ISO 9001 : 2008

CoreJava With SCJP and JVM Architecture

Volume-1B

COURSE MATERIAL

TM

NARESH technologies

Opp. Satyam Theatre,
Ameerpet, Hyderabad - 500 016.

SOFTWARE TRAINING & DEVELOPMENT

E-mail:info@nareshit.com www.nareshit.com

Ph:23746666, 23734842 Cell: 9000994007, 9000994008

An ISO 9001 : 2008 Certified Company

Index

Chapter 15

Classes and Types of Classes

- In this chapter, You will learn
  - Definition and need of a class
  - Types of classes
  - Class creation syntax
  - *concrete class* creation syntax and its usage
  - *final class* creation syntax and its usage
  - *abstract class* creation syntax and its usage
  - *interface* creation syntax and its usage
  - Difference between abstract class and interface
  - *enum* creation syntax and its usage
  - Allowed modifiers
  - Allowed members
- By the end of this chapter- you will learn defining, declaring, using classes, and its execution control flow.

- Method Hiding, overriding, and overloading, rules in overriding and overloading
- When a method is considered as overriding method?
- When we call overridden method from which class is it executed?
- Difference between hiding and overriding methods execution
- What do mean by
  - refVar.m1()
  - this.m1()
  - super.m1()
- When a method is considered as overloading method?
- Overloaded methods invocation process
- Constructor overloading
- Use of this() and super() calls and their rules
- Constructor Chaining

- Understanding Final method and final class
  - Definition and need of final method and class
  - When should we define a method or class as final
  - Rule on final method and class
  - Is final method inherited to subclass?
  - What is the difference between private method and final method?
  - Can we overload final method?
  - Can we override non-final method as final method?
  - Can we instantiate final class?
  - If a class is declared as final all its members are final?

- Understanding abstract method and abstract class
  - Definition and need of abstract class and method
  - Rule in creating abstract method, and abstract class
  - Can we instantiate abstract class?
  - Where should we provide implementation of abstract method
  - Rule in defining subclass from abstract class
  - Is it mandatory to override a concrete method?
  - How can we ensure a superclass method is not overridden in subclass?
  - How can we force subclass to override a superclass method?
  - What are the members allowed in abstract class?
  - Can we compile and execute abstract class?
  - Can we access abstract class non-static members using subclass object?
  - Does abstract class have constructor?
  - Can we declare abstract method as static, final, private?
  - What are the allowed modifiers in combination with abstract modifier?
  - Can an abstract method have return type other that void?
  - Can we declare concrete class as abstract?

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition and need of a class
- Types of classes
  - Interface
  - Abstract class
  - Concrete class
  - Final class
  - Enum

- Common interview questions on all above types of classes
  1. Syntax for creation
  2. Definition
  3. Allowed modifiers
  4. Allowed superclasses, superinterfaces
  5. Allowed members
  6. Object creation

- Understanding Concrete methods and concrete class
  - Definition and need of concrete method and class
  - What are the modifiers allowed for a concrete class?
  - What are the members allowed inside a concrete class?
  - Inheritance with concrete class
  - In how many ways we can access one class members from other class?
  - Compiler and JVM activities in compiling and executing a class with inheritance
  - Static control flow and instance control flow with inheritance
  - If we create subclass object, is superclass object also created?
  - What does a subclass object contains?
  - Who does initialize superclass fields in subclass object?
  - In subclass can we create variable and methods with super class variable names and methods, if so how can we differentiate them in subclass?
  - Can we define superclass members in subclass, if so how can we differentiate them?
  - Use of super keyword and its forms
  - Variable hiding, and difference in accessing variable using super class referenced variable and subclass referenced variable?
  - What do mean by
    - refVar.x
    - this.x
    - super.x
  - How many times a superclass is loaded into JVM?
  - How superclass static and non-static variables are shared to all its subclasses?

