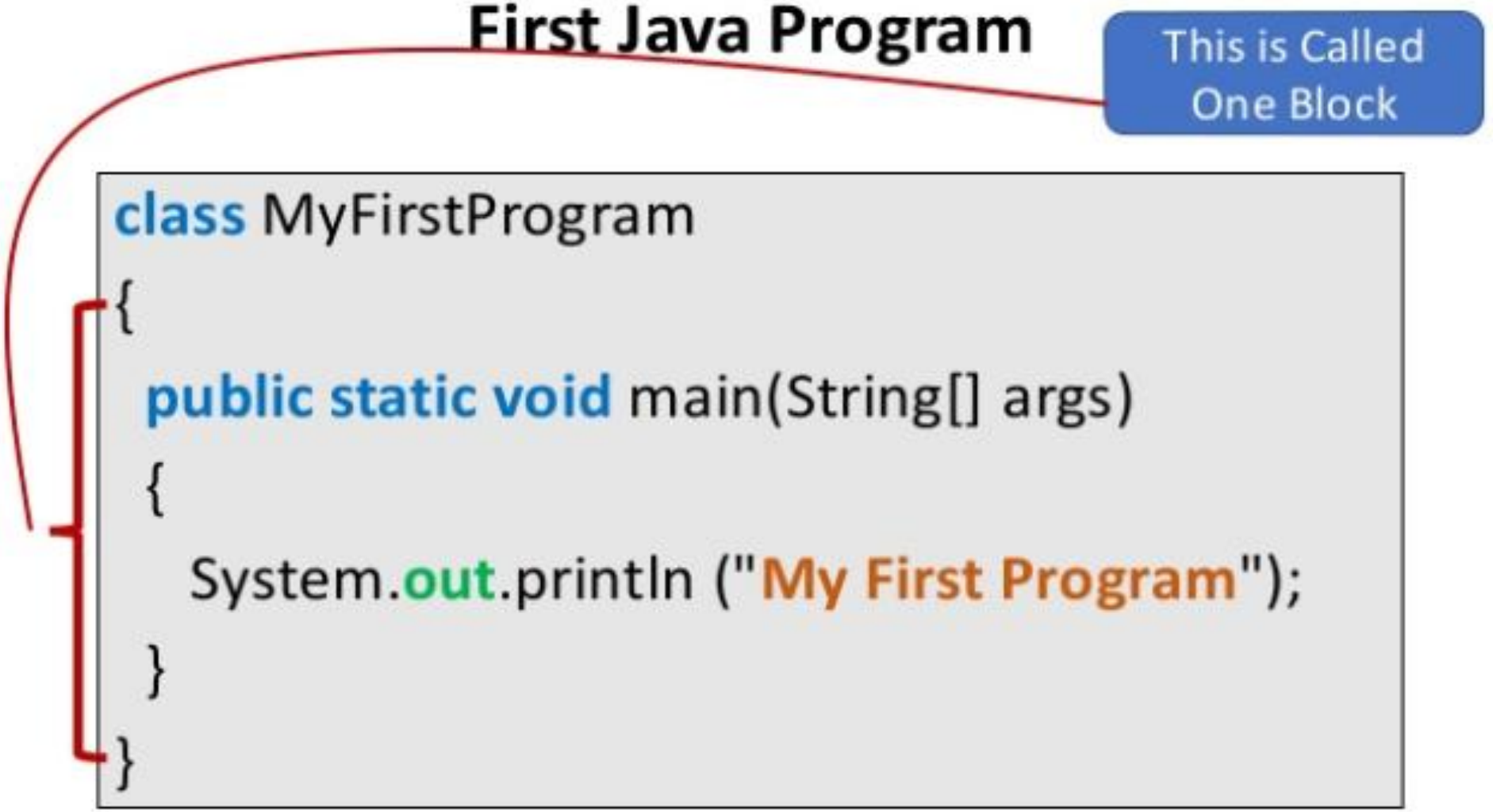


# First Java Program

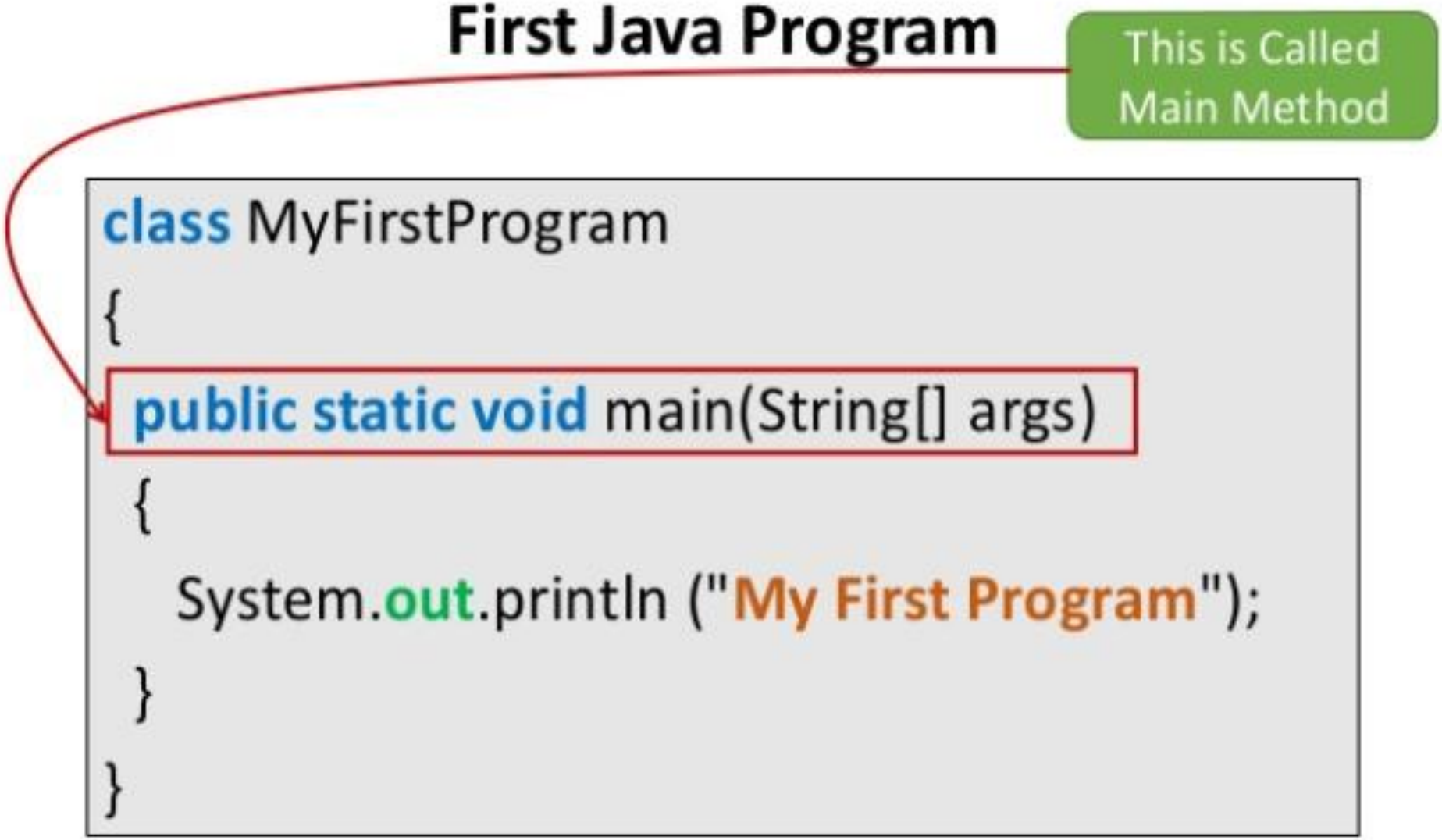
Starting of Class Level Scope

```
class MyFirstProgram
{
  public static void main(String[] args)
  {
    System.out.println ("My First Program");
  }
}
```

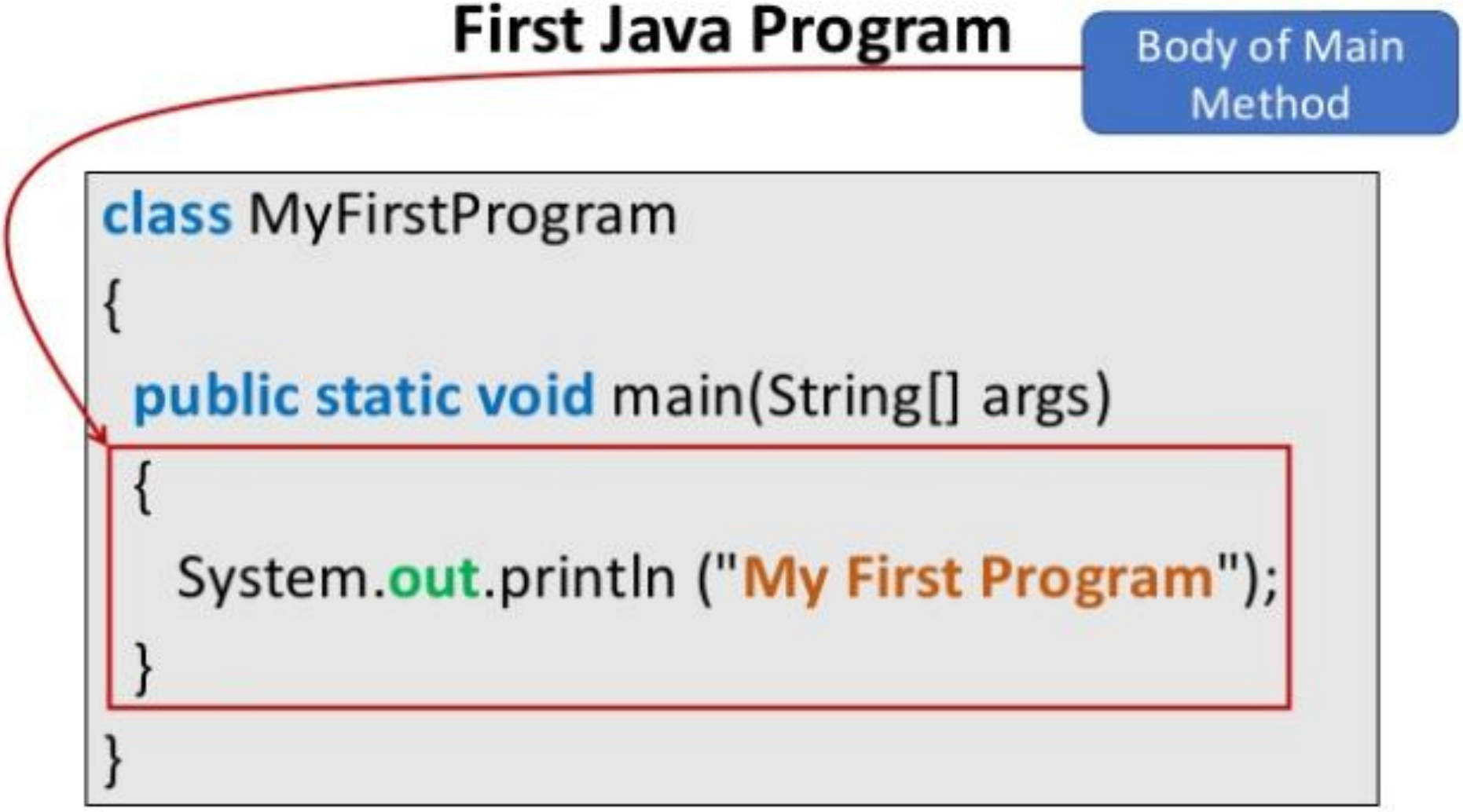
# First Java Program



# First Java Program



# First Java Program



# First Java Program

Semicolon use to terminate the statement.

```
class MyFirstProgram
{
  public static void main(String[] args)
  {
    System.out.println ("My First Program");
  }
}
```

# Blocks in Java

- A *block* (or a ***compound statement***) is a group of statements surrounded by **braces** { }.
- **All the statements** inside the block is **treated as one unit**. Blocks are used as the *body* in constructs like function, if-else and loop, which may contain multiple statements but are treated as one unit.
- **For example**

```
public static void main(String[] args)
{
   System.out.println ("My First Program");
}
```

This is one block

- **class** – is use to declare a class in java.
- **public** – is a access modifies which represent visibility. public means it is visible to everyone.
- **static** – is a keyword. If we any method as static , it is known as static method. The advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it does not require to create object to invoke the main method. So it saves memory.
- **void** – is the return type of the method. void means method does not return any value.
- **main** – represent startup of the program.
- **String args[]** – is used for command line argument. we will learn it later.
- **System.out.println()** – is used to print statement.

# First Java Program

- **class** – is use to declare a class in java.
- **public** – is a access modifies which represent visibility. public means it is visible to everyone.
- **static** – is a keyword. If we any method as static , it is known as static method. The advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it does not require to create object to invoke the main method. So it saves memory.

# First Java Program

- **void** – is the return type of the method. void means method does not return any value.
- **main()** : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.
- **String args[]** – is used for command line argument. We will learn it later.
- **System.out.println()** – is used to print statement. We will learn about the internal working of System.out.println statement later.

# Explanation of main Method

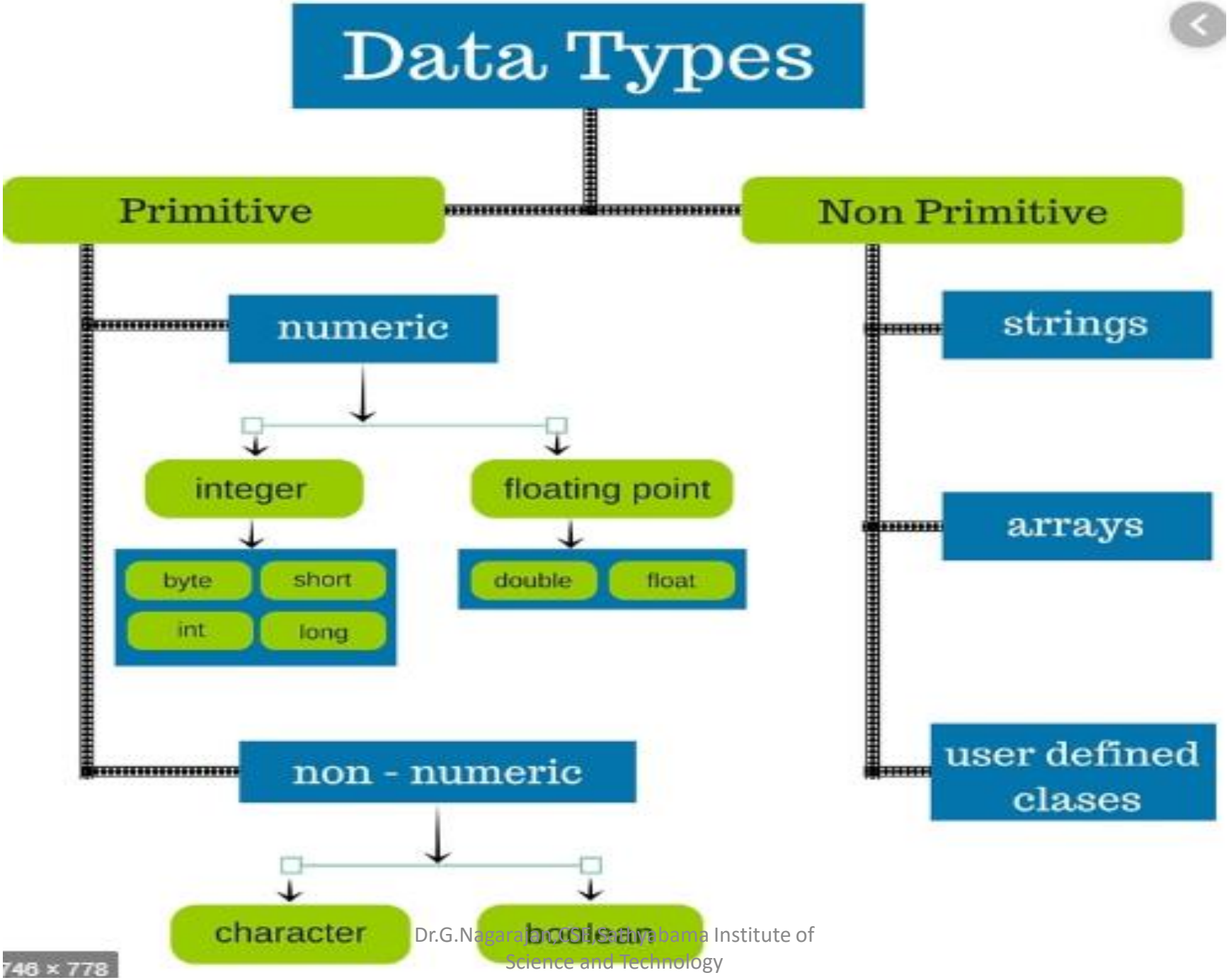
```
public static void main( String[] args )
{

}
```

"**public**" means that main() can be called from anywhere.
"**static**" means that main() doesn't belong to a specific object
"**void**" means that main() returns no value
"**main**" is the name of a function. main() is special because it is the start of the program.
"**String[]**" means an array of String.
**String[] args"** is a single parameter for the method. **String[]** is the type of the parameter, indicating an array of Strings.
**"args"** is the name of the parameter. Parameters must be named. "args" is not special; you could name it anything else and the program would work the same.

# Explanation of Print Statement

**system.out.println()**

- Java comes up with the pre-defined class to let us print strings or variables in the screen. The pre-defined or built-in class is the ***system class*** that provides a few useful methods and variables.
- The whole line ***system.out.println*** is explained below:
- **System**: System is the class provided by Java that contains variables and methods.
- **Out**: Out is the system's static variable.
- **Println**: *Println* is the method of **system class** that is used to print the given text or variable etc.


Data Types
Primitive
Non Primitive
numeric
strings
integer
floating point
arrays
byte
short
int
long
double
float
non - numeric
user defined clases
character
boolean

# Object and class in Java

- Object is the physical as well as logical entity where as class is the only logical entity.
- **Class:** Class is a blue print which is containing only list of variables and method and no memory is allocated for them. A class is a group of objects that has common properties.
- A class in java contains:
- Data Member
- Method
- Constructor
- Block
- Class and Interface

# Object and class in Java

- **Object:** Object is a instance of class, object has state and behaviors.
- An Object in java has three characteristics:
- State
- Behavior
- Identity
- **State:** Represents data (value) of an object.
- **Behavior:** Represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **Identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.
- Class is also can be used to achieve user defined data types.

# Real life Example of object and class

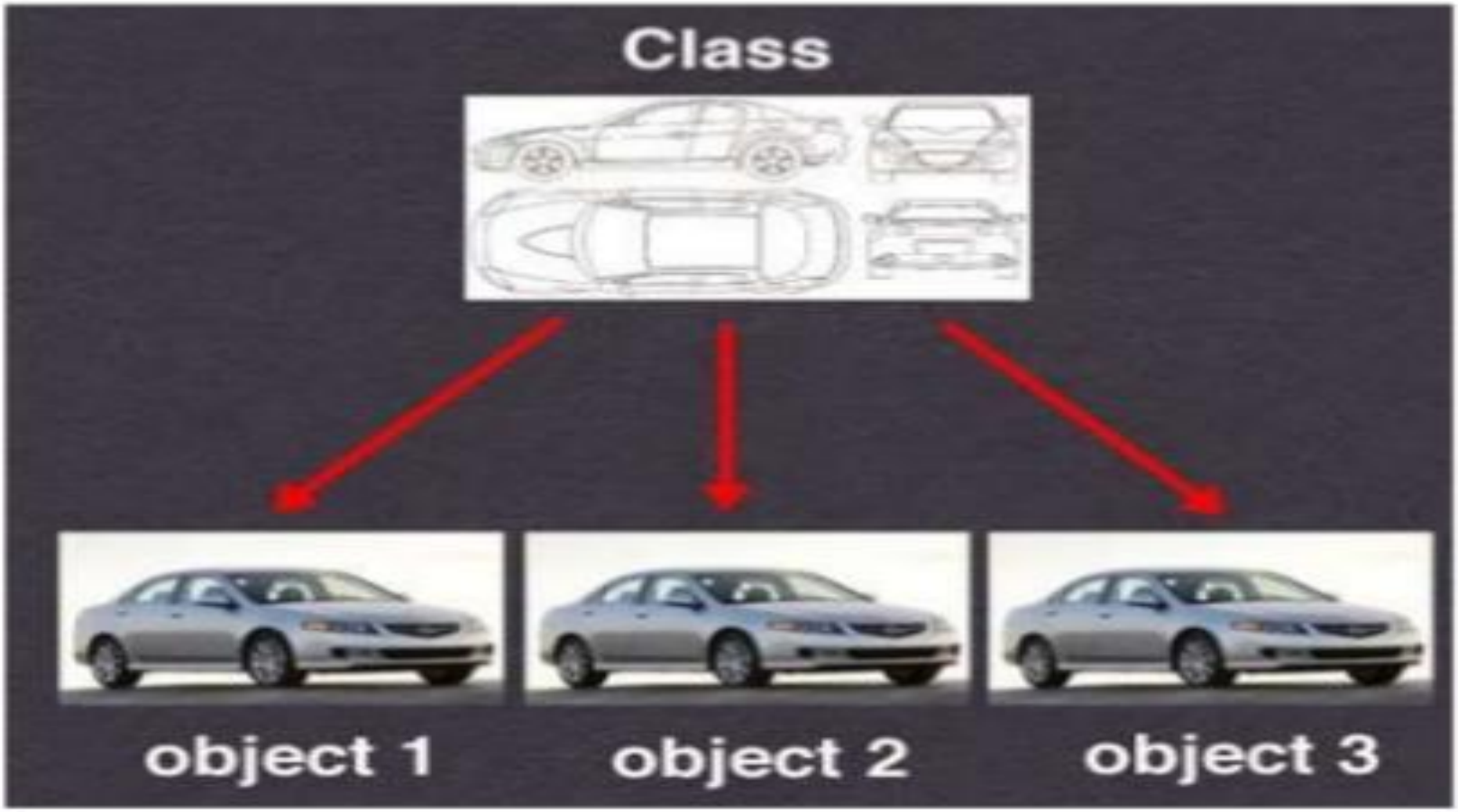
- In real world many examples of object and class like dog, cat, and cow are belong to animal's class. Each object has state and behaviors.
- For example a dog has **state:** color, name, height, age as well as **behaviors:** barking, eating, and sleeping.

