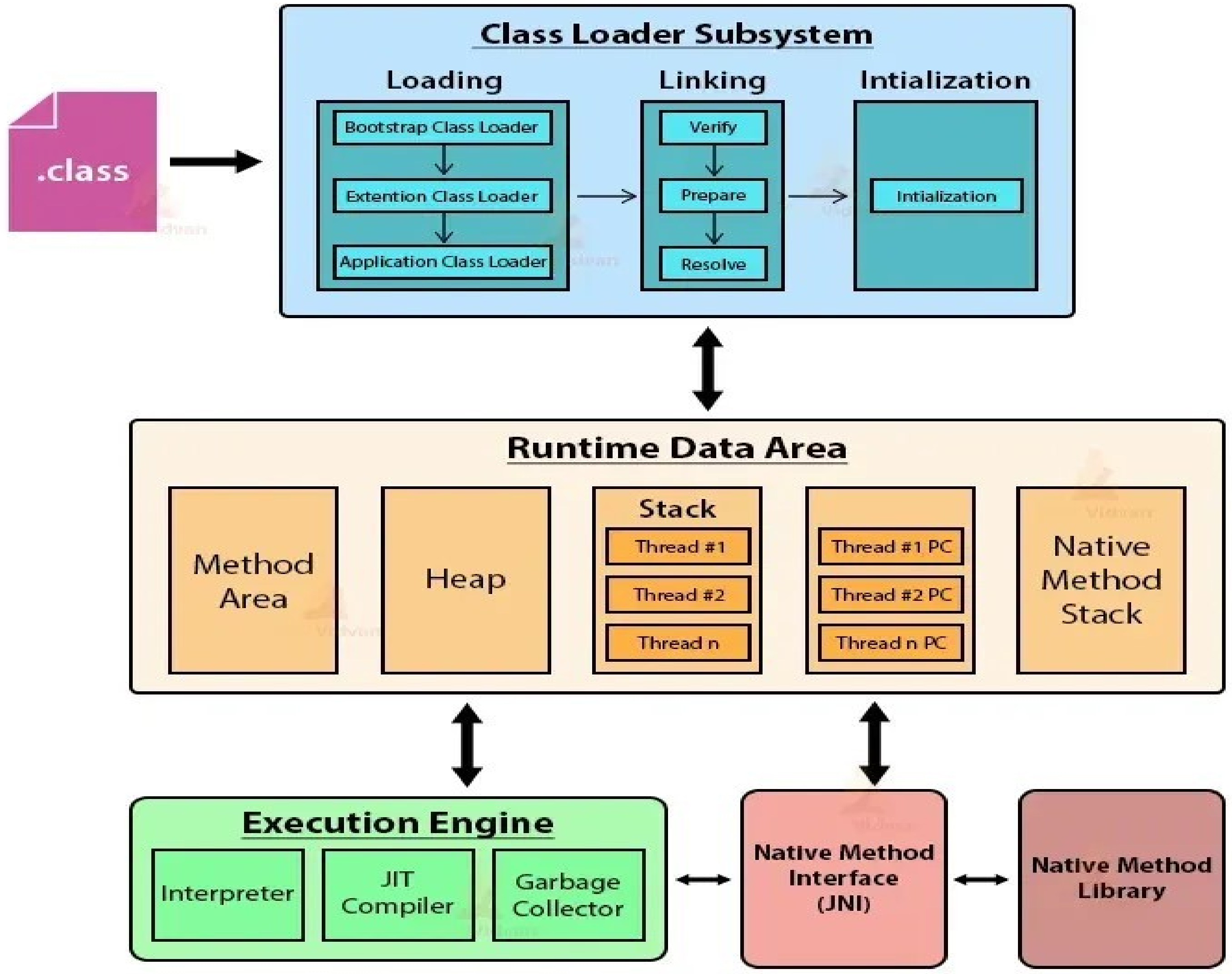
OOP using Java

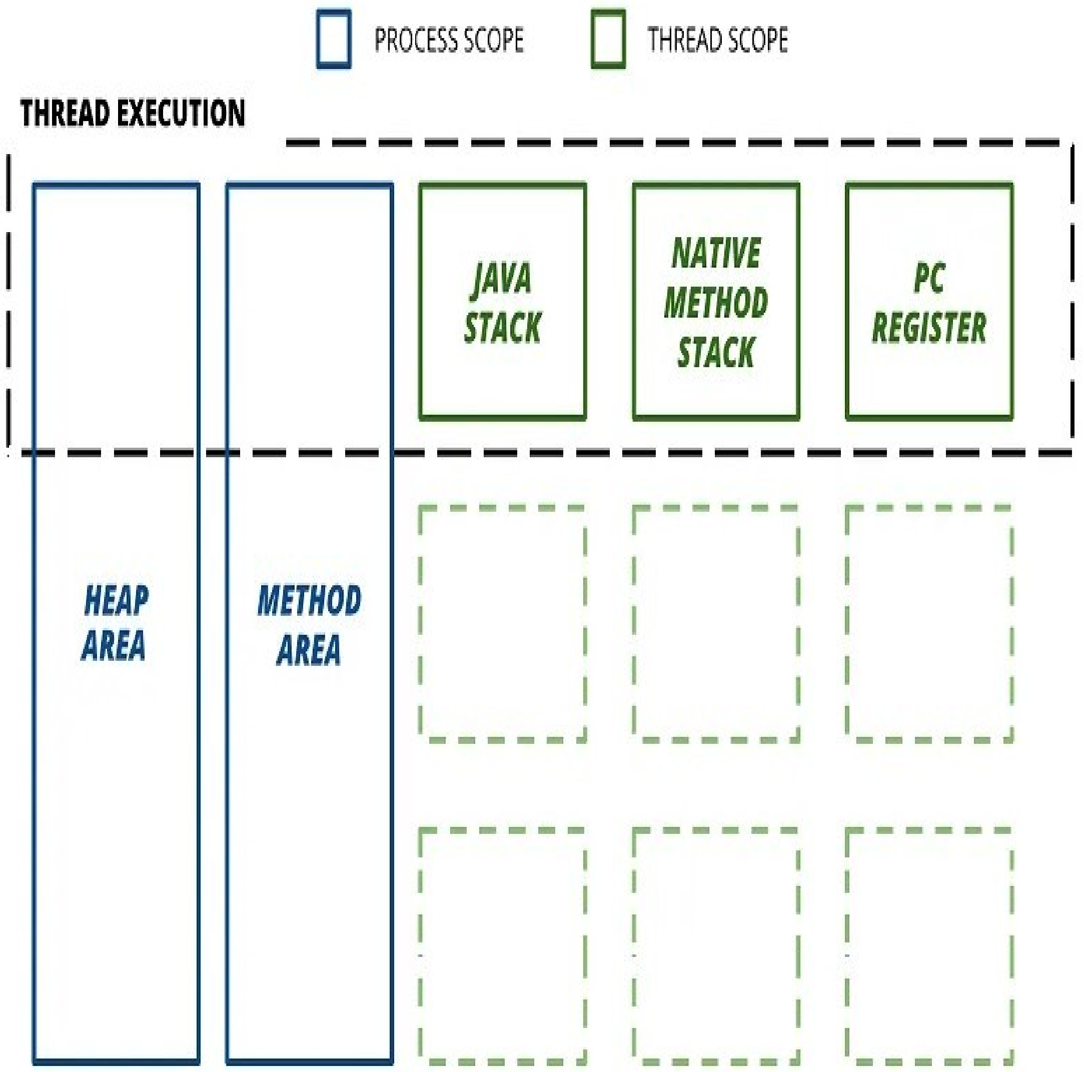
Trainer: Mr. Rohan Paramane

Agenda

- JVM Architecture
- Association
  - Aggegration
  - Composition
- Inheritance
- Types of inheritance
- Access Modifiers
- Upcasting & Dynamic Method Dispatch

JVM Architecture





JVM Architecture (ClassLoader)

- ClassLoader is responsible for loading class file into the memory area.
- The classloader is an abstract class, generates the data which constitutes a definition for the class using a class binary name which is constituted of the package name, and a class name.
- The ClassLoading mechanism consists of three main steps as follows.
  1. Loading
  2. Linking
  3. Initialization

- **Loading :**
- Whenever JVM loads a file, it will load and read,
  - Fully qualified class name
  - Variable information (instance variables)
  - Immediate parent information
  - Whether class or interface or enum

JVM Architecture

- There are three main mechanisms inside the JVM as shown in the above diagram.
  1. Class Loader
  2. Memory Area
  3. Execution Engine

- **ClassLoader :**
- Classloader consists of 3 class loaders as below -
  1. Bootstrap ClassLoader — Load classes from JRE/lib/rt.jar
  2. Extension ClassLoader — Load classes from JRE/lib/ext
  3. Application/System ClassLoader — Load classes from classpath, -cp

JVM Architecture (ClassLoader)

- **Linking :**
- This is the process of linking the data in the class file into the memory area. It begins with verification to ensure this class file and the compiler.
- Make sure this compiler is valid
- The class file has the correct formatting and the correct structure
- When the verification process is done, the next step is preparation.
- In this stage, all the variables are initialized with the default value.
- As an example, it will assign 0 for int variables, null for all objects, false for all boolean variables, etc.