- Understanding interface
  - Definition and need of interface
  - What are the members allowed inside an interface?
  - Can we compile and execute interface?
  - Is interface derived from any other interface or class?
  - Can interface have concrete members?
  - Can interface have constructor?
  - Can we instantiate interface?
  - Is it mandatory to declare method as abstract in interface?
  - Can we create a variable without assigning a value in interface?
  - Can we declare interface method as private or protected?
  - Why interface members are by default public?
  - What is the rule in implementing interface methods in subclass?
  - Can we create an empty interface?
  - What is the marker interface, what are the predefined marker interfaces?
  - Can we create custom marker interface, if so what is the procedure?
  - What are the similarities and differences between interface and abstract classes

- Understanding enum
  - Definition and need of enum
  - A rule based on enum keyword
  - enum creation syntax
  - Can we create normal variables, methods, blocks and constructors in enum?
  - importance of ; in enum
  - Code changes done by compiler
  - What is the type of named constants in enum?
  - What is printed if we print named constant?
  - enum object memory structure
  - Does enum have constructor?
  - What is the difference between default constructor in class and enum?
  - Can we create constructor in an enum?
  - What is the default accessibility of enum constructor
  - How can we assign values to named constants?
  - Can we create object of enum like normal class?
  - Accessing enum members
  - What is the superclass of enum?
  - Can create enum from another enum?
  - What is the separator of named constants, is it "," or ";"?
  - When ";" is mandatory in enum?
  - Can we assign values to named constant directly?
  - What is the procedure to assign values to named constants?
  - Can we create inner enum?
  - Can we create enum inside a method?

Definition and need of class

A class is a top level block that is used for grouping variables and methods for developing logic.

Types of classes

In Java we have five types of classes

1. Interface
2. Abstract class
3. Concrete class
4. Final class
5. enum

Common interview questions on all above types of classes

1. Syntax
2. Definition
3. Allowed modifiers
4. Allowed superclasses, superinterfaces
5. Allowed members
6. Object creation

Syntax to create a class

```
[Accessibility Modifier] [Non-accessibility Modifier]
<class/interface/enum> <classname> [extends <superclass>
                                    implements <superinterfaces>]{

    ➢ static and non-static variables (fields)
    ➢ static and non-static blocks
    ➢ constructor
    ➢ static and non-static methods
    ➢ abstract methods
    ➢ innerclasses
}
```

Note: Things placed in bracket "[]" are optional. In an object creation process all above four classes has special role.

Definitions:

interface is a fully unimplemented class used for declaring a set of operations of an object. It contains only public static final variables and public abstract methods. It is created by using the keyword *interface*. It is always created as a root class in object creation process. We must design an object starts with interface if its operations have different implementations. *So interface tells what should implement but not how.*

For example: Vehicle, Animal objects creation must be started with interface. These two objects operations have different implementations based on different type of vehicles -like Bus, Car, Bike, etc... and different type of animals -like Lion, Tiger, Elephant, Rabbit, Dog, etc...

//Vehicle.java

```
interface Vehicle{
        public void engine();
        public void breaks();
}
```

abstract class a class that is declared as abstract using *abstract* modifier is called abstract class. It is a partially implemented class used for developing some of the operations of an object which are common for all next level subclasses. So it contains both abstract methods, concrete methods including variables, blocks and constructors. It is created by using keywords *abstract class.* It is always created as super class next to interface in object's inheritance hierarchy for implementing common operations from interface.

For example: In Vehicle object case

- We create Vehicle as interface and
- We create Bus, Car, Bike as abstract class, because we have different types of Buses, Cars, Bikes with some common operations and individual operations. For example: RedBus, Express, Volvo

//Bus.java

```
abstract class Bus implements Vehicle{
        public void breaks(){
                SopIn("Bus has two breaks");
        }
}
```

At Bus level the method breaks() can only be implemented because it is common for all Bus subclasses. The method engine() will be implemented in subclasses. So this class must be declared as abstract.

Concrete class is a fully implemented class used for implementing all operations of an object. It is created just using the keyword *class.* It contains only concrete methods including variables, blocks and constructor. It is always created as sub class next to abstract class or interface in object's inheritance hierarchy.

For example: In Vehicle object case

- We create *Vehicle* as interface with all its operations declarations
- We create *Bus, Car, Bike* as abstract class with all common operations implementation, and other individual operations are left as abstract
- *RedBus, Express, Volvo* are created as concrete class by implementing all other operations which are left as abstract in its superclass, in this case engine() method. Check it in the below two applications.