# Real life Example of object and class

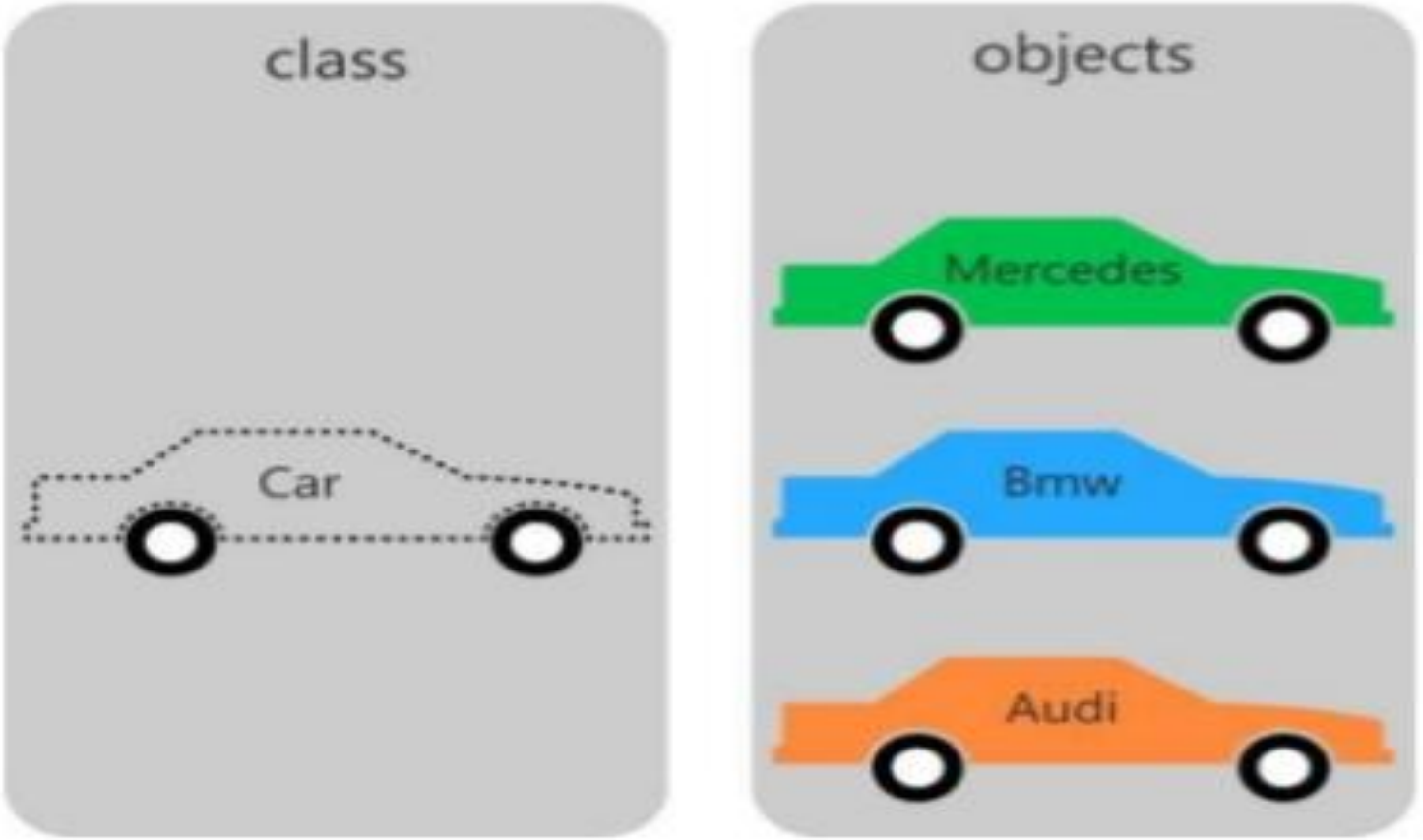
- **Vehicle class**
- Car, bike, truck these all are belongs to vehicle class. These Objects have also different states and behaviors. For Example car has **state:** color, name, model, speed, Mileage. And **behaviors**: distance travel

Vechicle class

# Object and Class in Java



# Object and Class in Java



# Difference between Class and Object

| Class | Object | |
|---|---|---|
| 1 | Class is a container which collection of variables and methods. | object is a instance of class |
| 2 | No memory is allocated at the time of declaration | Sufficient memory space will be allocated for all the variables of class at the time of declaration. |
| 3 | One class definition should exist only once in the program. | For one class multiple objects can be created. |

# Class in Java

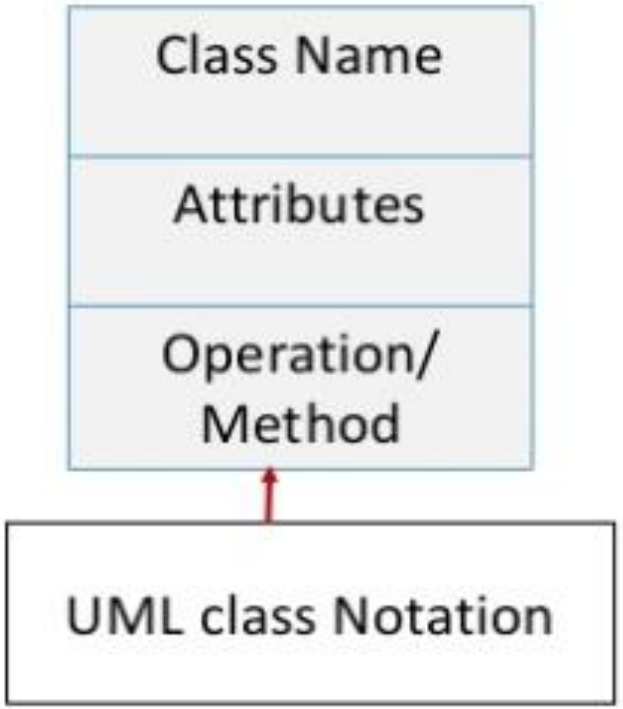
- In Java everything is encapsulated under classes. Class is the core of Java language.
- Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type can be used to create object of that type.
- Object is an instance of class. You may also call it as physical existence of a logical template class.
- A class is declared using **class** keyword.
- A class contain both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.

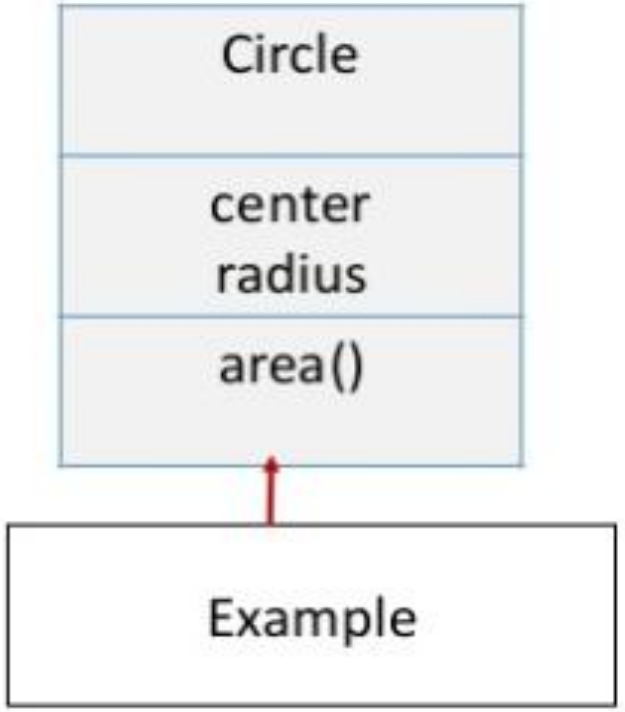
# Rules for Java Class

- A class can have only public or default(no modifier) access specifier.
- It can be either abstract, final or concrete (normal class).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces {}.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

# Class in Java

- A ***class*** is a collection of ***fields*** (data) and ***methods*** (procedure or ,functions) that operate on that data.



# Java Naming Conventions

- Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.
- But, it is not forced to follow. So, it is known as convention not rule.
- All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.
- Advantage of naming conventions in java
  - By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

# Java Naming Conventions

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

## Class Declaration

- Declaration of class must start with the **keyword** ***class*** followed by the class name and class members are declared within braces.
- **Syntax of declaring class**

```
class class_name
  {
  // some data/some fields
  // some functions/methods
  }
```

Note: A class is a user defined data type.

# Class Declaration

- Declaration of class must start with the **keyword *class*** followed by the class name and class members are declared within braces.
- **Syntax of declaring class (general)**

```
access specifier class class_name
  {
  // some data/some fields
  // some functions/methods
  }
```

## Syntax of Declaring Class

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

# Class Explanation Of Syntax

- **Class Name**

```
access specifier class classname
{
}
```

- **Access Specifier : Access** modifiers (or **access specifiers**) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.
- **Access Specifier:** is Optional (public. Private, protected)
- **class** is Keyword in Java used to create class in java.
- classname is Name of the User defined Class.

# Class Explanation Of Syntax

- **Class Naming Example**

```
public class Rectangle
{
}
```


- **Access Specifier : Access** modifiers (or **access specifiers**) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.
- **Access Specifier:** is Optional (public. Private, protected)
- **class** is Keyword in Java used to create class in java.
- classname is Name of the User defined Class.

# Class Explanation Of Syntax

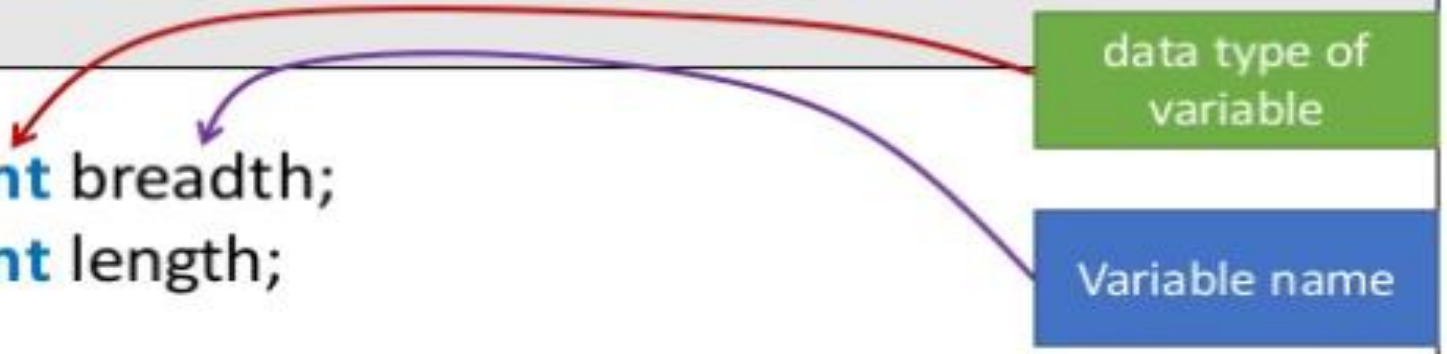
## • Class Instance Variable

```
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
```

- Instance Variables are Class Variables of the class.
- When a number of objects are created for the same class, **the same copy of instance variable is provided to all**.
- **Instance variables have different value** for different objects.
- **Access Specifiers** can be applied to instance variable i.e public, private.
- Instance Variable are also called as "**Fields**"

# Class Explanation Of Syntax

- **Class Instance Variable Example**




- Instance Variables are Class Variables of the class.
- When a number of objects are created for the same class, **the same copy of instance variable is provided to all**.
- **Instance variables have different value** for different objects.
- **Access Specifiers** can be applied to instance variable i.e public, private.
- Instance Variable are also called as "**Fields**"

# Class Explanation Of Syntax

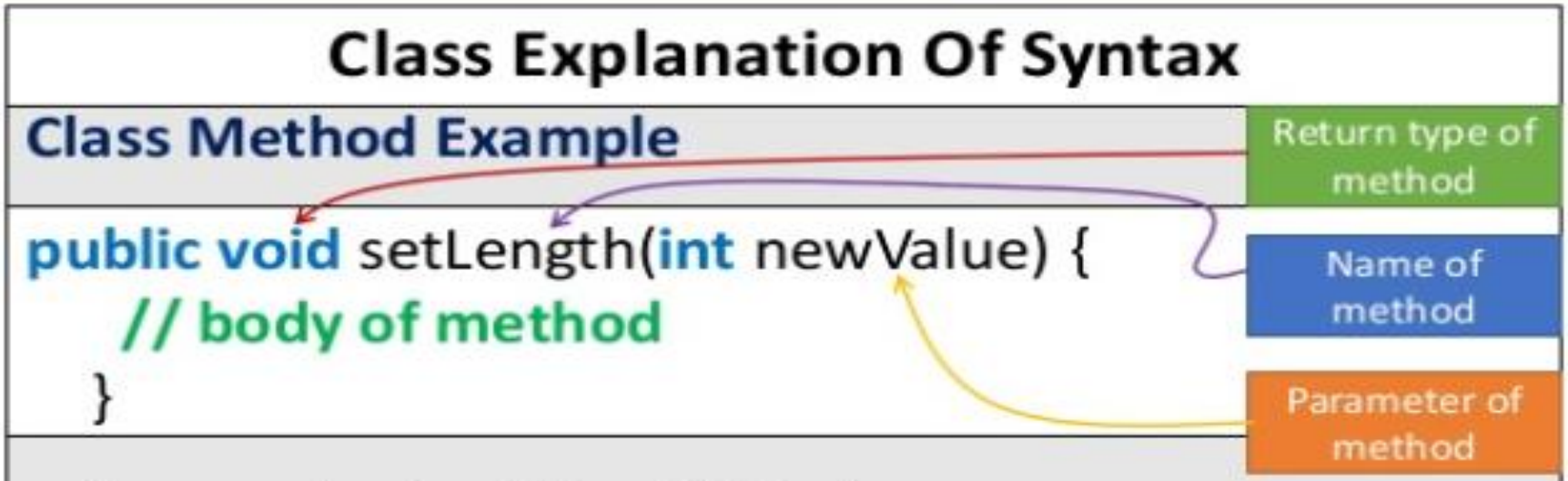
## Class Method

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

- Above syntax is of **Class Methods**.
- These methods are equivalent to function in C Programming Language.
- Class methods **can be declared public or private**.
- These methods are meant for operating on class data i.e **Class Instance Variables**.
- Methods have **return type as well as parameter list**.

# Class Explanation Of Syntax

## Class Method Example




- Above syntax is of **Class Methods**.
- These methods are equivalent to function in C Programming Language.
- Class methods **can be declared public or private**.
- These methods are meant for operating on class data i.e **Class Instance Variables**.
- Methods have **return type as well as parameter list**.

## Complete Example of Class in Java

```
public class Rectangle {
   // two fields
   public int breadth;
   public int length;

   // two methods
   public void setLength(int newValue) {
      length = newValue;
   }
   public void setBreadth(int newValue) {
      breadth = newValue;
   }
}
```

Opening curly brace of the class

Two Instance Variables

closing curly brace of the class

# Syntax of declaring class

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

# Simple Example of Object and Class

```
class Employee
{
 int eid;                  // data member (or instance variable)
 String ename;             // data member (or instance variable)
 eid=101;                  //Assign value to instance variable
 ename="Adil";             //Assign value to instance variable
 public static void main(String args[])
 {
  Employee e=new Employee();  // Creating an object of class Employee
  System.out.println("Employee ID: "+e.eid);
  System.out.println("Name: "+e.ename);
 }
}
```

**Output is:**
Employee ID: 101
Name: Adil

# Ways to Create Objects in Java?

There are five different ways to create objects in java:

- Using new keyword
- Using Class.forName():
- Using clone():
- Using Object Deserialization:
- Using newIntance() method

# Creating an Object (using new keyword)

- As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.
- There are three steps when creating an object from a class :
  - **Declaration** – A variable declaration with a variable name with an object type.
  - **Instantiation** – The 'new' keyword is used to create the object.
  - **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

# Ways to Create Objects in Java?

- **Using new keyword:**
- This is the most common way to create an object in java. Almost 99% of objects are created in this way.

- **Syntax:**

```
class_name object_name = new class_name();
```

- **Example**

```
Rectangle obj = new Rectangle();
```

# Ways to Create Objects in Java?

- **Using new keyword:**

```
Rectangle obj = new Rectangle();
```

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

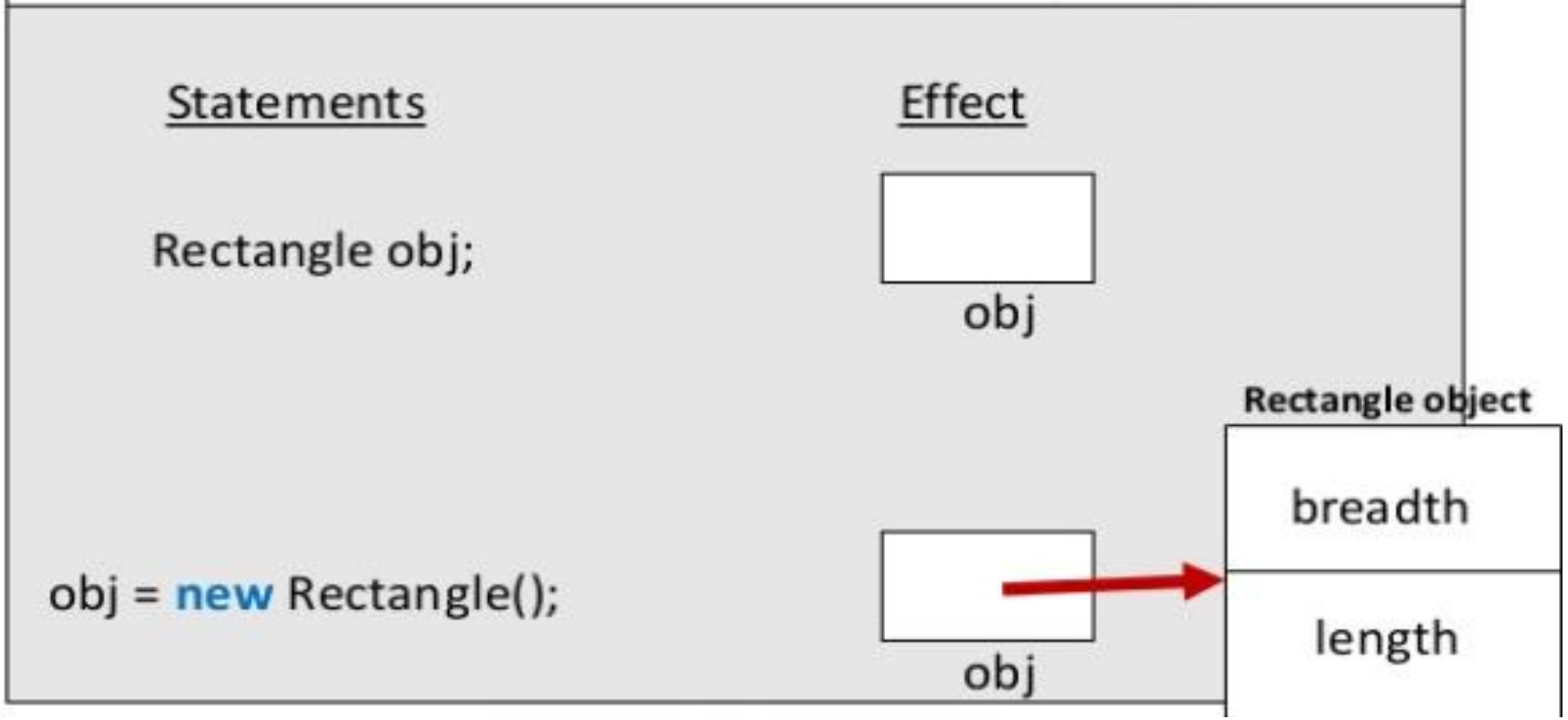
```
Rectangle obj;    // declare reference to object
obj = new Rectangle();    // allocate a Rectangle object
```

- The first line declares **obj** as a reference to an object of type **Rectangle**. At this point, **obj** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **obj**. After the second line executes, you can use **obj** as if it were a **Rectangle** object. But in reality, **obj** simply holds, in essence, the memory address of the actual **Rectangle** object.