- **Initialization :**
- This is the final stage of class loading. In this stage, the actual values are assigned to all static and instance variables.

JVM Architecture (Memory Area)

- This is the place data will store until the program is executed. It consists of 5 major components as follows.

1. Method Area :

- This is a shared resource (only 1 method area per JVM). Method area stores class-level information such as
  - ClassLoader reference
  - Run time constant pool
  - Constructor data — Per constructor: parameter types (in order)
  - Method data — Per method: name, return type, parameter types (in order), modifiers, attributes
  - Field data — Per field: name, type, modifiers, attributes

2. Heap Area

- All the objects and their corresponding instance variables are stored in the heap area. As an example, if your program consists of a class called “Employee” and you are declaring an instance as follows,
- Employee employee = new Employee();
- When the class is loaded, there is an instance of employee is created and it will be loaded into the Heap Area.

JVM Architecture (Memory Area)

3. Stack Area

- There are separate stack areas for each thread.
- The stack is responsible for hold the methods (method local variables, etc) and whenever we invoke a method, a new frame creates in the stack.
- These frames use Last-In-First-Out structure.
- They will be destroyed when the method execution is completed.

4. Program Counter (PC) Register

- PC Register keeps a record of the current instruction executing at any moment.
- That is like a pointer to the current instruction in a sequence of instructions in a program.
- Once the instruction is executed, the PC register is updated with the next instruction.
- If the currently executing method is ‘native’, then the value of the program counter register will be undefined.

5. Native Method Area

- A native method is a Java method that is implemented in another programming language, such as C or C++. This memory area is responsible to hold the information about these native methods.

JVM Architecture (Execution Engine)

- At the end of the loading and storing mechanisms, the final stage of JVM is executing the class file. Execution Engine has three main components.
    1. Interpreter
    2. JIT Compiler
    3. Garbage Collector

1. Interpreter :

- When the program is started to execute, the interpreter reads byte code line by line.
- This process will use some sort of dictionary that implies this kind of byte code should be converted into this kind of machine instructions.
- The main advantage of this process is the interpreter is very quick to load and fast execution.
- But whenever it executes the same code blocks (methods, loops, etc) again and again, the interpreter executes repeatedly.
- It means the interpreter cannot be optimized the run time by reducing the same code repeated execution.

JVM Architecture (Execution Engine)

2. JIT Compiler :

- Just In Time Compiler (JIT Compiler) is introduced to overcome the main disadvantage of the interpreter.
- That means the JIT compiler can remember code blocks that execute again and again.
- As an example, there is a class called “Employee” and getEmployeeID() method is declared in this class. If a program uses getEmployeeID() method 1000 times, each time it is executed by the interpreter.
- But the JIT compiler can identify those repeated code segments and they will be stored as native codes in the cache.
- Since it is stored, next time JIT compiler uses the native code which stored in the cache.

3. Garbage Collector (GC) :

- All the objects are stored in the heap area before JVM starts the execution.
- Since this area is limited, it is required to manage this area efficiently by removing the objects that are no longer in use.
- The process of removing unused objects from heap memory is known as Garbage collection and this is a part of memory management in Java.

Hireracy

- It is a major pillar of oops.
- Level / order / ranking of abstraction is called hierarchy.
- Main purpose of hierarchy is to achieve reusability.
- Advantages of code reusability
  1. We can reduce development time.
  2. We can reduce development cost.
  3. We can reduce developers effort.
- Types of hierarchy:
  1. Has-a / Part-of => Association
  2. Is-a / Kind-of => Inheritance /
  3. Use-a => Dependency
  4. Creates-a => Instantiation

Association

- If has-a relationship association is exist between the types then we should use
- Example
  1. Car has a engine
  2. Room has a chair