// RedBus.java

```
class RedBus extends Bus{
        public void engine(){
                SopIn("RedBus engine capacity is 40 kmph");
        }
}
```

// Volvo.java

```
class Volvo extends Bus{
        public void engine(){
                SopIn("Volvo engine capacity is 110 kmph");
        }
}
```

Develop a runtime polymorphic class to use all Vehicle operations

//Driver.java

```
class Driver{
        public void assignVehicle(Vechicle v){
                v.engine();
                v.breaks();
        }
}
```

//Depo.java

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

                Driver driver1 = new Driver();
                driver1.assignVehicle( new Volvo() );

                Driver driver2 = new Driver();
                driver2.assignVehicle( new RedBus() );
        }
}
```

Final class a concrete class which is declared as final is called final class. It does not allows subclasses. So it is the last subclass in an object's inheritance hierarchy.

Understanding concrete methods and concrete class

Q1) Definition and need of concrete method and class

A method that has body is called concrete method, and a class that has only concrete methods is called concrete class. It is used for implementing operations of an object.

Q2) What are the modifiers allowed for a concrete class?

public, final, abstract, strictfp

Q3) What are the members allowed inside a concrete class?

static and non-static variables and methods, blocks, constructors, main method are allowed

Q4) Inheritance with concrete class

A concrete class can be derived from another concrete class by using extends keyword.

For example

```
class Example{}
class Sample extends Example{}
```

Q5) In how many ways we can access one class members from other class?

In two ways we can access one class members from another class:

1. By using its class name or object name (with *HAS-A* relation)
2. By using inheritance – means – by using subclass name or object (with *IS-A* relation)

For example:

Given:

```
class Example{
        static int a = 10;
        int x = 20;
}
```

We can access *A1* class members in two ways

1. *From a normal class we can access*
   a. its static members by using class name and
   b. non-static members by using object

Check below code:

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

                Example e = new Example();
                System.out.println( e.x );
        }
}
```

2. *From a subclass*
   a. its static members by using subclass name and
   b. non-static members by using subclass object

Check below code:

```
class Sample extends Example{
        public static void main(String[] args){
                System.out.println( Sample.a );

                Sample s = new Sample();
                System.out.println( s.x );
        }
}
```

Q6) Compiler and JVM activities in compiling and executing a class with inheritance

Compiler Activities:

While compiling a class with inheritance relationship, compiler first compiles the super class and then it compiles the subclass. It gets super class name from extends keyword.

In finding method or variables definitions, compiler always first search in sub class. If it is not available in sub class, it searches in its immediate parent class, next in its grant parent class. If there also it is not available compiler throws CE: **"cannot find symbol"**.

JVM activities:

JVM also executes the invoked members from subclass, if that member definition is not available in subclass, JVM executes it from super class where ever it is appeared first.

When subclass is loaded its entire super classes are also loaded, and also when subclass object is created all its super class's non-static variables memory is created in subclass object. So the order of execution of static and non-static members with inheritance is:

- *Class loading* order is from *super class to subclass*
  - *SV and SB are identified and then executed from super class to subclass*

- *Object creation* is also starts from *super class to subclass*
  - *NSV, NSB, and invoked constructor are identified and executed from super class to subclass*

- *For executing method its definition searching* is starts from *sub class to super class*
  - *Invoked method is executed from sub class, if it is not defined in sub class, it is executed from super class,* but not from both.

How JVM can load super class when sub class is loaded?

- *Super class is loaded* by using *extends* keyword

How JVM can create super class non-static variables memory and initialize it in subclass object?

- using *super()*

Q10) Static control flow and instance control flow with inheritance

Static members control flow:

When subclass is loaded:

- JVM Identifies SVs, SBs, SMs, and MM from super class to subclass in the order they are defined from top to bottom, then
- It executes SVs, SBs from super class to subclass in the order they are defined.
- Finally it executes MM from current loaded subclass, if it does not contain main method, JVM executes it from super class, if there are also not found it throws RE: java.lang.NoSuchMethodError: main
- JVM executes the called SMs from current loaded subclass.