# Ways to Create Objects in Java?

- **Using new keyword:**

```
Rectangle obj;    // declare reference to object
obj = new Rectangle();    // allocate a Rectangle object
```



# Ways to Create Objects in Java?

- **Using new keyword:**
- This is the most common way to create an object in java. Almost 99% of objects are created in this way.
- **Syntax:**

```
class_name object_name = new class_name();
```

- **Example**

Now allocate memory to object (obj)

```
Rectangle obj = new Rectangle();
```

# Object Creation in Java

```
class Student.
{
 String name;
 int rollno;
 int age;
}
```

- When a reference is made to a particular student with its property then it becomes an **object**, physical existence of Student class.

```
Student std = new Student();
```

- After the above statement std is instance/object of Student class. Here the new keyword creates an actual physical copy of the object and assign it to the std variable. It will have physical existence and get memory in heap area. The new operator dynamically allocates memory for an object

## Accessing Instance Variables and Methods

- Now that we have created object, each containing its own set of variables, we should assign values to these variables in order to use in our program.
- Remember, all variables must be assigned values before they are used. Since if we are outside the class, we can't access the instance variables and methods directly.
- To do this we must use concerned object and dot operator shown in next slide.

## Accessing Instance Variables and Methods

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

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

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

/* Now you can call a class method as follows */
ObjectReference.MethodName(ParameterList);
```

# Accessing Instance Variables and Methods

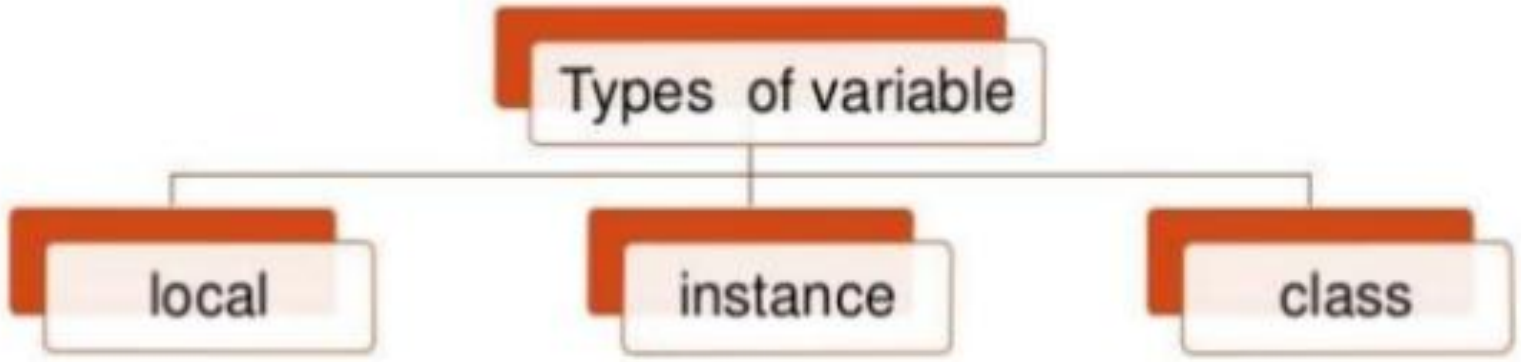
```
/* Now call a variable as follows */
ObjectReference.variableName;
/* Now you can call a class method as follows */
ObjectReference.MethodName(ParameterList);
```

- Here ObjectReference is the name of the object and variableName is the name of the instance variable inside the object that we wish to access.
- MethodName is the name of the method that we wish to call, and ParameterList is a comma separated list "actual values" or (expressions) that must in type and number with the parameter of the methodname declared in the class.

# Accessing Instance Variables and Methods

```
public class Example {
   int myCount = 0;
   void increment ( ) {
      myCount = myCount + 1;  }
    void print ( ) {
      System.out.println ("count = " + myCount);  }
   public static void main(String[] args) {
    Example c1 = new Example ( );
      c1.increment ( );  // c1's myCount is now 1
      c1.increment ( );  // c1's myCount is now 2
      c1.print();
      c1.myCount = 0;    // effectively resets the c1 counter
      c1.print();
   }
}
```

# Variable Types



# Variable Types

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

# Local Variables Example

```
public class Test{
   public void age() {
      int age = 0 ;  //initializing with 0
      age = age + 7;
      System.out.println("Age is : " + age);
   }
public static void main(String[] args) {
    Test test = new Test();  //Creating an object
    test.age();
  }
}
```

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

Calling age method with the reference of test object

# Local Variables Example(some change)

```
public class Test{
  public void age() {
    int age ;
    age = age + 7;
    System.out.println("Age is : " + age);
 }
public static void main(String[] args) {
   Test test = new Test();
   test.age();
  }
}
```

Same program as previous but in this program we use local variable *age* without initializing it, so it would give an error at the time of compilation.

**Compiler Error**

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

# Instance Variables-1

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.

# Instance Variables-2

- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class.

# Instance Variables

```
class Rectangle {
 //instance variables
  double length;
  double breadth;
// This class declares an object of type Rectangle.
  public static void main(String args[]) {
    Example myrect = new Example();
    double area;
    // assign values to myrect1's instance variables
    myrect.length = 10;
    myrect.breadth = 10;
    // Compute Area of Rectangle
    area =  myrect.length * myrect.breadth ;
    System.out.println("Area of Rectangle  : " + area);
  }
}
```

Instance variables

Using dot Operator we can access instance variable of object.

**Output is:**
Area of Rectangle  : 100.0s:

# Class/Static Variables-1

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

# Class/Static Variables-2

- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name*ClassName.VariableName*.

## Static Variables Example

```
public class Employee {
  // salary  variable is a private static variable
  private static double salary;
  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";
  public static void main(String args[]) {
    salary = 1000;
    System.out.println(DEPARTMENT + "average salary:" + salary);
  }
}
```

**Output is:**

Development average salary:1000.0

## Summary

| Characteristic | Local variable | Instance variable | Class variable |
|---|---|---|---|
| *Where declared* | In a method, constructor, or block. | In a class, but outside a method. Typically private. | In a class, but outside a method. Must be declared static. Typically also final. |
| *Use* | Local variables hold values used in computations in a method. | Instance variables hold values that must be referenced by more than one method | Class variables are mostly used for *constants*, variables that never change from their initial value |
| *Lifetime* | Created when method or constructor is entered. Destroyed on exit. | Created when instance of class is created with new. Destroyed when there are no more references to enclosing object (made available for garbage collection). | Created when the program starts. Destroyed when the program stops. |

| Characteristic | Local variable | Instance variable | Class variable |
|---|---|---|---|
| *Declaration* | Declare before use anywhere in a method or block. | Declare anywhere at class level (before or after use). | Declare anywhere at class level with static. |
| *Initial value* | None. Must be assigned a value before the first use. | Zero for numbers, false for booleans, or null for object references. May be assigned value at declaration or in constructor. | Same as instance variable, and it addition can be assigned value in special static *initializer block*. |
| *Name syntax* | Standard rules. | Standard rules, but are often prefixed to clarify difference from local variables, eg withmy, m, or m_ (for member) as in myLength, or this as inthis.length. | static public final variables (constants) are all uppercase, otherwise normal naming conventions. |

| Characteristic | Local variable | Instance variable | Class variable |
|---|---|---|---|
| *Access from outside* | Impossible. Local variable names are known only within the method. | Instance variables should be declared private to promote information hiding, so should not be accessed from outside a class. However, in the few cases where there are accessed from outside the class, they must be qualified by an object (eg, myPoint.x). | Class variables are qualified with the class name (e.g. Color. BLUE). They can also be qualified with an object, but this is a deceptive style. |

# Constructor in Java

# Constructor in Java

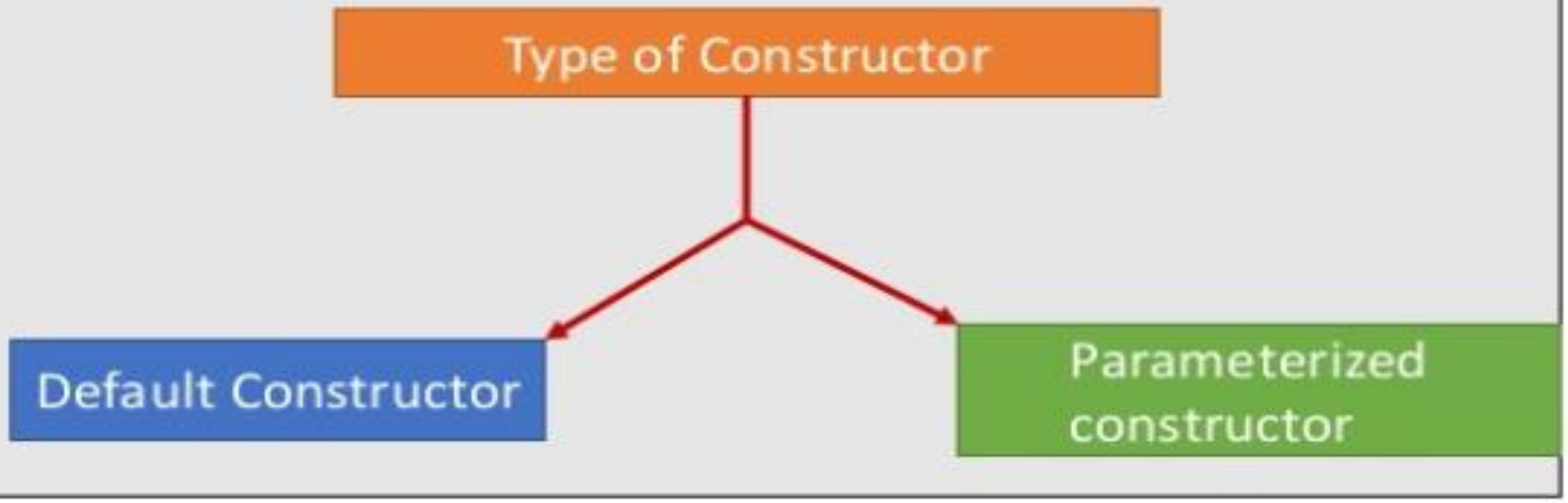
- **Constructor in java** is a *special type of method* that is used to initialize the object.
- Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.
- **Rules for creating java constructor**
- There are basically two rules defined for the constructor.
  - Constructor name must be same as its class name
  - Constructor must have no explicit return type

# Constructor in Java

- **Some Rules of Using Constructor :**
  - Constructor **Initializes an Object**.
  - Constructor **cannot be called** like methods.
  - Constructors **are called automatically** as soon as object gets created.
  - Constructor **don't have any return Type.** (even Void)
  - Constructor name is same as that of "**Class Name**".
  - Constructor **can accept parameter**.
  - Default constructor **automatically called** when object is created.

# Constructor in Java

- **Types of java constructors**
- There are two types of constructors:
  - Default constructor (no-arg constructor)
  - Parameterized constructor



# Constructor in Java

- **Java Default Constructor:**
- A constructor that have no parameter is known as default constructor.
- **Syntax of default constructor:**

```
class_name()
{
}
```

- **Example**

```
Rectangle()
{
}
```

default Constructor

# Example of default Constructor

```
public class Rectangle {
  Rectangle()
   {
    System.out.println("Rectangle is created");
   }
   public static void main(String[] args) {
     //Creating new object rec below
      Rectangle rec=new Rectangle();
   }
}
```

Class name and Constructor name are same

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

**Output is:**
Rectangle is created

# Another Example of default Constructor

```
public class Rectangle {
  int length;
  int breadth;
  Rectangle() {
  length  = 20;
  breadth = 10;
  }
  public static void main(String args[]) {
  Rectangle r1 = new Rectangle();
  System.out.println("Length of Rectangle : " + r1.length);
  System.out.println("Breadth of Rectangle : " + r1.breadth);
  }
}
```

Initializing the values of instance variables

**Output is:**

Length of Rectangle : 20

Breadth of Rectangle : 10

# Explanation of Previous Program

- **new** Operator will create an object.
- As soon as Object gets created it will call Constructor:

```
Rectangle() {    //This is default Constructor
 length  = 20;
 breadth = 10;
 }
```

- In the above Constructor Instance Variables of Object r1 gets their own values.
- Thus Constructor **Initializes an Object as soon as after creation**.
- It will print Values initialized by Constructor :

```
System.out.println("Length of Rectangle : " + r1.length);
System.out.println("Breadth of Rectangle : " + r1.breadth);
```

## Another Example of default Constructor-1

```
class Rectangle{
  int length;
  int breadth;

  Rectangle() {
  length  = 10;
  breadth = 10;
  }
  void setDimentions() {
  length  = 20;
  breadth = 20;
  }
```

Two instance Variables

default Constructor

Method setDimensions return nothing

# Default Constructor

- If there is no constructor in a class, compiler automatically creates a default constructor.
- **What is the purpose of default constructor?**
  - Default constructor provides the default values to the object like 0, null etc. depending on the type.

## Example of default Constructor that Displays the default Values

```
public class Example{
  int id;
  String name;
  void display() {
    System.out.println(id+"  "+name); }

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

**Output is:**

0 null

0 null

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

# Constructor in Java

- **Java parameterized constructor**
- A constructor that have parameters is known as parameterized constructor.
- **Why use parameterized constructor?**
  - Parameterized constructor is used to provide different values to the distinct objects.
- **Syntax of parameterized constructor**

```
class_name(parameter_list)
{
}
```

- **Example**

```
Rectangle (int len , int bre)
{
}
```

# Constructor in Java

- **Java parameterized constructor**
  - Constructor Can Take Value , Value is Called as – "**Argument**".
  - Argument **can be of any type** i.e Integer, Character, Array or any Object.
  - Constructor **can take any number of Argument**.

## Example of Parameterized Constructor

```
class Rectangle {
  int length;
  int breadth;
  Rectangle(int len , int bre)
  {
  length  = len;
  breadth = bre;
  }
  public static void main(String args[]) {
  Rectangle r1 = new Rectangle(20,10);
  System.out.println("Length of Rectangle: " + r1.length);
  System.out.println("Breadth of Rectangle: "+ r1.breadth);
  }
}
```

**Output is:**

Length of Rectangle: 20
Breadth of Rectangle: 10

## Example of Parameterized Constructor

```
class Rectangle {
  int length;
  int breadth;
  Rectangle(int length , int breadth)
  {
  length  = length;
  breadth = breadth;
  }
  public static void main(String args[]) {
  Rectangle r1 = new Rectangle(20,10);
  System.out.println("Length of Rectangle: " + r1.length);
  System.out.println("Breadth of Rectangle: "+ r1.breadth);
  }
}
```

Here parameter (formal arguments) and instance variables name are same

There is ambiguity between the instance variable and parameter

## Difference between Constructor and Method

- The purpose of constructor is to create object of a class while the purpose of a method is to perform a task by executing java code.
- Constructors cannot be abstract, final, static and synchronized while methods can be.
- Constructors do not have return types while methods do.

# Difference between Constructor and Method

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

# Constructor Overloading

- Like methods, a constructor can also be overloaded.
- Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters.
- Constructor overloading is not much different than method overloading.
- In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading we have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.
- **Why do we Overload constructors ?**
  - Constructor overloading is done to construct object in different ways.

# Example of Constructor Overloading-1

```
class Student{
   int id;
   String name;
   int age;
   Student(int i, String n) {
   id = i;
   name = n;
   }
   Student(int i, String n, int a){
   id = i;
   name = n;
   age=a;
   }
```

Both constructor having same name and different number of arguments

## Example of Constructor Overloading-2

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

   public static void main(String args[]){
   Student s1 = new Student(11,"Adil");
   Student s2 = new Student(22,"Hina",20);
   s1.display();
   s2.display();
  }
}
```

**Output is:**

11 Adil 0

22 Hina 20

# Java Copy Constructor

- There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.
- There are many ways to copy the values of one object into another in java. They are:
  - By constructor
  - By assigning the values of one object into another
  - By clone() method of Object class
- Let see first example of copy the values of one object into another using java constructor.

## Copy Constructor(By constructor)-1

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

    Student(Student s){
    id = s.id;
    name =s.name;
    }
```

## Copy Constructor(By constructor)-2

```
void display()
   {
      System.out.println(id+"   "+name);
   }
   public static void main(String args[]){
   Student s1 = new Student(11,"Adil");
   Student s2 = new Student(s1);
   s1.display();
   s2.display();
  }
}
```

**Output is:**

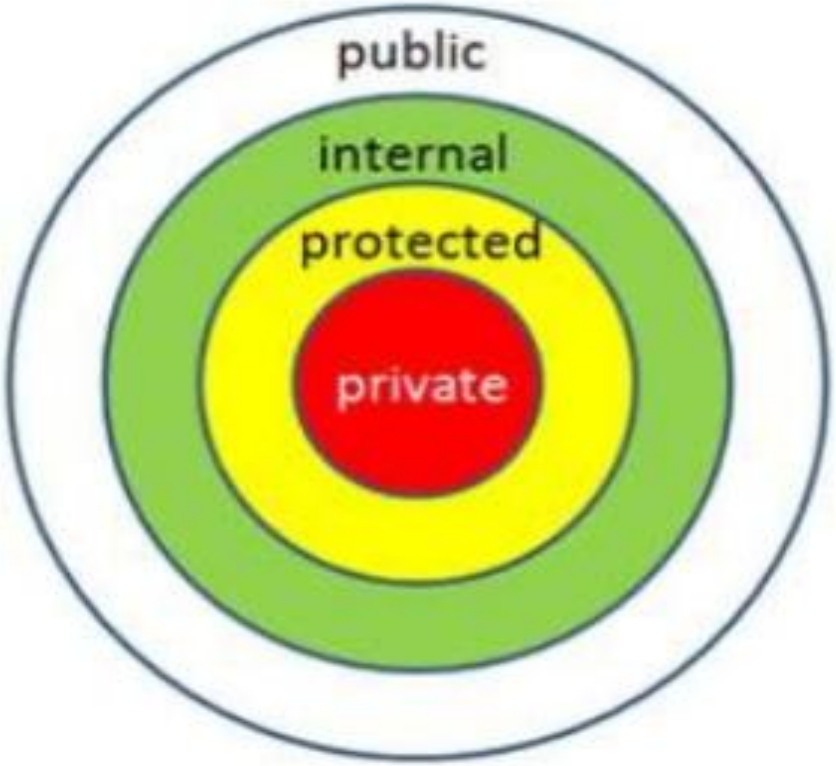
11 Adil
11 Adil

# Constructor Chaining

- In object-oriented programming, **constructor chaining** is the technique of creating an instance of a class with multiple constructors, then using one constructor to call another. The primary use of constructor chaining is to make a program simpler, with fewer repeated lines of code.