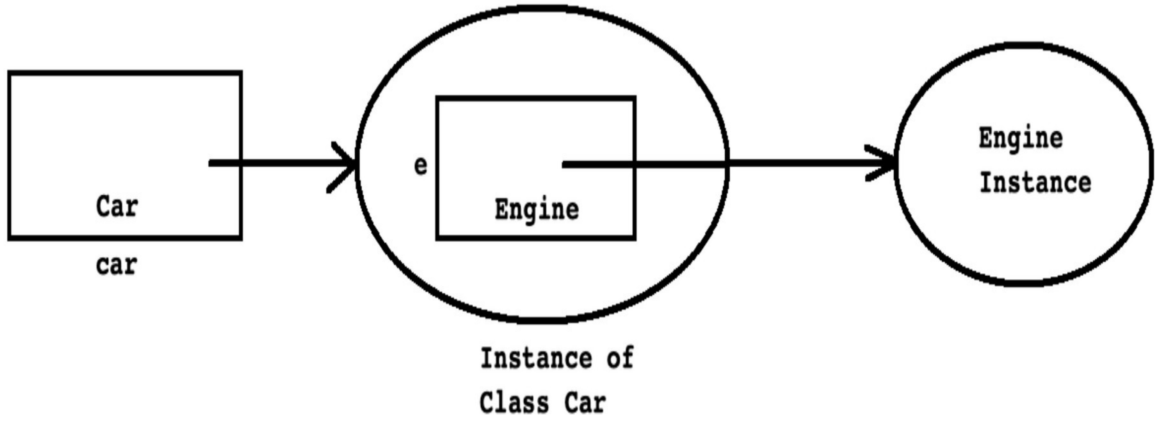
- Let us consider example of car and engine:
  - Car has a engine
  - Engine is part of Car.

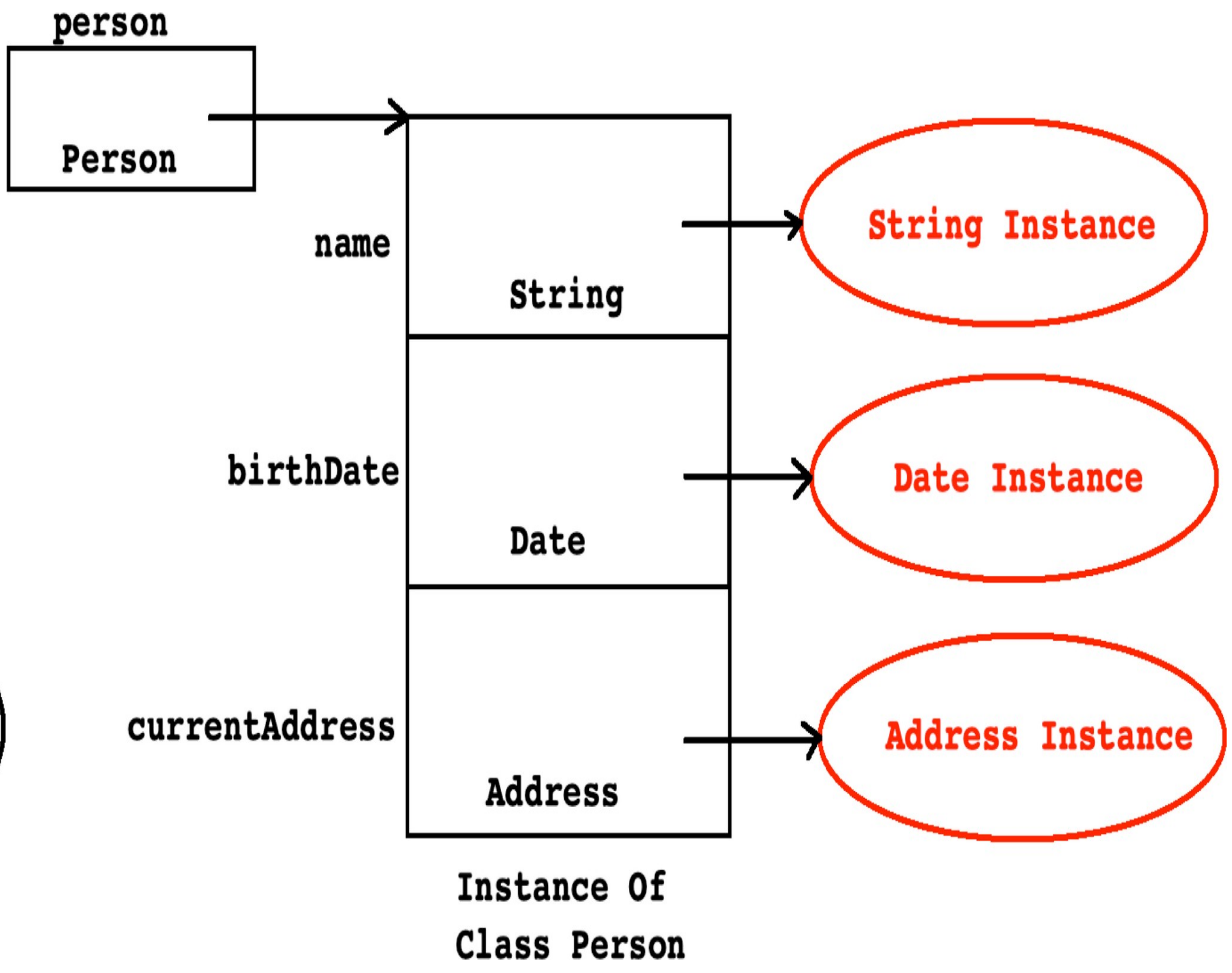
- If object/instance is a part/component of another instance then it is called as association.
- To implement association, we should declare instance of a class as a field inside another class.