What is the output from the below program?

```
class A1 {

        static int a = m1();
        static int m1(){
                Sopln("A:a");
                return 10;
        }
        static{
                Sopln("A class is loaded");
        }
        public static void main(String[] args) {
                Sopln("A main");
        }
}
```

```
class B1 extends A1 {

        static int b = m2();
        static int m2(){
                Sopln("B:b");
                return 20;
        }
        static{
                Sopln("B class is loaded");
        }
        public static void main(String[] args) {
                Sopln("B main");
                Sopln("B main a: "+a);
                Sopln("B main b: "+b);
        }
}
```

>javac B1.java

>java B1 *Output:*

JVM Architecture:

What is the output from the below program?

```
class A2 {
        static int a = 10;

        static{
                Sopln("In A SB");
                Sopln("a: "+a);
                Sopln("b: "+b);
                Sopln("b: "+B2.b);
                Sopln("b: "+B2.getB());
        }
}
```

```
class B2 extends A2 {

        static int b = 20;
        static{
                Sopln("In B SB");
                Sopln("a: "+a);
                Sopln("b: "+ b);
                Sopln("b: "+ getB());
        }
        static int getB(){
                return b;
        }

        public static void main(String[] args) {
                Sopln("In B main");
                Sopln("a: "+a);
                Sopln("b: "+b);
        }
}
```

>javac B2.java
>java B2
Output:

JVM Architecture:

Non-Static members control flow with inheritance:

When subclass object is created:

- JVM Identifies NSVs, NSBs, NSMs from super class to subclass in the order they are defined from top to bottom, then
- It executes NSVs, NSBs and invoked constructor from super class to subclass.
- JVM executes the called NSMs from current subclass; if it is not available it is executed from super class.

What is the output from the below program?

```
class A3 {
        int x = 10;

        {
                Sopln("A NSB");
                Sopln("x: "+x);
        }

        A3(){
                Sopln("A Constructor");
                x = 5;
        }
}
```

```
class B3 extends A3{
        int y = 20;

        {
                Sopln("B NSB");
                Sopln("x: "+x);
                Sopln("y: "+y);
        }

        B3(){
                Sopln("B3 Constructor");
                y = 6;
        }
        public static void main(String[] args) {
                Sopln("B main");

                B3 b1 = new B3();
                Sopln("x: "+b1.x);
                Sopln("y: "+b1.y);
        }
}
```

>javac B3.java
>java B3
Output:

JVM Architecture:

Below program contains combination of static and non-static members. Find out output of the below program with JVM architecture

```
//Example.java
class Example {
        static int a = m1();

        static{
                System.out.println("Example SB");
        }

        int x = m2();

        {
                System.out.println("Example NSB");
        }

        Example(){
                System.out.println("Example Constructor");
        }

        static int m1(){
                System.out.println("Example Static Variable is created");
                return 10;
        }

        int m2(){
                System.out.println("Example non-static variable is created");
                return 20;
        }

        void abc(){
                System.out.println("Example abc");
        }

        void bbc(){
                System.out.println("Example bbc");
        }
}
```

```
//Sample.java
class Sample extends Example {
        static int b = m3();

        static {
                System.out.println("Sample SB");
        }

        int y = m4();

        {
                System.out.println("Sample NSB");
        }

        Sample(){
                System.out.println("Sample Constructor");
        }

        static int m3(){
                System.out.println("Sample Static variable is created");
                return 30;
        }

        int m4(){
                System.out.println("Sample Non Static variable is created");
                return 40;
        }

        void abc(){
                System.out.println("Sample abc");
        }

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

                Sample s = new Sample();
                s.abc();
                s.bbc();
        }
}
```

>java Sample **Output**

JVM architecture:

Use of super keyword:

The **super** is java keyword, used to access super class members and constructors from subclass members and constructors. It has two syntaxes they are:

1. super() - is used to call super class constructors from subclass constructors
2. super. - is used to call super class variables and methods from subclass members and constructors

Understanding "super()" working functionality

"super()" is used to invoke super class constructor from subclass constructors to provide memory for superclass fields in subclass object and further to initialize them with the initialization logic given by super class developer.