We cover this topic in "**use of this keyword**" in detail

# Access Modifiers in java



# Access Modifiers in java

- There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
  - private
  - default
  - protected
  - public
- There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

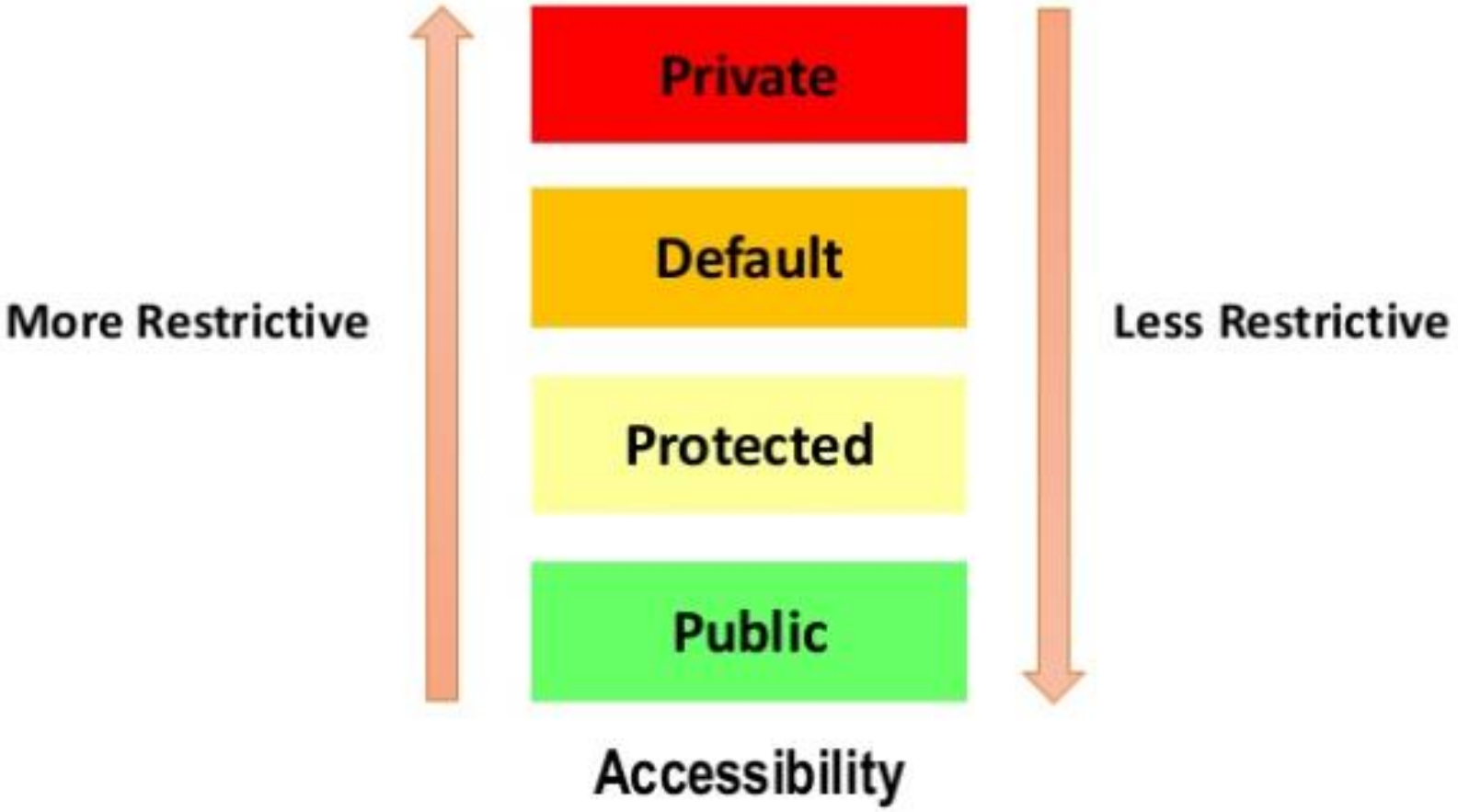
# Access Modifiers in java

- **Access Control Modifiers**
- Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are –
  - Visible to the package, the default. No modifiers are needed.
  - Visible to the class only (private).
  - Visible to the world (public).
  - Visible to the package and all subclasses (protected).

# Access Modifiers in java

- **Non-Access Modifiers**
- Java provides a number of non-access modifiers to achieve many other functionality.
  - The ***static*** modifier for creating class methods and variables.
  - The ***final*** modifier for finalizing the implementations of classes, methods, and variables.
  - The ***abstract*** modifier for creating abstract classes and methods.
  - The ***synchronized*** and ***volatile*** modifiers, which are used for threads.

# Access Modifiers in java



# Access Modifiers in java

- **1) private Access Modifier**
  - The private access modifier is accessible only within class.
  - Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
  - Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
  - Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
  - Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

## Private Access Modifier

```
public class Test{
  private int data=40;
 public static void main(String args[]){
   Test ex = new Test();
   System.out.println("Data is: "+ex.data);


   }
}
```

Here the *data* variable of the Example class is private and this variable accessed from same class itself

# Private Access Modifier

```
class A{
   private int data=40;
   private void msg() {
   System.out.println("Hello java");}
   }
public class Test{
 public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);  //Compile Time Error
  obj.msg();  //Compile Time Error
  }
}
```

Here, the data variable of the A class is private, so there's no way for other classes to retrieve or set its value directly.

# Private Access Modifier

- So, if we want to make this variable available to the outside world, we defined two public methods:
  - *getter()*, which returns the value of variable, and
  - *setter(parameter)*, which sets its value of the variable.

```
class A {
  private int data;
  public int getA() {
    return this.data;
  }
  public void setA(int data) {
    this.data=data;
  }
}
```

# Private Access Modifier

```
class A{
  private int data;
  public void setA(int data) {
    this.data=data;  }
  public int getA() {
    return this.data; }
}
public class Test{
 public static void main(String args[]){
  A obj=new A();
  obj.setA(12);
  System.out.println("Data is: "+obj.getA());
  }
}
```

Accessing possible

Try to avoid using setter and getter methods
Because these methods make your data is publically

**Output is:**
Data is: 12

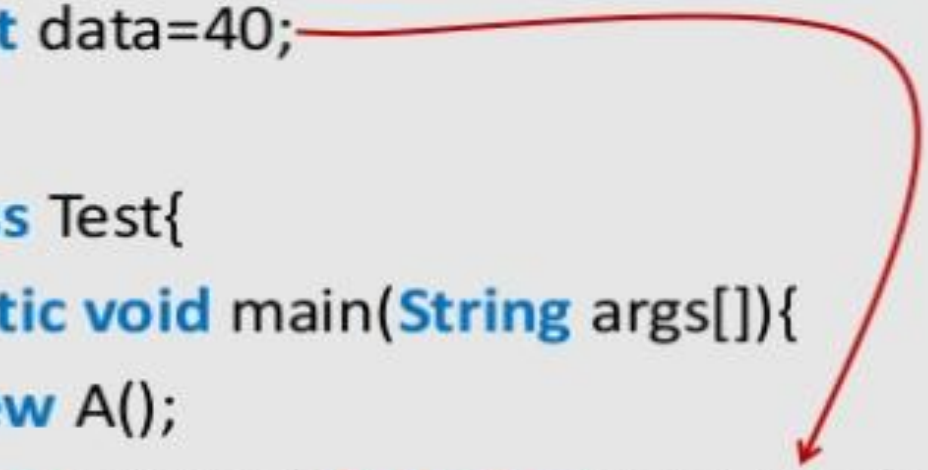
# Access Modifiers in java

- **2) public Access Modifier**
  - The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.
  - A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.
  - However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Note: About package we learn later in detail.. ☺

## public Access Modifier

```
class A{
   public int data=40;
}
public class Test{
 public static void main(String args[]){
   A obj=new A();
   System.out.println("Data is: "+obj.data);
   }
}
```



**Output is:**
Data is: 40

## Access Modifiers in java

- **3) protected access modifier**
  - 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.
  - Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

## protected Access Modifier

```
class A{
  protected int data=40;
}
public class Test{
 public static void main(String args[]){
  A obj=new A();
  System.out.println("Data is: "+obj.data);
  }
}
```

Access protected instance variable in other class

**Output is:**
Data is: 40

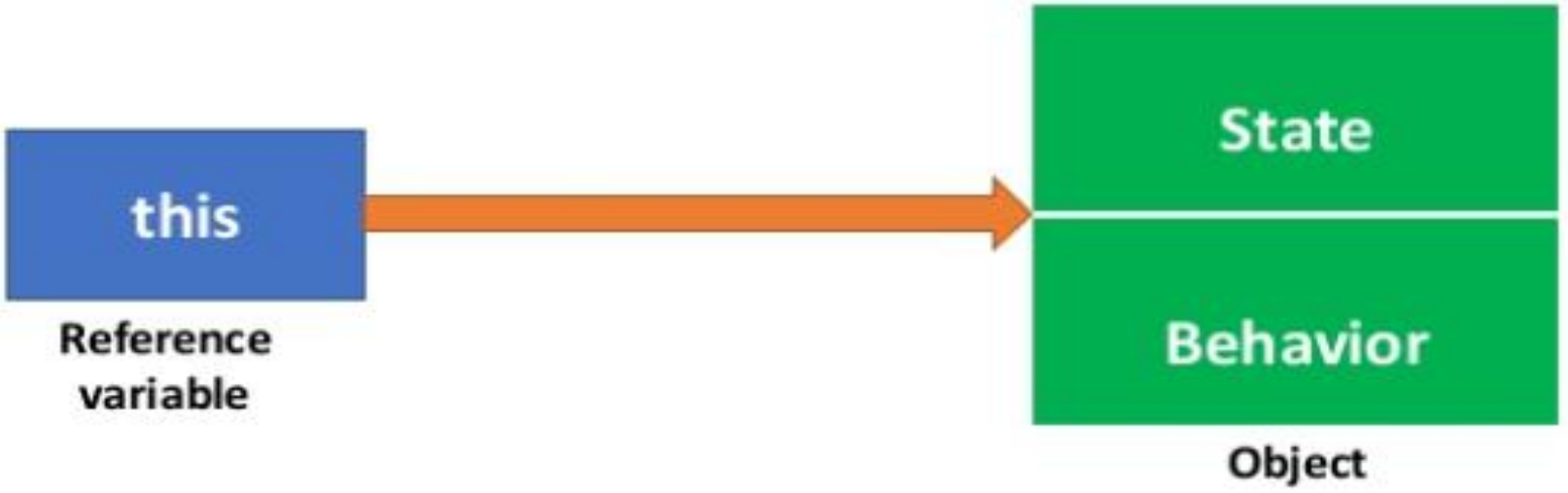
# Access Modifiers in java

| | public | private | protected | < unspecified > |
|---|---|---|---|---|
| **class** | allowed | not allowed | not allowed | allowed |
| **constructor** | allowed | allowed | allowed | allowed |
| **variable** | allowed | allowed | allowed | allowed |
| **method** | allowed | allowed | allowed | allowed |

| | class | subclass | package | outside |
|---|---|---|---|---|
| **private** | allowed | not allowed | not allowed | not allowed |
| **protected** | allowed | allowed | allowed | not allowed |
| **public** | allowed | allowed | allowed | allowed |
| **< unspecified >** | allowed | not allowed | allowed | not allowed |

# This keyword in java

- **this** is a reference variable that refers to the current object. It is a keyword in java language represents current class object.



## This keyword in java

- **Usage of this keyword**
- It can be used to refer current class instance variable.
- this() can be used to invoke current class constructor.
- It can be used to invoke current class method (implicitly)
- It can be passed as an argument in the method call.
- It can be passed as argument in the constructor call.
- It can also be used to return the current class instance.

# This keyword in java

- **Why use this keyword in java ?**
- The main purpose of **using this keyword** is to differentiate the formal parameter and data members of class, whenever the formal parameter and data members of the class are similar then jvm get ambiguity (no clarity between formal parameter and member of the class)
- To differentiate between formal parameter and data member of the class, the data member of the class must be preceded by "this".
- **"this"** keyword can be use in two ways.
  - this . (this dot)
  - this() (this off)

# This keyword in java

- **Usage of this keyword**
- **It can be used to refer current class instance variable.**
- this() can be used to invoke current class constructor.
- It can be used to invoke current class method (implicitly)
- It can be passed as an argument in the method call.
- It can be passed as argument in the constructor call.
- It can also be used to return the current class instance.

# This keyword in java

- **this . (this dot)**
- which can be used to differentiate variable of class and formal parameters of method or constructor.
- **"this"** keyword are used for two purpose, they are
  - It always points to current class object.
  - Whenever the formal parameter and data member of the class are similar and JVM gets an ambiguity (no clarity between formal parameter and data members of the class).
- To differentiate between formal parameter and data member of the class, the data members of the class must be preceded by **"this"**.

# This keyword in java

- **Syntax**

This is dot operator

this.data member of current class.

- **Note:** If any variable is preceded by **"this"** JVM treated that variable as class variable.

## Example without using this keyword-1

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

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

Parameter (formal arguments) and instance variables are same

local variable and instance variable name are same

## Example without using this keyword-2

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


   Student s1 = new Student(11,"Adil");
   Student s2 = new Student (22,"Hina");
   s1.display();
   s2.display();
   }
}
```

default values of instance variables

In the above example, parameter (formal arguments) and instance variables are same that is why we are using **"this"** keyword to distinguish between local variable and instance variable.

**Output is:**

0 null

0 null

## Solution of The Previous Problem by this Keyword-1

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

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

using this keyword to distinguish between local variable and instance variable.

## Solution of The Previous Problem by this Keyword-1

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

   Student s1 = new Student(11,"Adil");
   Student s2 = new Student (22,"Hina");
   s1.display();
   s2.display();
   }
}
```

**Output is:**

11 Adil
22 Hina

# This keyword in java

- **Usage of this keyword**
- It can be used to refer current class instance variable.
- **this() can be used to invoke current class constructor.**
- It can be used to invoke current class method (implicitly)
- It can be passed as an argument in the method call.
- It can be passed as argument in the constructor call.
- It can also be used to return the current class instance.

# This keyword in java

- **this ()**
- which can be used to call one constructor within the another constructor without creation of objects multiple time for the same class.
- The this() constructor call can be used to invoke the current class constructor (constructor chaining).
- This approach is better if you have many constructors in the class and want to reuse that constructor.
- **Syntax**

```
this();     // call no parametrized or default constructor
this(value1,value2,.....)     //call parametrize constructor
```

## this() Used to Invoked Current Class Constructor-1

```
//Program of this() constructor call (constructor chaining)
class Student{
   int id;
   String name;
   Student()  {
     System.out.println("default constructor is invoked");
    }
   Student(int id, String name) {
   this ();
   this.id = id;
   this.name = name;
   }
```

Here this() used to invoked current class constructor.

## this() Used to Invoked Current Class Constructor-1

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

  public static void main(String args[]){
  Student e1 = new Student(11,"Adil");
  Student e2 = new Student(22,"Hina");
  e1.display();
  e2.display();
  }
}
```

**Output is:**

```
default constructor is invoked
default constructor is invoked
11   Adil
22   Hina
```

# this() Used to Invoked Current Class Constructor

## Correct use of "this()"

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

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

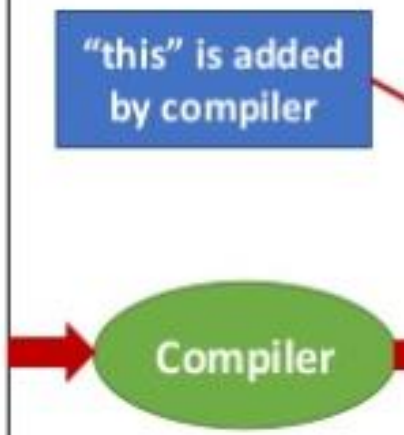
## Incorrect use of "this()"

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

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

# This keyword in java

- By using this keyword you can invoke the method of the current class. If you do not use the this keyword, compiler automatically adds this keyword at time of invoking of the method.

```
class Student {
   void show() {}
    void marks() {
    show();  }
    }
 public static void main(
String args[]){
   Student s = new Student();
   s.marks();
   }
}
```




```
class Student {
   void show() {}
    void marks() {
    this.show();  }
    }
 public static void main(
String args[]){
   Student s = new Student();
   s.marks();
   }
}
```

## this Keyword Used to Invoke Current Class Method (Implicitly)-1

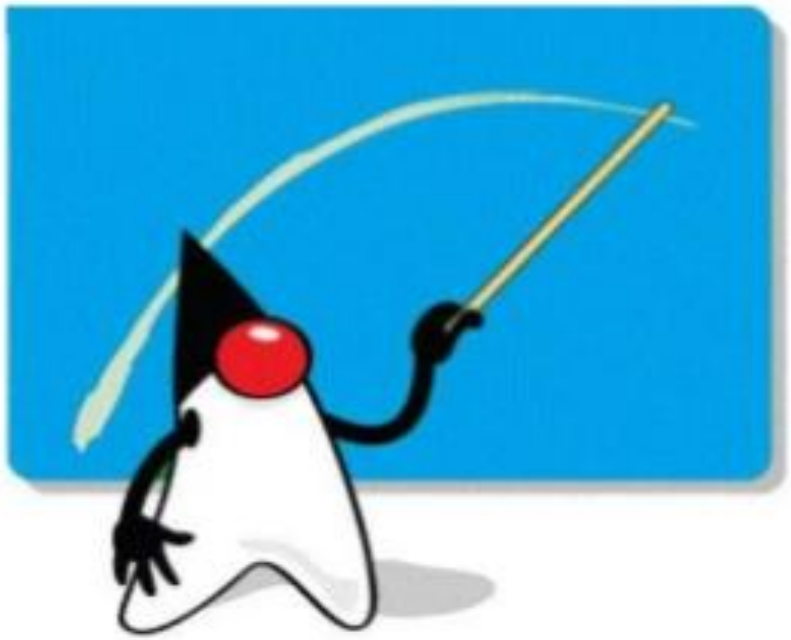
```
class Student {
 void show() {
  System.out.println("Hina got A+");
  }
  void marks() {
  //no need to use this here because compiler does it.
  this.show();
  }
void display() {
  show();   //compiler act marks() as this.marks()
  }
```

## this Keyword Used to Invoke Current Class Method (Implicitly)-2

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

**Output is:**
Hina got A+

# static keyword in java



# Static keyword in java

- The **static keyword** is used in java mainly for memory management. It is used with variables, methods, blocks and nested class. It is a keyword that are used for share the same variable or method of a given class. This is used for a constant variable or a method that is the same for every instance of a class. The main method of a class is generally labeled static.
- No object needs to be created to use static variable or call static methods, just put the class name before the static variable or method to use them. Static method can not call non-static method.
- **In java language static keyword can be used for following**
  - variable (also known as class variable)
  - method (also known as class method)
  - block
  - nested class

# Static keyword in java

- **1) Java static variable**
- If we declare any variable as static, it is known static variable.
  - The static variable can be used to refer the common property of all objects (that is not unique for each object). For Example company name of employees, college name of students etc. Name of the college is common for all students.
  - The static variable allocate memory only once in class area at the time of class loading.
- **Advantage of static variable**
  - It makes your program **memory efficient** (i.e. it saves memory).

# Static keyword in java

- 1) Java static variable
- **When and why we use static variable**
- Suppose we want to store record of all employee of any company, in this case employee id is unique for every employee but company name is common for all. When we create a static variable as a company name then only once memory is allocated otherwise it allocate a memory space each time for every employee.

# Static keyword in java

- **Understanding problem without static variable**

```
class Student{
    int roll-no;
    String name;
    String university="SAC";
}
```

- Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All student have its unique roll-no and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

# Static keyword in java

- **Applicable to**
- The Static keyword can be applied to
  - *Method*
  - *Variable*
  - *Class nested within another Class*
  - *Initialization Block*
- **Not Applicable to**
- The Static keyword can not be applied to
  - Class (Not Nested)
  - Constructor
  - Interfaces
  - Method Local Inner Class(Difference then nested class)
  - Inner Class methods
  - Instance Variables
  - Local Variables

# Static keyword in java

- **Static Keyword Rules**
- ***Variable or Methods*** marked static belong to the **Class** rather then to any particular Instance.
- **Static Method or variable** can be used without creating or referencing any instance of the Class.
- If there are instances, a static variable of a Class will be shared by all instances of that class, This will result in **only one copy**.
- A static Method can't access a non static variable nor can directly invoke non static Method (It can invoke or access Method or variable via *instances*).

## Static keyword in java

- **Syntax for declare static variable:**

```
public static variableName;
```

- **Example**

```
public static double PI=3.1415
```

# Example of static Variable-1

```
class Student{
  int roll-no;
  String name;
  static String college ="SBC";