Association

- Engine is part of Car

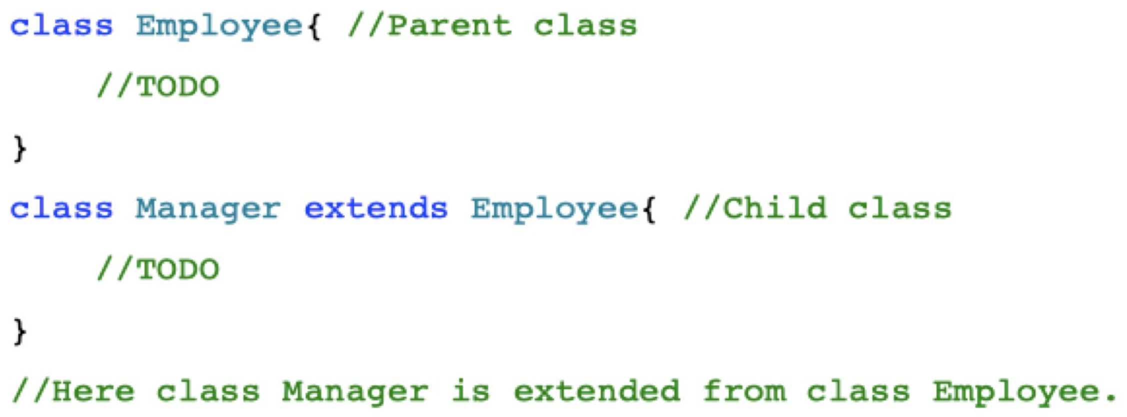
```
class Engine{
  //TODO
}
class Car{
  Engine e = new Engine( ); //Association
}
Car c = new Car( );
```





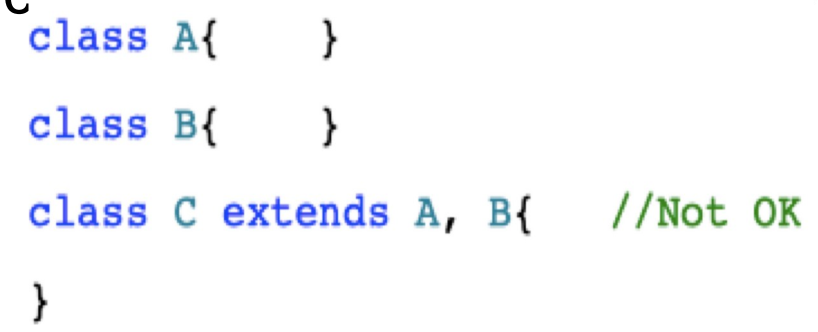
Inheritance

- If "is-a" relationship is exist between the types then we should use inheritance.
- Inheritance is also called as generalization.
- Example
    1. Manager is a employee
    2. Book is a product
    3. Triangle is a shape
    4. SavingAccount is a account.



```
class Employee{ //Parent class
    //TODO
}
class Manager extends Employee{ //Child class
    //TODO
}
//Here class Manager is extended from class Employee.
```

- If we want to implement inheritance then we should use extends keyword.
- In Java, parent class is called as super class and child class is called as sub class.
- Java do not support private and protected mode of inheritance
- In Java, class can extend only one class.
- In other words, multiple class inheritance is not allowed.
- Consider following code:



```
class A{    }
class B{    }
class C extends A, B{   //Not OK
}
```