Who does place super() in all constructors?

Compiler places super() call in all constructors at the time of compilation provided there is no explicitly super() or this() call is placed by developer.

Rule on super()

It must be placed only in constructor as first statement.

Else it leads to CE: **"call to super must be first statement in constructor".**

Why it should be the first statement in subclass constructor?

Because to send control to super class for identifying and providing memory for non-static variables before subclass non-static variables identification and initialization.

Q) Invoking superclass constructor does it mean creating an object of superclass?

No. super class object is not created when sub class object is created, instead its non-static variables memory is provided in subclass object.

Q7) If we create subclass object, is superclass object also created?

No, super class object is not created instead its fields gets memory location in subclass object.

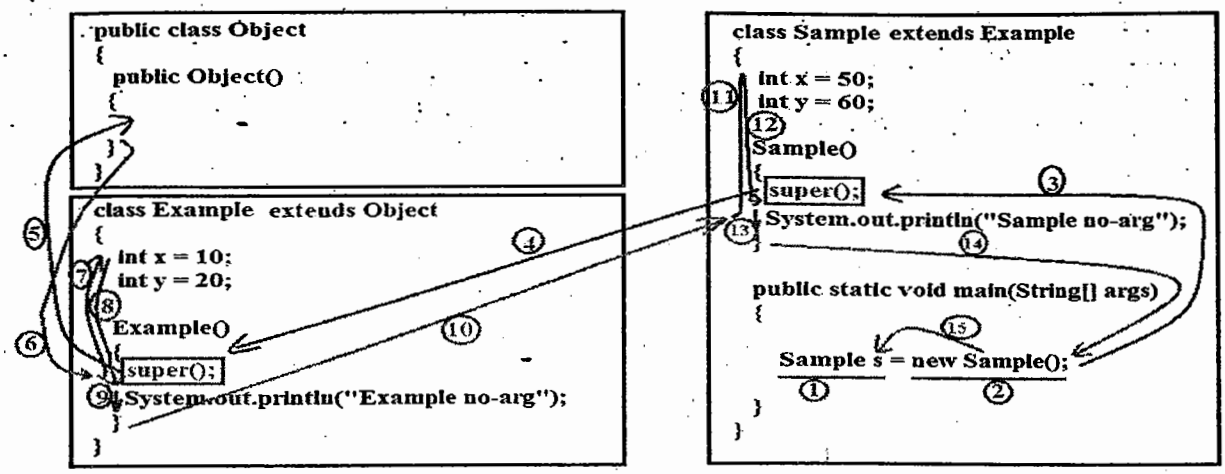
Q8) So what does an object contain?

An object has the fields of its own **class** plus all fields of its parent class, grandparent class, all the way up to the root **class** Object.

Q9) Who does initialize super class non-static variables?

The particular class constructor which is called by *super()*. It's necessary to initialize all fields; therefore all constructors must be called! The Java compiler automatically inserts the necessary constructor calls in the process of constructor chaining, or you can do it explicitly.

Below diagram shows control flow of sub class object creation.



Explain below program execution control flow with JVM architecture:

```
class A4                                {
        static int a = 10;
        int x = 30;

        A4(){

                Sopln("A class constructor");
                x = 5;
        }
}
```

```
class B4 extends A4{
        static int b = 20;
        int y = 40;

        B4(){

                Sopln("B class constructor");
                y = 6;
        }

        public static void main(String[] args) {
                B4 b = new B4();
                System.out.println(a);
                System.out.println(b);
                System.out.println(b.x);
                System.out.println(b.y);
        }
}
```

>javac B4.java

>java B4 *Output:*

JVM Architecture:

What is the output from below program?

```
class Example{
        Example(){
                System.out.println("Ex no-arg");
        }
        Example(int a){
                System.out.println("Ex int-arg");
        }
}
```

```
class Sample extends Example{
        Sample(){
                System.out.println("Sa no-arg");
        }
        Sample(int a){
                System.out.println("Sa int-arg");
        }
        public static void main(String[] args){
                Sample s1 = new Sample();
                Sample s2 = new Sample(10);
        }
}
```

O/P

Q) How can we initialize super class non-static variables by using parameterized constructor?