  Student(int r, String n){
  roll-no = r;
  name = n;
  }
   void display (){
    System.out.println(roll-no+" "+name+" "+college);
  }
```

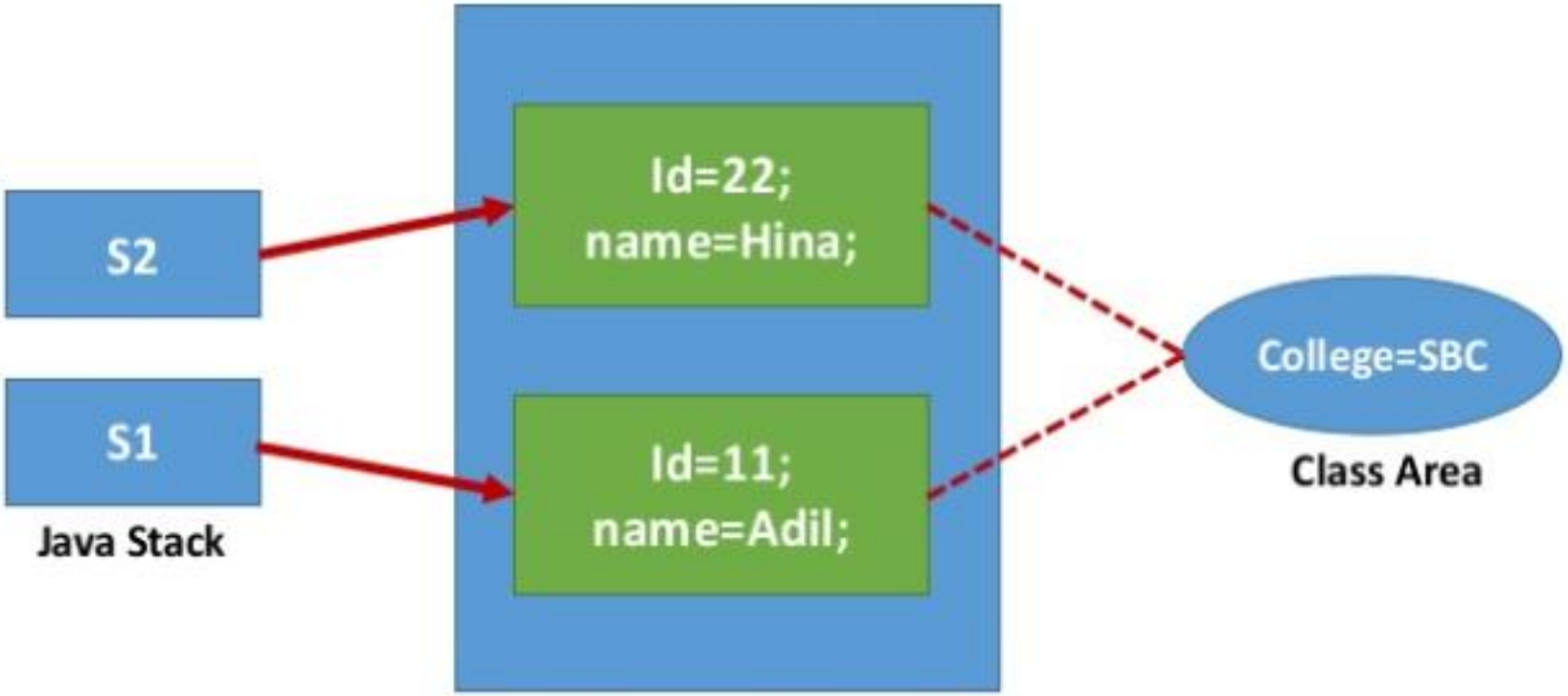
This college variable is commonly sharable by both S1 and S2 objects.

## Example of static Variable-1

```
public static void main(String args[]){
  Student s1 = new Student(11,"Adil");
  Student s2 = new Student(22,"Hina");
  s1.display();
  s2.display();
 }
}
```

**Output is:**
11 Adil SBC
22 Hina SBC

# static variable



# Program Of Counter By Static Variable

```
class Counter{
  //will get memory when instance is created
  static int count=0;
  Counter(){
  count++;
  System.out.println(count);
  }
public static void main(String args[]){
  Counter c1=new Counter();
  Counter c2=new Counter();
  Counter c3=new Counter();
 }
}
```

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

**Output is:**

1
2
3

## Difference Between Static and non-Static Variable in Java

| Non-static variable | Static variable |
| --- | --- |
| • These variables should not be proceeded by any static keyword | • These variables are proceeded by any static keyword |
| `class A { int a; }` | `class A { static int a; }` |
| • Memory is allocated for these variables whenever an object is created.<br>• Memory is allocated is multiple times whenever a new object is created. | • Memory is allocated for these variables at the time of loading of the class.<br>• Memory is allocated for these variables only once in the program. |

## Difference Between Static and non-Static Variable in Java

| Non-static variable | Static variable |
| --- | --- |
| • Non-static variable also known as instance variable while because memory is allocated whenever is created.<br>• Non-static variable are specific to an object.<br>• Non-static variable can access with object reference.<br>• **Syntax** | • Memory is allocated at the time of loading of class so that these are also known as class variables.<br>• Static variable are common for every object that mean these memory location can be shareable by every object reference or same class.<br>• static variable can access with class reference.<br>• **Syntax** |
| Obj_ref.variable_name | class_name.variable_name |

## Static keyword in java

- **2) Java static method**
- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

# Static keyword in java

- **Methods declared as static have several restrictions:**
  - They can only call other **static** methods
  - They must only access **static** data.
  - They cannot use **this** or **super** in anyway (**Super** is a keyword used in Inheritance ).
- **Syntax for declare static method:**

```
public static void methodName()
{
.....
.....
}
```

## static method Example

```
class Cube{
  static void cube(){
    int x=5*5*5;
    System.out.println("Cube is: "+x);
  }

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

## Difference Between Non-static and Static Method

| Non-static Method | Static Method |
| --- | --- |
| • These method never be preceded by static keyword **Example:** | • These method always preceded by static keyword **Example:** |
| `void fun()`<br>`{`<br>`.....`<br>`.....`<br>`}` | `static void fun()`<br>`{`<br>`.....`<br>`.....`<br>`}` |
| • Memory is allocated multiple time whenever method is calling. | • Memory is allocated only once at the time of class loading. |

# Difference Between Non-static and Static Method

| Non-static Method | Static Method |
| --- | --- |
| • It is specific to an object so that these are also known as instance method.<br>• These methods always access with object reference<br>**Syntax:** | • These are common to every object so that it is also known as member method or class method.<br>• These property always access with class reference<br>**Syntax:** |
| **Objref.methodname()** | **className.methodname();** |
| • If any method wants to be execute multiple time that can be declare as non static. | • If any method wants to be execute only once in the program that can be declare as static . |

## Static keyword in java

- **3) Java static block**
- Is used to initialize the static data member.
- It is executed before main method at the time of class loading.
- **Syntax of static block**

```
Static
{
   ..........
   ..........
}
```

# Example of static block

```
class Cube{
 static{
    int a=3*3*3;
    System.out.println("Cube is:"+a);
 }
 public static void main(String args[]){
  System.out.println("Hello main");
 }
}
```

**Output is:**

```
Cube is:27
Hello main
```

# Inheritance in Java

# Inheritance in Java

- **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object.
- When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class(Parent) and Sub class(child) in Java language.
- Inheritance defines is-a relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.

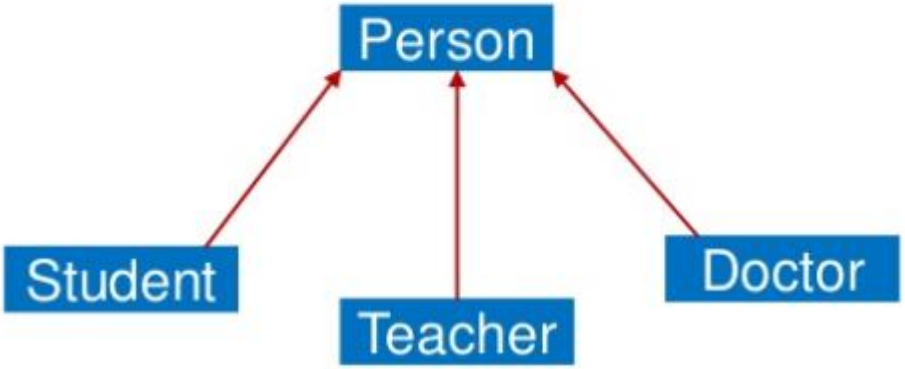
# Inheritance in Java

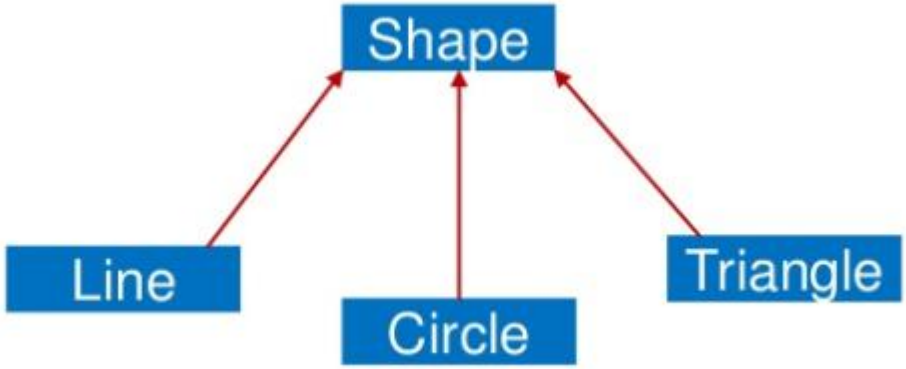
- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.
- **Why use Inheritance ?**
- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

# Inheritance in Java

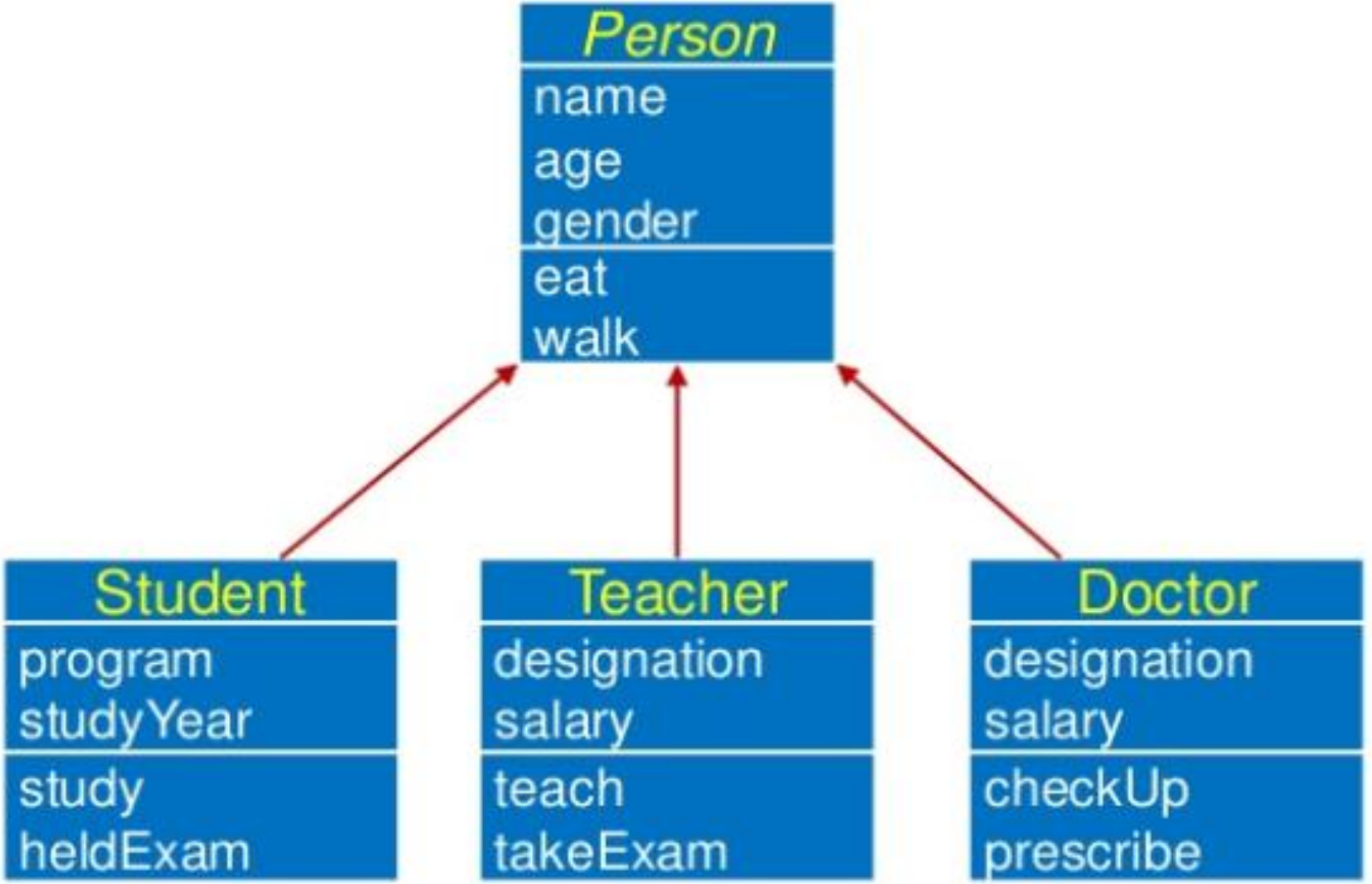
| No | Term | Definition |
|---|---|---|
| 1 | Inheritance | **Inheritance** is a process where one object acquires the properties of another object |
| 2 | Subclass | Class which inherits the properties of another object is called as **subclass** |
| 3 | Superclass | Class whose properties are inherited by subclass is called as **superclass** |
| 4 | Keywords Used | extends and implements |

# Example – Inheritance



# Example – "IS A" Relationship



# Inheritance in Java

- **Inheritance in Java**
- Inheritance in Java is done using
  - **extends** – In case of Java class and abstract class
  - **implements** – In case of Java interface.
- **What is inherited**
  - In Java when a class is extended, sub-class inherits all the **public, protected** and **default (Only if the sub-class is located in the same package as the super class)** methods and fields of the super class.
- **What is not inherited**
  - **Private** fields and methods of the super class are not inherited by the sub-class and can't be accessed directly by the subclass.
  - Constructors of the super-class are not inherited. There is a concept of constructor chaining in Java which determines in what order constructors are called in case of inheritance.

# Inheritance in Java

- **Syntax of Inheritance**

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

- The **extends keyword** indicates that you are making a new class that derives from an existing class.

# Inheritance in Java

- **extends Keyword**
- **extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

```
class Super {
  .....
  .....
}
class Sub extends Super {
  .....
  .....
}
```

# Inheritance in Java

- **IS-A Relationship with Example**
- IS-A is a way of saying : This object is a type of that object.

```
public class Vehicle{
}

public class FourWheeler extends Vehicle{
}

public class TwoWheeler extends Vehicle{
}

public class Car extends FourWheeler{
}
```

# Inheritance in Java

```
public class Vehicle{
}

public class FourWheeler extends Vehicle{
}

public class TwoWheeler extends Vehicle{
}

public class Car extends FourWheeler{
}
```

- **Conclusions from above Example :**
- Vehicle is the **superclass** of TwoWheeler class.
- Vehicle is the **superclass** of FourWheeler class.
- TwoWheeler and FourWheeler are **subclasses** of Vehicle class.
- Car is the **subclass** of both FourWheeler and Vehicle classes.

# Inheritance in Java

- Please Note:
- In inheritance Parent Class and Child Class having multiple names, List of the name are below:
- **Parent Class = Base Class = Super Class**
- **Child Class = Derived Class = Sub Class**

# Simple Example of Inheritance

```
class Parent
{
   public void p1() {
      System.out.println("Parent method");
   }
}
public class Child extends Parent {
   public void c1() {
      System.out.println("Child method");
   }
   public static void main(String[] args) {
      Child cobj = new Child();
      cobj.c1();  //Calling method of Child class
      cobj.p1();  //Calling method of Parent class
   }
}
```

**Output is:**

Child method
Parent method

# Another example of Inheritance

```
class Parent
{
  int id=11;
  String name="Adil";
}
public class Child extends Parent {
   public void display()
   {
      System.out.println("ID is: "+id);
      System.out.println("Name is: "+name);
   }
   public static void main(String[] args)  {
      Child obj = new Child();
      obj.display();
   }
}
```

Here in Child Class we Can Inherit Or Access the Properties of Parent Class

**Output is:**

ID is: 11

Name is: Adil

# Access Control and Inheritance

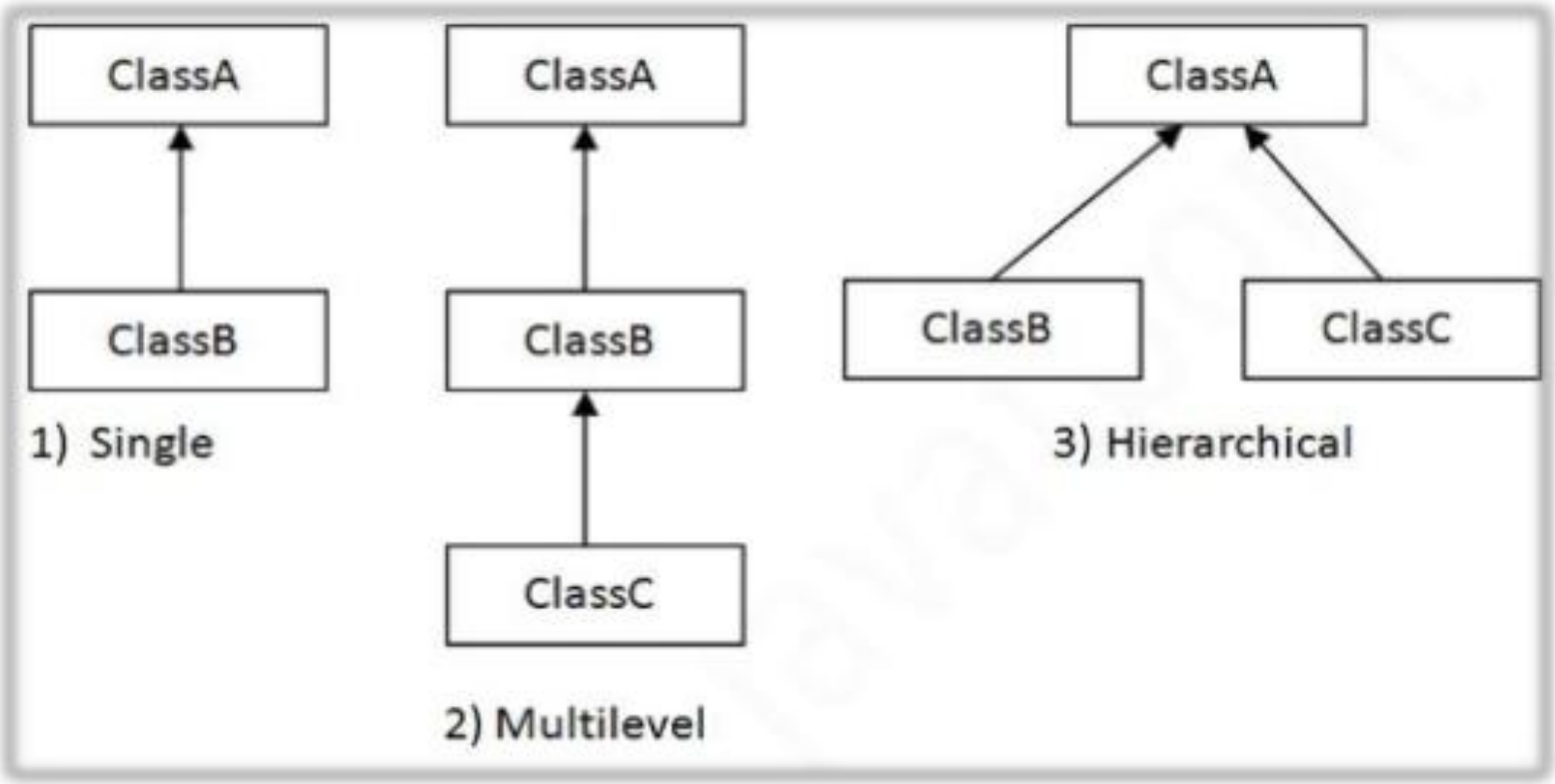
- A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the members of derived classes should be declared private in the base class.

| Access | public | protected | private |
|---|---|---|---|
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

# Inheritance in Java

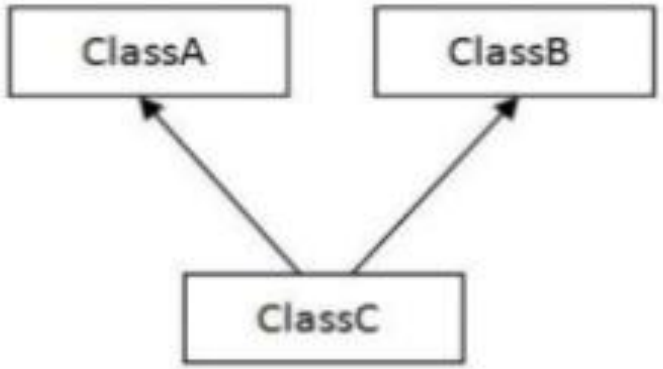
- **Types of Inheritance**
- Based on number of ways inheriting the feature of base class into derived class we have five types of inheritance; they are:
- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