Inheritance

- If we create instance of sub class then all the non static fields declared in super class and sub class get space inside it. In other words, non static fields of super class inherit into sub class.
- Static field do not get space inside instance. It is designed to share among all the instances of same class.
- Using sub class, we can access static fields declared in super class. In other words, static fields of super class inherit into sub class.
- All the fields of super class inherit into sub class but only non static fields gets space inside instance of sub class.
- Fields of sub class, do not inherit into super class. Hence if we create instance of super class then only non static fields declared in super class get space inside it.
- If we declare field in super class static then, all the instances of super class and sub class share single copy of static field declared in super class.
- We can call/invoke, non static method of super class on instance of sub class. In other words, non static method inherit into sub class.
- We can call static method of super class on sub class. In other words, static method inherit into sub class.
- Except constructor, all the methods of super class inherit into sub class.

Inheritance

- If we create instance of super class then only super class constructor gets called.
- But if we create instance of sub class then then JVM first give call to the super class constructor and then sub class constructor.
- From any constructor of sub class, by default, super class's parameterless constructor gets called.
- Using super statement, we can call any constructor of super class from constructor of sub class.
- Super statement, must be first statement inside constructor body.
- According to client's requirement, if implementation of super class is logically incomplete / partially complete then we should extend the class. In other words we should use inheritance.
- According to client's requirement, if implementation of super class method is logically incomplete / partially complete then we should redefine method inside sub class.
- Process of redefining method of super class inside sub class is called as **method overriding.**

Types of Class Inheritance

- During inheritance, if super type and sub type is class, then it is called as implementation inheritance.
- Types of inheritance
  1. Single Implementation Inheritance
  2. Multilevel Implementation Inheritance
  3. Hierarchical Implementation Inheritance

```
Single implementation Inheritance
class A{     }
class B extends A{ }     //OK
```

```
Hierarchical implementation Inheritance
class A{     }
class B extends A{ } //OK
class C extends A{ } //OK
```

```
Multiple implementation Inheritance
class A{     }
class B{     }
class C extends A, B{ } //Not OK
```

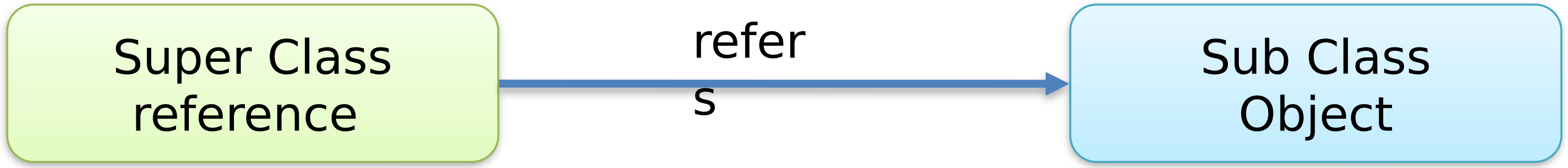
```
Multilevel implementation inheritance
class A{     }
class B extends A{ } //OK
class C extends B{ } //OK
```

Access Modifiers

	Same Package				Other Package	
Access Modifiers	**Same Class**	**Different Class**	**Sub Class**		**Different Class**	**Sub Class**
private	A	NA	NA		NA	NA
protected	A	A	A		NA	A
public	A	A	A		A	A
default	A	A	A		NA	NA

Upcasting

- When the reference variable of super class refers to the object of subclass, it is known as upcasting in java.
- when subclass object type is converted into superclass type upcasting.



Superclass s = new SubClass();

- Upcasting gives us the flexibility to access the parent class members, but it is not possible to access all the child class members using this feature.
- Instead of all the members, we can access some specified members of the child class.
- For instance, we can only access the overridden methods in the child class.

Dynamic Method Dispatch

- Process of redefining method of super class inside sub class is called as **method overriding.**
- When we keep instance of subclass in superclass reference we call it as **upcasting.**
- When such super class ref is used to invoke the overriding method, then the decision to send the method for execution is taken by JRE & not by compiler.
- In such case overriding form of the method(sub-class version) will be dispatched for execution.
- This is called as **Dynamic Method Dispatch.**
- Javac resolves the method binding by the type of the reference & JVM resolves the method binding by type of the object it's referring to.

Thank you!

Rohan Paramane

rohan.paramane@sunbeaminfo.com