We must place "super" call explicitly with argument in subclass constructors. The argument type should be the constructor parameter type.

What is the output from the below program?

```
class Example{
        Example(){
                System.out.println("Ex no-arg");
        }
        Example(int a){
                System.out.println("Ex int-arg");
        }
}
```

```
class Sample extends Example{
        Sample(){
                System.out.println("Sa no-arg");
        }
        Sample(int a){
                super(50);
                System.out.println("Sa int-arg");
        }
        public static void main(String[] args){
                Sample s1 = new Sample();
                Sample s1 = new Sample(10);
        }
}
```

O/P

Q) When should developer place super() call explicitly?

Developer should place super() call for calling parameterized constructor in below two cases

1. If super class does not have no-arg constructor or it is declared as private.
2. If developer wants to initialize super class non-static variables with parameterized constructor.

In the above two cases developer should define constructor in sub class with explicit super() call by passing argument of type same as super class constructor parameter type.

Rule: Inheritance can only be implemented if super class has a visible constructor. It means non-private constructor if subclass is created in same package & only protected and public constructors in case subclass is created in other package.

Focus: So In solving inheritance bits first we must check what constructors are available in super class and which constructor's calls are available in subclass constructors with super keyword. If they are matched then only we must confirm inheritance is possible then after we must check the logic part of the bit. Else we must choose the option CE.

Identify whether inheritance is possible in below cases or not?

Case 1: Empty super class

```
class Example{}
class Sample extends Example{}
```

Case 2: Super class with explicit no-arg constructor

```
class Example{
        Example(){
                SopIn("Example no-arg");
        }
}
class Sample extends Example{}
```

Case 3: Super class with explicit parameterized constructor

```
class Example{
        Example(int a){
                SopIn("Example parameterized");
        }
}

class Sample extends Example{}
```

Case 4: Super and sub classes with explicit parameterized constructors

```
class Example{
        Example(int a){
                SopIn("Example parameterized");
        }
}
```

```
class Sample extends Example{
        Sample(int a){
            SopIn("Sample parameterized");
        }
        public static void main(String[] args){
                Sample s = new Sample(10);
        }
}
```

Case 5: Super class with private constructor

```
class Example{
        private Example(){
                Sopln("Example no-arg");
        }
}
class Sample extends Example{
        Sample(){
                Sopln("Sample no-arg");
        }
        public static void main(String[] args){
                Sample s = new Sample();
        }
}
```

Case 6: Super class with private constructor and visible parameterized constructor

```
class Example{
        private Example(){
                Sopln("Example no-arg");
        }
        Example(int a){
                Sopln("Example parameterized");
        }
}
class Sample extends Example{
        Sample(){

                Sopln("Sample parameterized");
        }
        public static void main(String[] args){
                Sample s = new Sample();
        }
}
```

Q11) Can we define superclass members in subclass?

Yes, we can define superclass members in subclass. When we call them in subclass, then subclass members are executed.

Variable hiding

Creating a variable in subclass with super class variable name is called variable hiding. Here subclass variable is hiding super class variable. It means when we call this variable using subclass referenced variable its value is read from subclass memory.

What is the output from the below program?

```
class A5 {

        static int a = 10;
        int x = 20;

        static void m1(){
                Sopln("A class m1");
        }
        void m2(){
                Sopln("A class m2");
        }
        void m3(){
                Sopln("A class m3");
        }
}
```

```
class B5 extends A5 {

        static int a = 50;
        int x = 60;

        static void m1(){
                Sopln("B class m1");
        }
        void m2(){
                Sopln("B class m2");
        }

        public static void main(String[] args) {
                Sopln("a: "+a);
                m1();

                B b = new B();
                Sopln("x: "+b.x);
                b.m2();
                b.m3();
        }
}
```

>javac B5.java
>java B5

Output:

a: 60
B class m1
x: 50
B class m2
A class m3

m3() method is not defined in B5 class, so it is executed from A5 class.

JVM architecture:

How can we differentiate superclass members from subclass members if both have same name? By using *super* keyword.