# Types of Inheritance

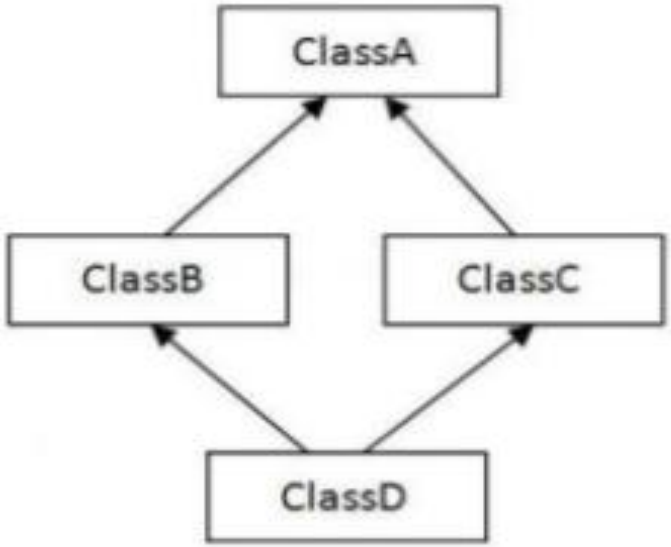


# Types of Inheritance

- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



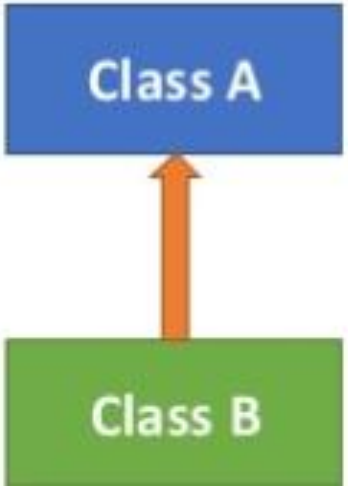

4) Multiple

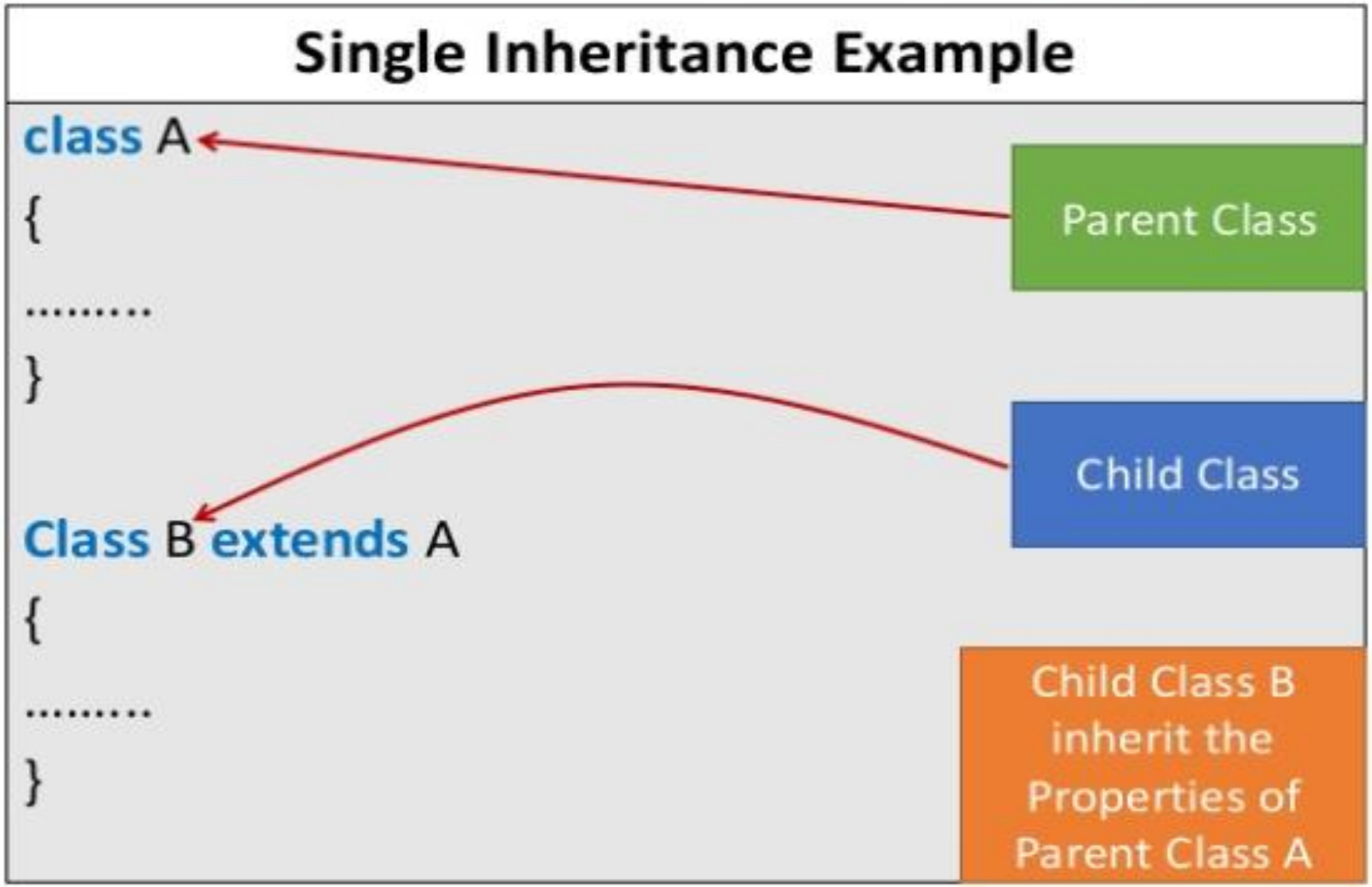



5) Hybrid

# Types of Inheritance

- **1) Single Inheritance**
- **Single inheritance** is easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.




Single Inheritance Example
class A
{
.....
}
Class B extends A
{
.....
}
Parent Class
Child Class
Child Class B inherit the Properties of Parent Class A

# Single Inheritance Example

```
class A {
  int data=10;
}
class B extends A {
   public void display()
   {
     System.out.println("Data is:"+data);
   }
   public static void main(String args[])
   {
     B obj = new B();
     obj.display();
  }
}
```

Child Class B inherit/Access the data field of Parent Class A

**Output is:**
Data is:10

# Another Single Inheritance Example

```
class Faculty {
  float salary=30000;
}
class Science extends Faculty {
  float bonus=2000;
public static void main(String args[]) {
 Science obj=new Science();
 System.out.println("Salary is:"+obj.salary);
 System.out.println("Bonus is:"+obj.bonus);
 }
}
```

**Output is:**

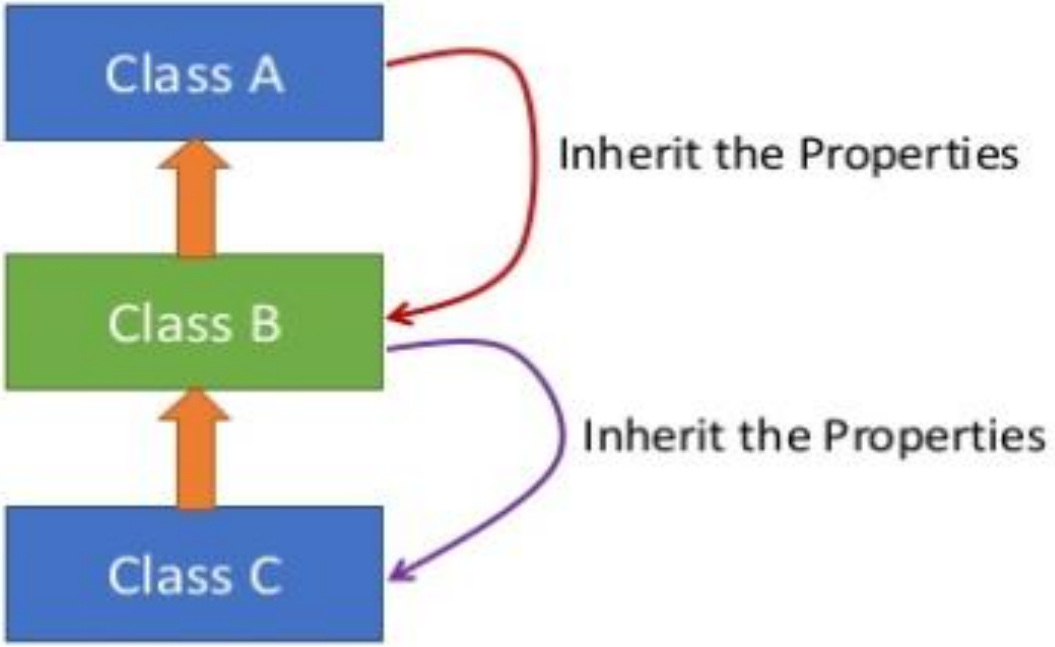
Salary is:30000.0

Bonus is:2000.0

# Types of Inheritance

- **2) Multilevel Inheritance**
- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.
- **Single base class + single derived class + multiple intermediate base classes.**
- **Intermediate base classes**
- An intermediate base class is one in one context with access derived class and in another context same class access base class.

# Multilevel Inheritance

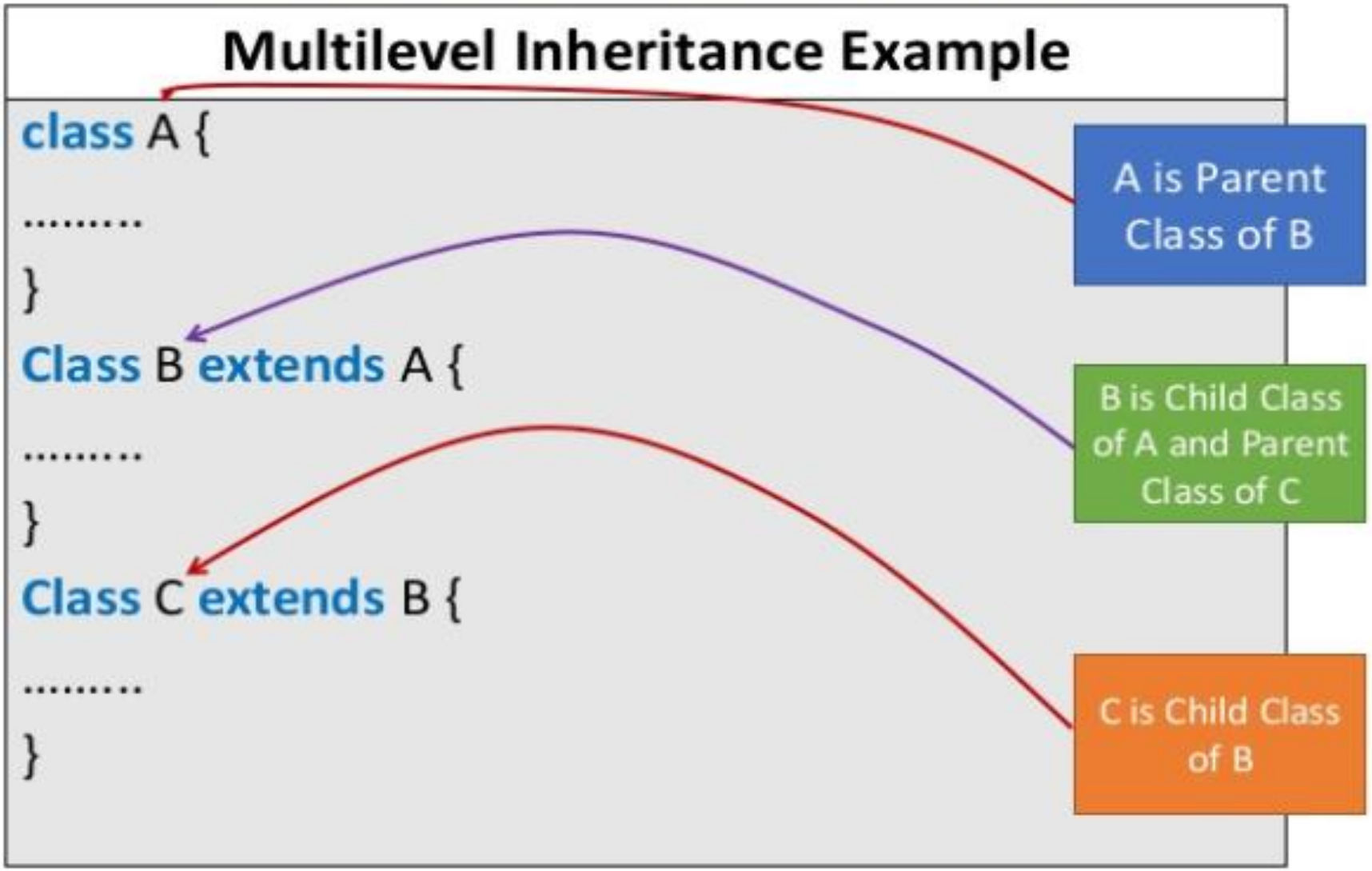



- Here class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that's what is called multilevel inheritance.

# Multilevel Inheritance

- Class C can inherit the Members of both Class A and B Show below :


**C Contains B Which Contains A**


Multilevel Inheritance Example
class A {
.....
}
Class B extends A {
.....
}
Class C extends B {
.....
}
A is Parent Class of B
B is Child Class of A and Parent Class of C
C is Child Class of B

# Multilevel Inheritance Example

```
class A {
  int data=10;
}
class B extends A{
}
class C extends B  {
   public void display() {
     System.out.println("Data is:"+data);
   }
 public static void main(String args[]) {
     C obj = new C();
     obj.display();
  }
}
```

Here Class B inherit the properties of Class A and Class C inherit the properties of Class B

**Output is:**
Data is:10

# Another Multilevel Inheritance Example

```
class Faculty {
  float total_sal=0, salary=1000;
}
class HRA extends Faculty {
  float hra=2000;
}
class DA extends HRA {
  float da=3000;
}
class Science extends DA {
  float bonus=4000;
public static void main(String args[]) {
  Science obj=new Science();
  obj.total_sal=obj.salary+obj.hra+obj.da+obj.bonus;
  System.out.println("Total Salary is:"+obj.total_sal);
 }
```

**Class C look Like This After inheritance**

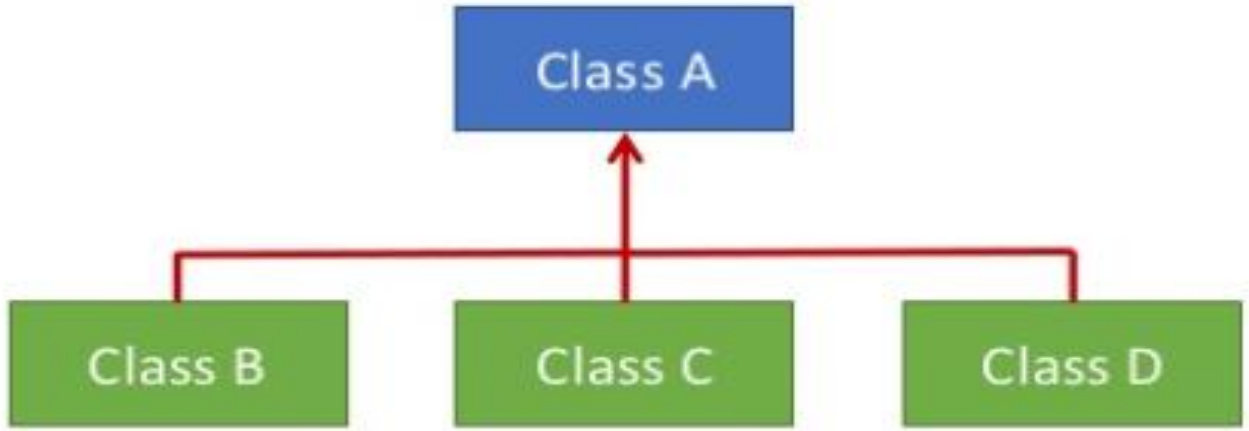
```
float total_sal=0, salary=1000;
float hra=2000;
float da=3000;
float bonus=4000;
```

Here we can Access the Properties of Class "Faculty" , Class "HRA" and Class "DA" using the Object of Class "Science"

**Output is:**
Total Salary is:10000.0

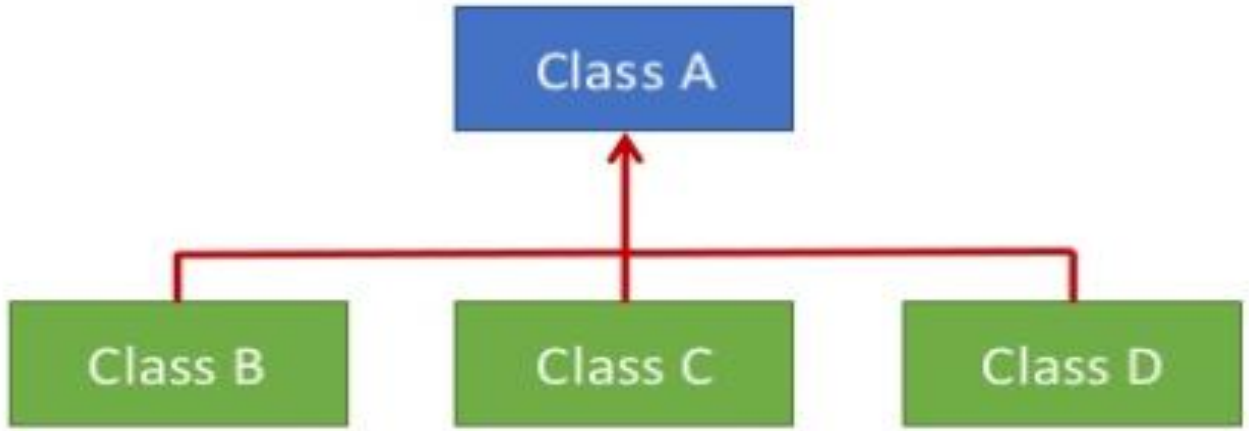
# Types of Inheritance

- **3) Hierarchical Inheritance**
- In this **inheritance** multiple classes inherits from a **single** class i.e there is one super class and **multiple** sub classes. As we can see from the below diagram when a same class is having more than one sub class (or) more than one sub class has the same parent is called as **Hierarchical Inheritance**.

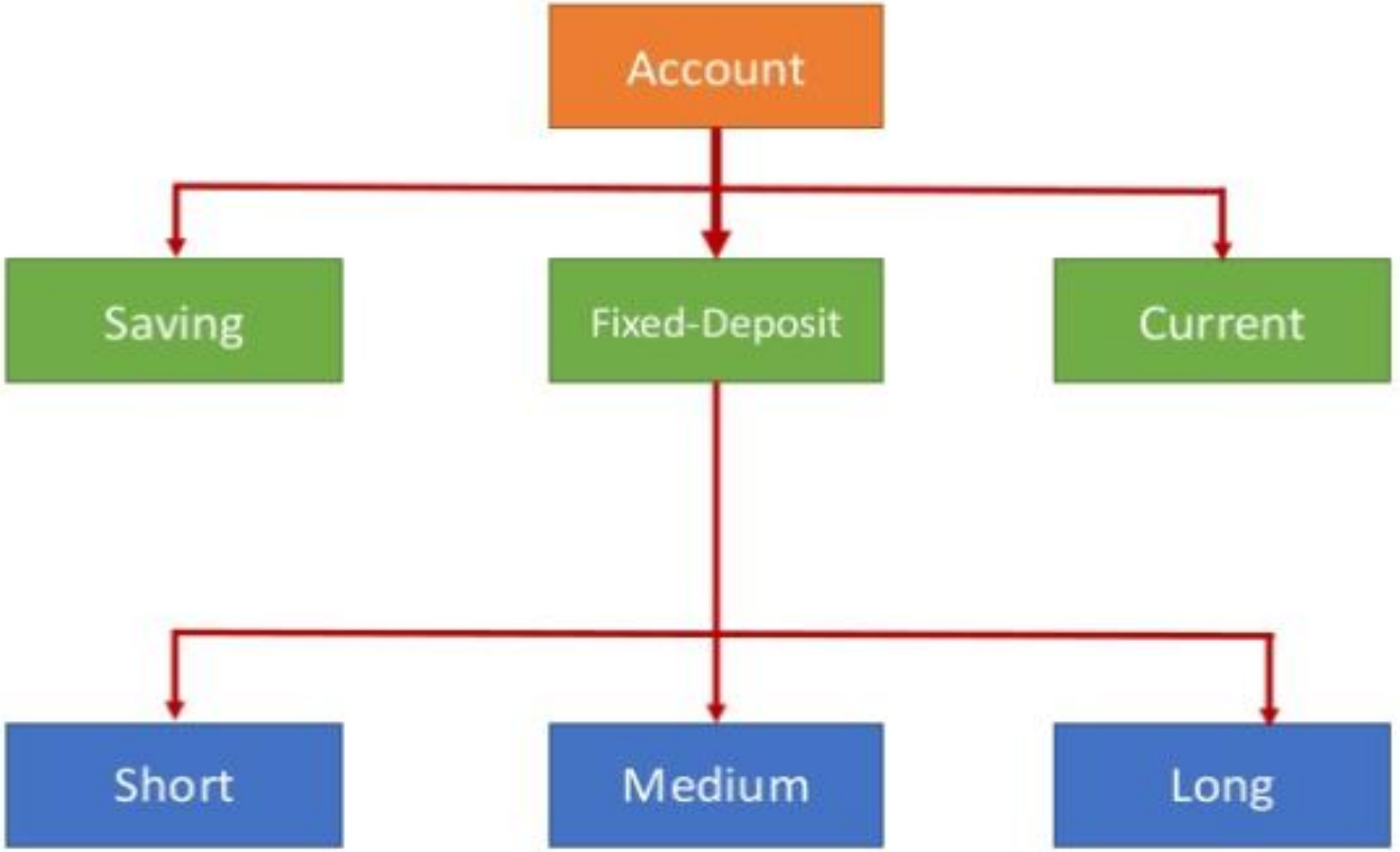


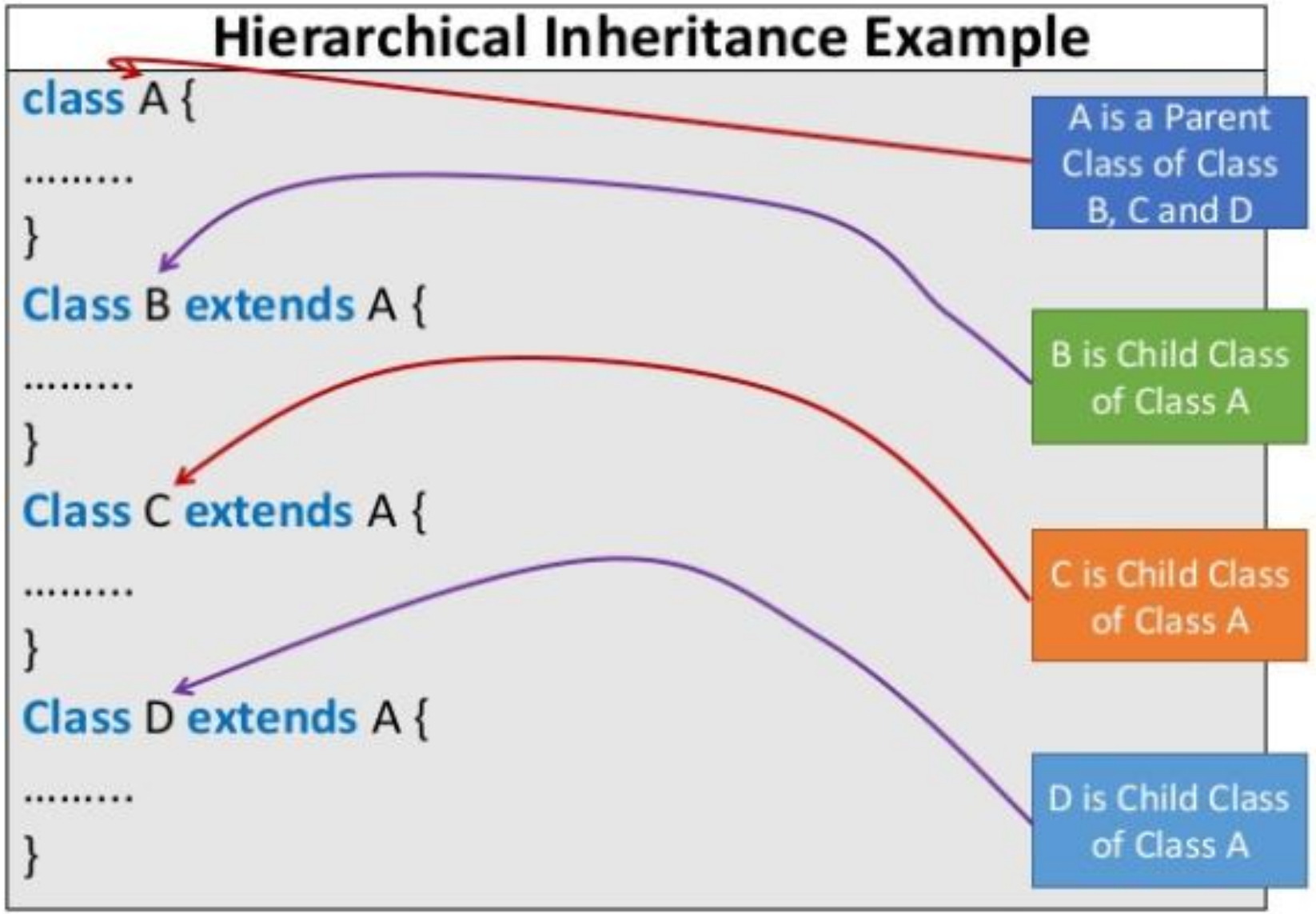
# Types of Inheritance

- **3) Hierarchical Inheritance**
- In this **inheritance** multiple classes inherits from a **single** class i.e there is one super class and **multiple** sub classes. As we can see from the below diagram when a same class is having more than one sub class (or) more than one sub class has the same parent is called as **Hierarchical Inheritance**.



# Hierarchical Inheritance(Real time Example)




Hierarchical Inheritance Example
class A {
.....
}
Class B extends A {
.....
}
Class C extends A {
.....
}
Class D extends A {
.....
}
A is a Parent Class of Class B, C and D
B is Child Class of Class A
C is Child Class of Class A
D is Child Class of Class A

## Hierarchical Inheritance Example

```
class A {
  int data=10;
}
class B extends A{
}
class C extends A  {
}
class D extends A  {
 public static void main(String args[]) {
    B obj1 = new B();
    C obj2 = new C();
    D obj3 = new D();
   System.out.println("Data in Class B is: "+obj1.data);
   System.out.println("Data in Class C is: "+obj2.data);
   System.out.println("Data in Class D is: "+obj3.data);
  }
}
```

**Output is:**

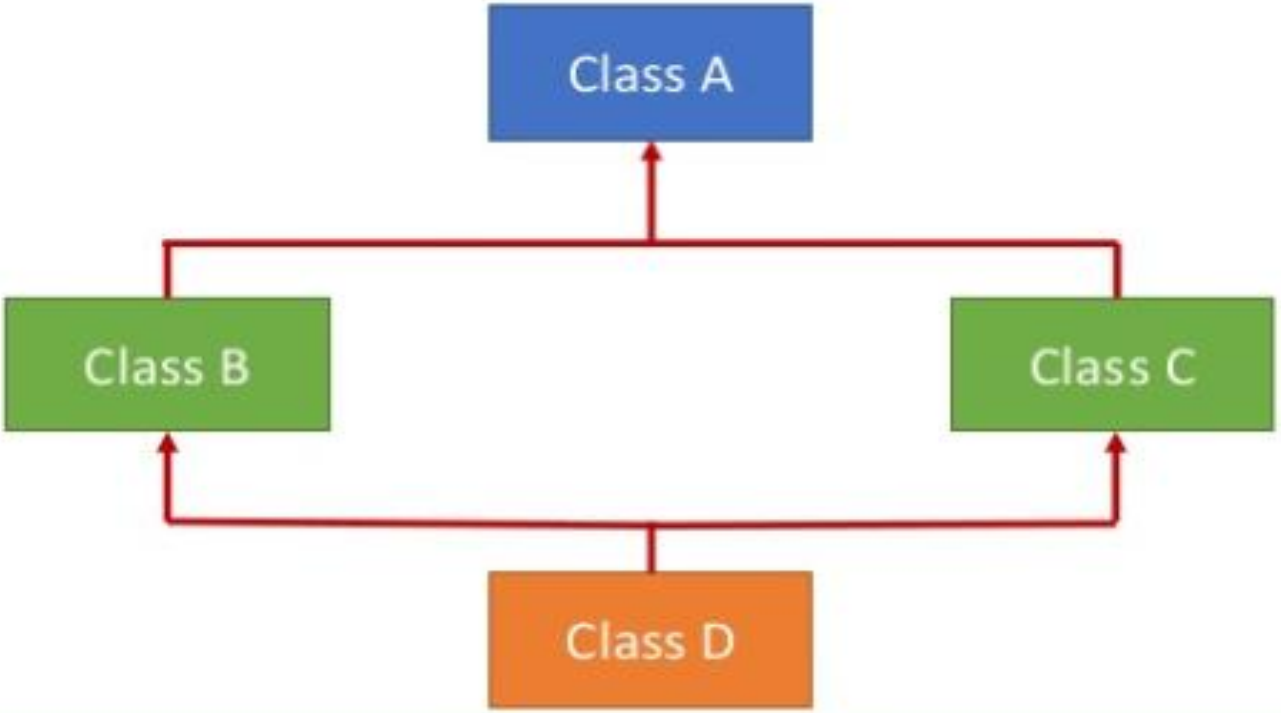
Data in Class B is: 10
Data in Class C is: 10
Data in Class D is: 10

## Types of Inheritance

- **4) Multiple Inheritance**
- In java programming, multiple and hybrid inheritance is not supported through classes but supported through interface only. We will learn about interfaces later.
- **Q) Why multiple inheritance is not supported in java?**
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

# Types of Inheritance

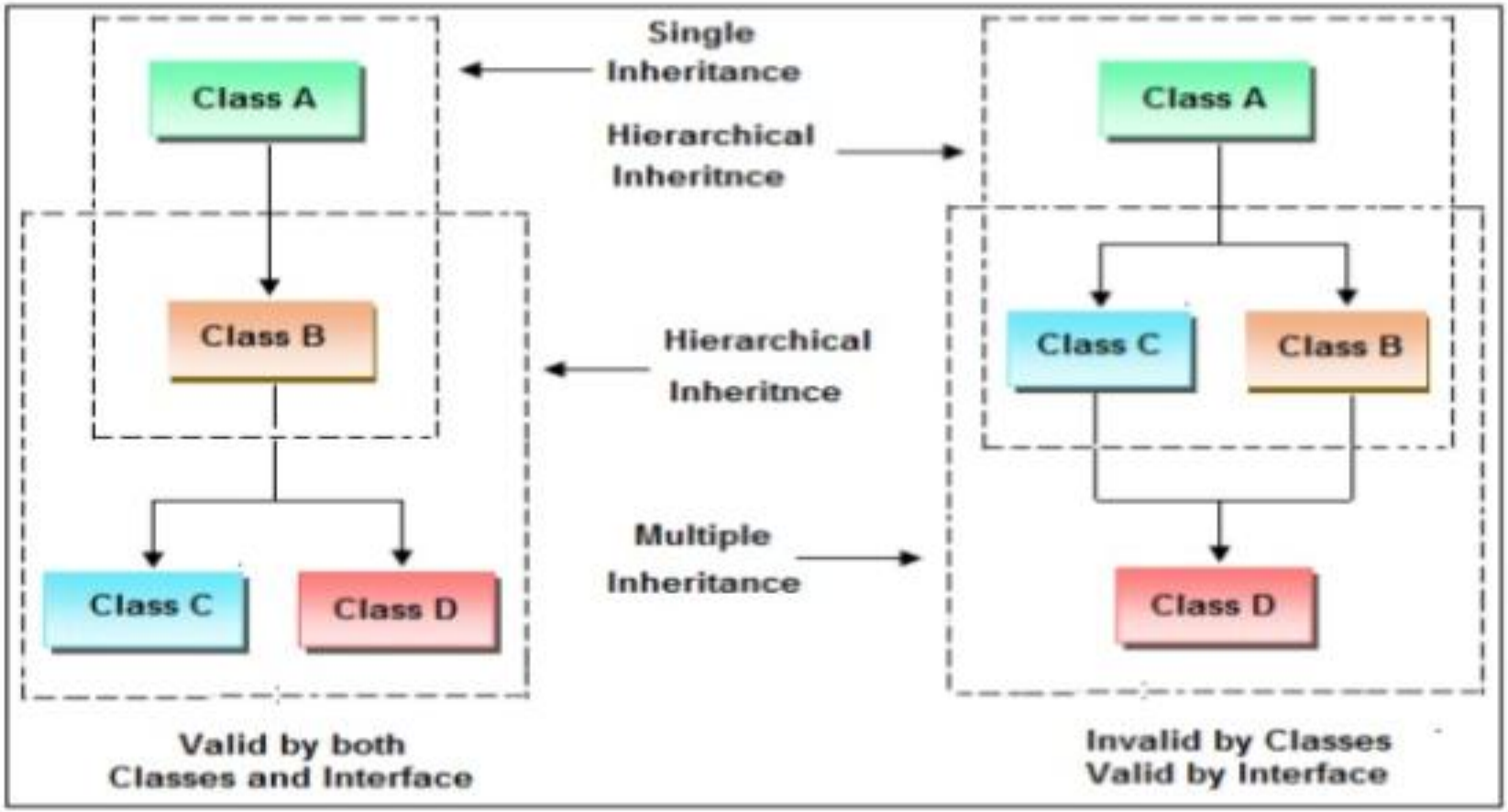
- **5) Hybrid inheritance**
- Any combination of previous three inheritance (single, hierarchical and multi level) is called as hybrid inheritance.



# Hybrid Inheritance

- In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple** inheritance
- A typical flow diagram would look like in the previous slide.
- A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right.
- By using **interfaces** we can have multiple as well as **hybrid inheritance** in Java.

# Hybrid inheritance



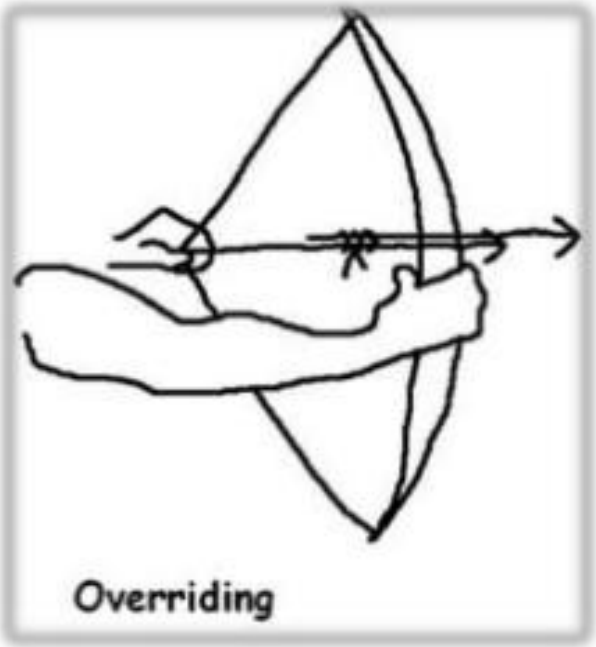
# Inheritance in Java

- **Advantage of inheritance**
- If we develop any application using concept of Inheritance than that application have following advantages,
- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

# Inheritance in Java

- **Disadvantages of Inheritance**
- Inheritance base class and child classes are tightly coupled. Hence If you change the code of parent class, it will get affects to the all the child classes.
- In class hierarchy many data members remain unused and the memory allocated to them is not utilized. Hence affect performance of your program if you have not implemented inheritance correctly.

# Method Overriding in Java



# Method Overriding in Java

- Declaring a method in **subclass** which is already present in **parent class** is known as method overriding.

OR

- In other words If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java.**

OR

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

Without Inheritance Method Overriding is not Possible

# Method Overriding in Java

- **Advantage of Java Method Overriding**
  - Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
  - Method Overriding is used for Runtime Polymorphism

- **Rules for Method Overriding**
  - method must have same name as in the parent class.
  - method must have same parameter as in the parent class.
  - must be IS-A relationship (inheritance).

## Problem Without Method Overriding

```
class Vehicle{
  void run() {
   System.out.println("Vehicle is running");
   }
}
class Bike extends Vehicle{

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

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

**Output is:**
Vehicle is running

# Example of Method Overriding

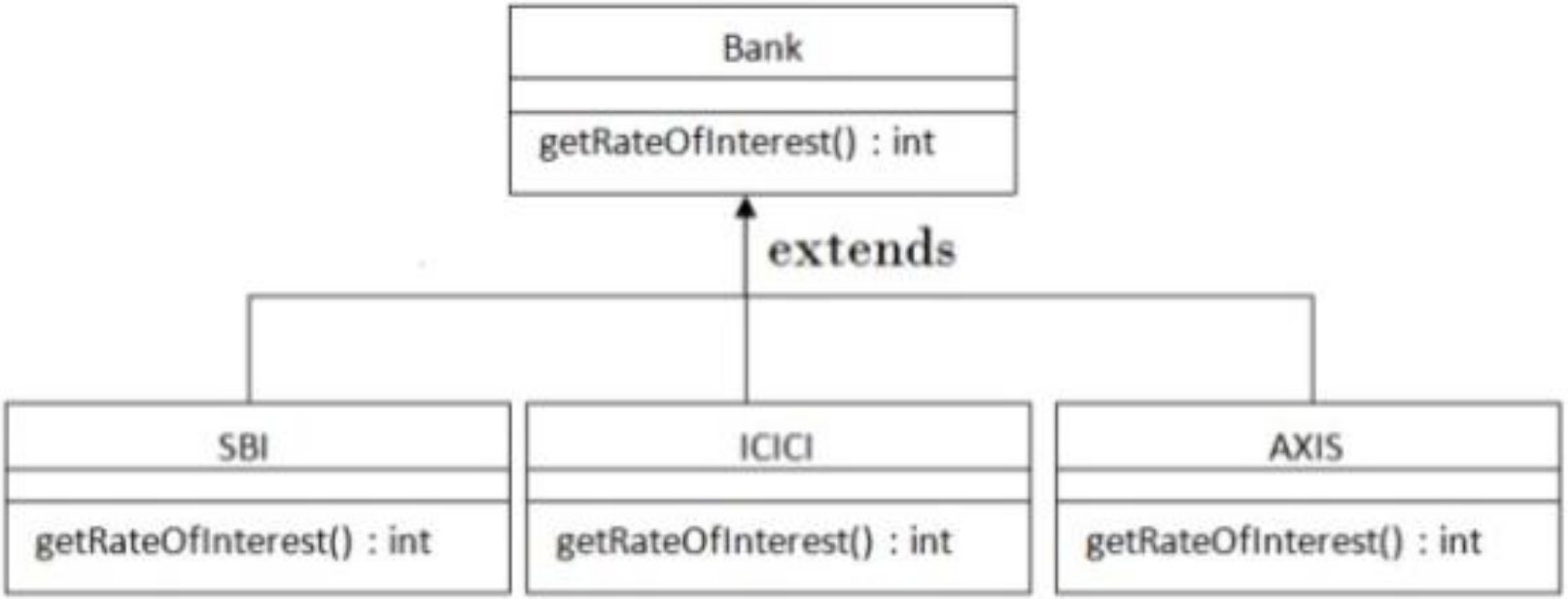
```
class Vehicle{
  void run(){
  System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{
  void run(){
  System.out.println("Bike is running safely");}

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

Method name are same

**Output is:**
Bike is running safely

# Real Example of Java Method Overriding

- Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



## Real Example of Java Method Overriding-1

```
class Bank{
  int getInterest(){return 0;}
}
class SBI extends Bank{
  int getInterest(){return 8;}
}

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

# Real Example of Java Method Overriding-2

```
class Test{
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.getInterest());
  System.out.println("ICICI Rate of Interest: "+i.getInterest());
  System.out.println("AXIS Rate of Interest: "+a.getInterest());
  }
}
```

**Output is:**

SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

# Method Overriding (Same Argument List)

```
public class Base {
   public int calculate(int num1,int num2) {
     return num1+num2;
   }
}
class Derived extends Base {
   public int calculate(int num1,int num2) {
     return num1*num2;
   }
   public static void main(String[] args) {
     Derived b1 = new Derived();
     int result = b1.calculate(10, 10);
     System.out.println("Result : " + result);
   }
}
```

Both Methods Having same number of Parameters List

Argument List is also same as Parameter List

**Output is:**
Result : 100

# Method Overriding in Java

- **Can we override static method?**
  - No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.
- **Why we cannot override static method?**
  - Because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.
- **Can we override java main method?**
  - No, because main is a static method.

## Java Access Modifiers With Method Overriding

| Access Level in Parent | Access Level in Child | Allowed ? |
|---|---|---|
| default | default | Allowed |
| default | public | Allowed |
| default | protected | Allowed |
| default | private | Not Allowed |
| public | default | Not Allowed |
| public | public | Allowed |
| public | protected | Not Allowed |
| public | private | Not Allowed |
| protected | default | Not Allowed |
| protected | public | Allowed |
| protected | protected | Allowed |
| protected | private | Not Allowed |

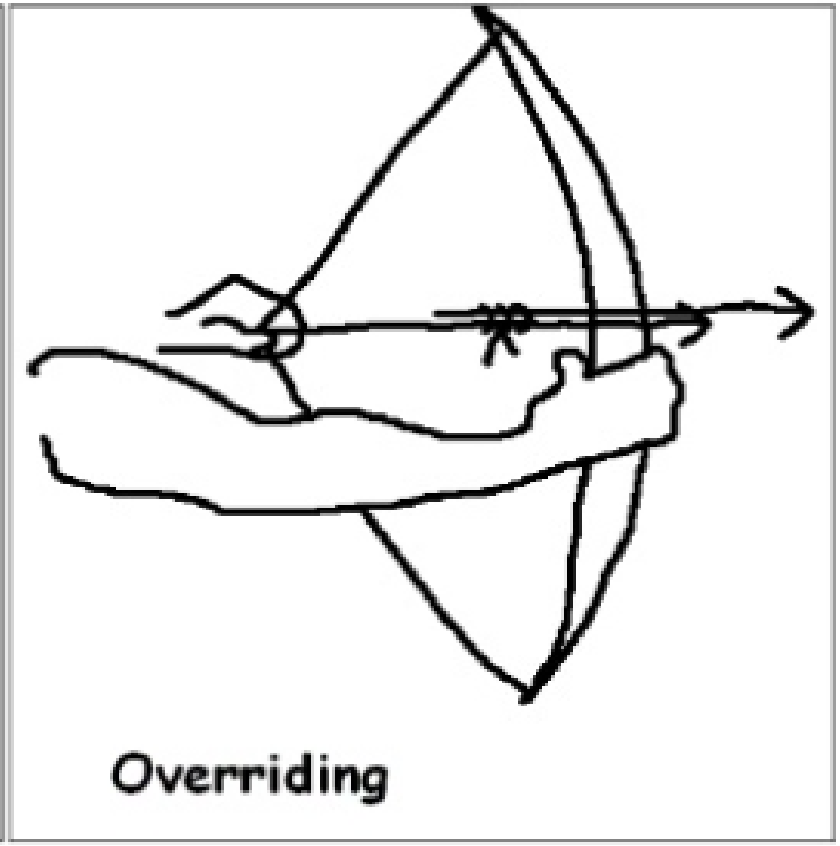
# Difference between Overloading and Overriding

| Overloading | Overriding |
|---|---|
| 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. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same.* |
| Method signature must be different. | Method signature must be same. |
| Private, static and final methods can be overloaded. | Private, static and final methods can not be override. |

# Difference between Overloading and Overriding

| Overloading | Overriding |
|---|---|
| Access modifiers point of view no restriction. | Access modifiers point of view not reduced scope of Access modifiers but increased. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| Overloading can be exhibited both are method and constructor level. | Overriding can be exhibited only at method label. |
| Overloading can be done at both static and non-static methods. | Overriding can be done only at non-static method. |
| For overloading methods return type may or may not be same. | For overriding method return type should be same. |

# Difference between Overloading and Overriding

## Java Method Overloading

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

## Java Method Overriding E

```
class Animal{
 void eat(){
 System.out.println("Eating");
  }
}
class Dog extends Animal{
 void eat(){
 System.out.println("Eating Bread");
  }
}
```

# super keyword in java

# super keyword in java

- Super keyword in java is a reference variable that is used to refer parent class object. Super is an implicit keyword create by JVM and supply each and every java program for performing important role in three places.
  - At variable level
  - At method level
  - At constructor level
- **Need of super keyword:**
- Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity. In order to differentiate between base class features and derived class features must be preceded by super keyword.

# super keyword in java

- **Usage of java super Keyword**
- We can user super keyword by using three ways:

- Accessing Instance Variable of Parent Class (Variable Level)
- Accessing Parent Class Method (Method Level)
- Accessing Parent Class Constructor (Constructor Level)

# super keyword in java

- **Super at Variable Level:**
- Whenever the derived class inherit base class data members there is a possibility that base class data member are similar to derived class data member and JVM gets an ambiguity.
- In order to differentiate between the data member of base class and derived class, in the context of derived class the base class data members must be preceded by super keyword.
- **Syntax**

```
super.baseclass datamember name
```

- if we are not writing super keyword before the base class data member name than it will be referred as current class data member name and base class data member are hidden in the context of derived class.

# Program without using super keyword

```
class Employee {
   float salary=10000;
}
class HR extends Employee
{
   float salary=20000;
   void display() {
    //print current class salary
    System.out.println("Salary is: "+salary);
   }
public static void main(String[] args) {
   HR obj=new HR();
   obj.display();
  }
}
```

Both Instance Variables Having same Name

**Output is:**
Salary is: 20000

## Program using super keyword at Variable level

```
class Employee {
   float salary=10000;
}
class HR extends Employee
{
   float salary=20000;
   void display() {
    //print base class salary
    System.out.println("Salary is: "+super.salary);
   }
public static void main(String[] args) {
   HR obj=new HR();
   obj.display();
  }
}
```

Using Super Keyword Here

**Output is:**
Salary is: 10000

## super keyword in java

- **Super at Method Level**
- The **super keyword** can also be used to invoke or call parent class method. It should be use in case of method overriding. In other word **super keyword** use when base class method name and derived class method name have same name.
- **Syntax**

```
Super.method-name();
```

# Example of super keyword at Method level

```
class Student{
  void message(){
  System.out.println("Good Morning Sir"); }
}
class Faculty extends Student{
  void message(){
  System.out.println("Good Morning Students");
  }
  void display(){
   message();  //will invoke or call current class message() method
   super.message();  //will invoke or call parent class message() method
  }
public static void main(String args[]) {
  Faculty s=new Faculty();
  s.display();
 }
}
```

**Output is:**
Good Morning Students
Good Morning Sir

## super keyword in java

- **Super at Constructor Level**
- The super keyword can also be used to invoke or call the parent class constructor. Constructor are calling from bottom to top and executing from top to bottom.
- To establish the connection between base class constructor and derived class constructors JVM provides two implicit methods they are:
- Super()
- Super(...)

# Super at Constructor Level

- **Super()**
- **Super()** It is used for calling super class default constructor from the context of derived class constructors.
- **Syntax**

```
super();
```

## Super keyword used to call base class Constructor

```
class Employee{
   Employee(){
   System.out.println("Employee class Constructor");
   }
}
class HR extends Employee{
   HR(){
  super(); //will invoke or call parent class constructor
   System.out.println("HR class Constructor");
   }
public static void main(String[] args){
   HR obj=new HR();
  }
}
```

**Output is:**

Employee class Constructor
HR class Constructor

## Explanation of Previous Program

- **Note:** super() is added in each class constructor automatically by compiler.


As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement. If you are creating your own constructor and we don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

# Super at Constructor Level

- **Super(...)**
- **Super(...)** It is used for calling super class parameterize constructor from the context of derived class constructor.
- **Syntax**

```
super(Arguments_List);
```

```
class Employee{
   int a;
  Employee(int a){
     this.a=a;  }
  public void getValue() {
     System.out.println("Value is:"+a);  }
}
class HR extends Employee{
  HR(int a){
  super(a);
  }
public static void main(String[] args){
  HR obj=new HR(10);
  obj.getValue();
 }
}
```

Constructor Parameters and Arguments List while calling the Constructor Must be Same

**Output is:**
Value is:10

# final keyword in java

# Final Keyword In Java

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context.
- Final can be:
  - variable
  - method
  - class
- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

# final variable

- **final variables:** If you use a variable as a final then the value of variable is become constant. We cannot change the value of a final variable once it is initialized.
- **Syntax**

```
final data_type variable_name= Some_Value;
```

- **Example**

```
final int data= 10;
```

## Example of final variable

```
class Test{

  final int MAX_VALUE=99;
  void myMethod(){
    MAX_VALUE=101;
  }
  public static void main(String args[]){
    Test obj=new  Test();
    obj.myMethod();
  }
}
```

We got a compilation error in this program because we tried to change the value of a final variable "MAX_VALUE".

Compiler Error

```
Exception in thread "main"
java.lang.RuntimeException: Uncompilable
source code - cannot assign a value to final
variable MAX_VALUE
```

# Final Keyword In Java

- **Final at method level** It makes a method final, meaning that sub classes can not override this method. The compiler checks and gives an error if you try to override the method.
- When we want to restrict overriding, then make a method as a final.
- **Syntax**

```
public final void fun()
{
.....
}
```

## Example of Final Keyword at Method Level

```
class Bike{
  final void run(){
  System.out.println("running");}
}
class Honda extends Bike{
  void run(){
   System.out.println("running safely with 100kmph");
}
  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

Final Method cannot override it

**Output is:**
Compile Time Error

# Final Keyword In Java

- **Final at class level** It makes a class final, meaning that the class can not be inheriting by other classes. When we want to restrict inheritance then make class as a final.
- **Syntax**

```
public final class A
{
.....
.....
}
public class B extends  A
{
// it gives an error, because we can not inherit final class
}
```

# Final Keyword In Java

- **Q) Is final method inherited?**
- **Ans)** Yes, final method is inherited but we cannot override it. For Example:

```
class Bike{
  final void run(){
  System.out.println("running...");}
}
class Honda extends Bike{
   public static void main(String args[]){
    new Honda().run();
   }
}
```

**Output is:**
running...

# Final Keyword In Java

- **blank or uninitialized final variable**
- A final variable that is not initialized at the time of declaration is known as blank final variable.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.
- It can be initialized only in constructor.
- **Example of blank final variable**

```
class Student{
  int id;
  String name;
  final String PAN_CARD_NUMBER;
...
}
```

# Final Keyword In Java

## initialize blank final variable

```
class Bike{
 final int speedlimit;    //blank final variable

 Bike(){
 speedlimit=50;
 System.out.println(speedlimit);
 }

 public static void main(String args[]){
   new Bike();
 }
}
```

**Output is:**
50

# Final Keyword In Java

- **What are the advantages of blank final variables?**
- Blank final variables allow the Programmer to make **immutable data types**. That is, final blank variables allow the class to declare immutable fields which are initialized at runtime by passing arguments to a constructor. Once assigned, the value cannot be changed **accidentally**.
- **Can we declare a constructor final?**
- No, it is not possible.

# Final Keyword In Java

- **static blank final variable**
- A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```
class A{
  static final int data;   //static blank final variable
  static{
  data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

**Output is:**
50

# final Keyword

| Final Entity | Description |
|---|---|
| Final Value | Final Value cannot be modified |
| Final Method | Final Method cannot be overriden |
| Final Class | Final Class cannot be inherited |

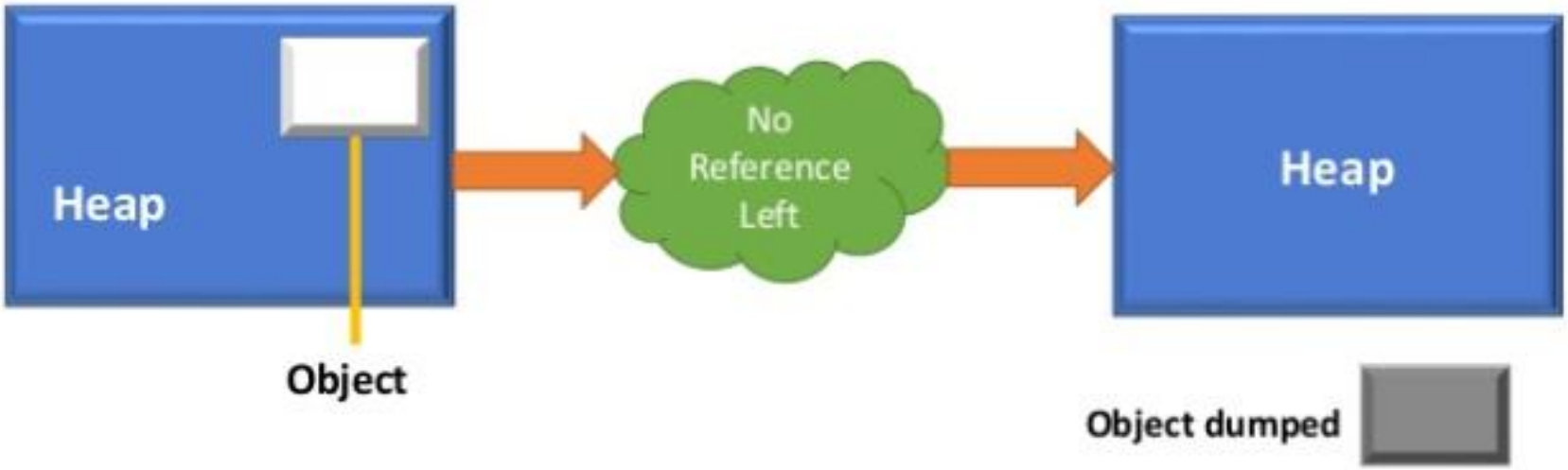
# Java Garbage Collection

# Java Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

# Java Garbage Collection

- In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called **Garbage Collection**.



# Java Garbage Collection

- **Can the Garbage Collection be forced explicitly ?**
- No, the Garbage Collection can not be forced explicitly. We may request JVM for garbage collection by calling System.gc() method. But This does not guarantee that JVM will perform the garbage collection.

- **Advantages of Garbage Collection**
- Programmer doesn't need to worry about dereferencing an object.
- It is done automatically by JVM.
- Increases memory efficiency and decreases the chances for memory leak.

# Java Garbage Collection

- **How can an object be unreferenced?**

- There are many ways:
  - By nulling the reference
  - By assigning a reference to another
  - By anonymous object etc.
- **1) By nulling a reference:**

```
Student  s=new Student();
//Assign null to object
s=null;
```

# Java Garbage Collection

- **2) By assigning a reference to another:**

```
Student s1=new Student();
Student s2=new Student();
s1=s2;     //now the first object referred by e1 is available for garbage collection
```

- **3) By anonymous object:**

```
new  Student();
```

# Java Garbage Collection

- **finalize() method**
- We have seen that a constructor method is used to initialize an object when it is declare. This process is known as initialization.
- Similarly, Java support a concept called finalization, which is just opposite to initialization. We know that Java run-time is an automatic garbage collection system. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or windows system fonts. The garbage collector cannot free these resources. In order to free these resources we must use finalizer method. This is similar to destructors in C++.
- The finalizer method is simply **finalizer()** and can be added to any class. Java call that method whenever it is about to reclaim the space for that object. The finalizer method should explicitly define the task to be performed.

# Java Garbage Collection

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

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

- The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).
- finalize() method is defined in **java.lang.Object** class, therefore it is available to all the classes.
- finalize() method is declare as protected inside Object class.

## Java Garbage Collection

- **gc() method**
- **gc()** method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection.
- It only request the JVM for garbage collection.
- This method is present in **System** and **Runtime** class.

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

## Example of garbage collection in java

```
public class Test{
  public void finalize(){
  System.out.println("Garbage Collected");
  }
 public static void main(String args[]){
  Test g1=new Test();
  Test g2=new Test();
  g1=null;
  g2=null;
  System.gc();
 }
}
```

**Output is:**

Garbage Collected

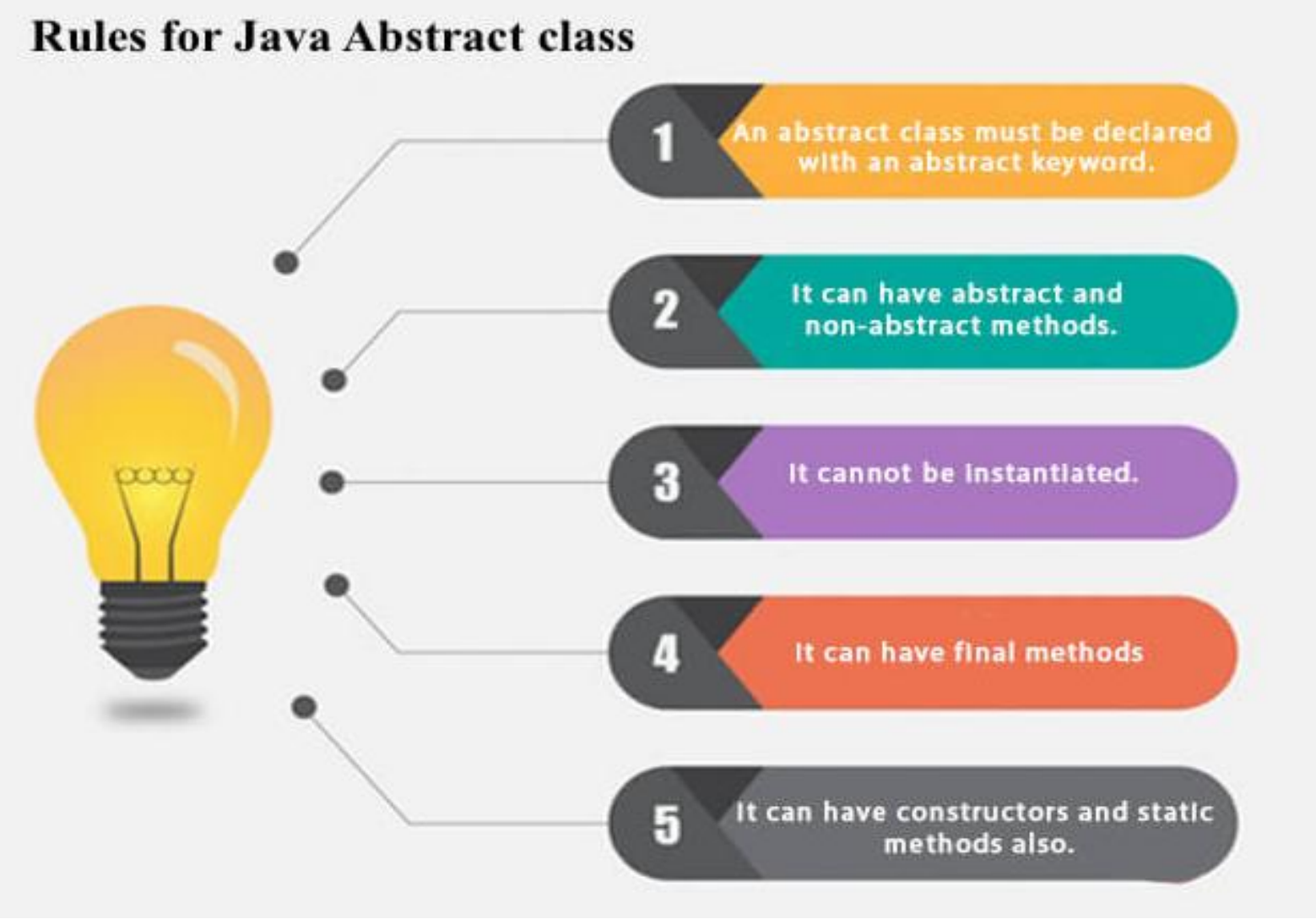
Garbage Collected

# Java Garbage Collection

- **Points to Note**
- Every class inherits the finalize() method from java.lang.Object.
- The finalize method is called by the garbage collector when it determines no more references to the object exist
- The finalize method of class Object performs no special action; it simply returns normally. Subclasses of Object may override this definition.
- If overriding finalize() it is a good programming practice to use a try-catch-finally statement and to always call super.finalize() as there is no concept of finalizer chaining.
- The finalize method is never invoked more than once by a Java virtual machine for any given object.

# Abstract class in Java

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- Before learning the Java abstract class, let's understand the abstraction in Java first.
- A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- Points to Remember
- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.


Rules for Java Abstract class
1
An abstract class must be declared with an abstract keyword.
2
It can have abstract and non-abstract methods.
3
It cannot be instantiated.
4
It can have final methods
5
It can have constructors and static methods also.

- 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).

```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");
  }
}
```

```java
class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

```
The pig says: wee wee
Zzz
```

Thank You ☺