Understanding "super." working functionality

"super." is used to access super class variables and method from subclass members or constructors. It is mandatory to use "super." in accessing superclass members only if subclass has member with super class name.

For example: In the above class B5 we must use "super." to access a, x, and m1 and m2 methods from the class A5, the syntax is:

super.member;

Here *member* can either be an instance variable or a method. super keyword most useful to handle situations where the members of a subclass hide the members of a super class having the same name.

Rule on *super.*: We are not allowed to user super keyword in static members.
It leads to CE: "non-static variable super cannot be referenced from static context"
It can be used in all non-static members anywhere in the program.

What is the output from the below program, comment comiletime error?

```
class A5 {

        static int a = 10;
        int x = 20;

        static void m1(){
                Sopln("A class m1");
        }
        void m2(){
                Sopln("A class m2");
        }
        void m3(){
                Sopln("A class m3");
        }
}
```

```
class B5 extends A5 {
        static int a = 50;
        int x = 60;

        static void m1(){
                super.m1();
                Sopln("B class m1");
        }
        void m2(){
                Sopln("B class m2");
                super.m2();
        }
        void m4(){
                Sopln(super.a +"..."+ a);
                Sopln(super.x +"..."+ x);
                super.m1();
                m1();
                super.m2();
                m2();
        }
        public static void main(String[] args) {
                B5 b = new B5();
                b.m4();
        }
}
```

>javac B5.java
>java B5
Output:

What is the output from the below program?

```
class Example {
        int x = 10;
        int y = 20;

        void m1()
        {
                System.out.println("m1");
        }
}
```

```
class Sample extends Example{
        int x = 30;
        int y = 40;

        void m2(){
                System.out.println("x: "+x);
                System.out.println("y: "+y);

                System.out.println("x: "+super.x);
                System.out.println("y: "+super.y);
        }
        void m3(){
                int x = 50,  y = 60;

                System.out.println("local  x: "+x);
                System.out.println("local  y  "+y);

                System.out.println("subclass x: "+this.x);
                System.out.println("subclass y  "+this.y);

                System.out.println("super class x: "+super.x);
                System.out.println("super class y  "+super.y);
        }
        public static void main(String[] args){
                Sample s = new Sample();
                s.m2();
                s.m3();
        }
};
```

Similarities and differences between this and super keywords:

this (current class)	**super** (super class)
It is a keyword used to store current object reference	It is a keyword used to store super class non-static member's reference in sub class object.
Pre define instance variable used to hold current object reference	Pre defined instance variable used to hold super class memory reference through subclass object.
Used to separate state of multiple objects of same class and also used to separate local variables and class level variables in a non-static method if both has same name.	Used to separate super class and subclass members if both have same name.
It must be used explicitly if non-static variable and local variable or parameter name is same.	***It must be used explicitly if super class and sub class members have same names.***
Can't be referenced from static context. It can be printed, means can be called from sopln() from non-static context. System.out.println(this); System.out.println(Example.this);	Can't be referenced from static context. It can't be printed, means cannot be called from sopln() System.out.println(super); System.out.println(Example.super); System.out.println(this.super);

Why we cannot print *super* keyword like *this* keyword?

Because super class non-static member's memory does not have its own hashcode and also its memory is part of subclass object.

Q) How many referenced variables are available in a non-static method of a class?

A) Minimum three referenced varaibles we can have in a non-static method of a class.

They are:

- One explicit referenced variable created by developer
- Two implicit referenced variables *this* and *super* created by JVM. So we can say in every non-static method of a class by default two referenced variables are available.

Q) What is the datatype of *this* and *super* variables?

- datatype of this is current enclosing class in which non-static method is defined
- datatype of super is current enclosing class's super class in which it is defined

Identify datatype of *this* and *super* keywords in the below classes, any CE?

```
class A{
        int x = 10;
        void m1(){
          System.out.println(this.x);
          System.out.println(super.x);
        }
}
```

```
class B extends A{
        int x = 20;
        void m2(){
          System.out.println(this.x);
          System.out.println(super.x);
        }
}
```

Q12) What is the difference in accessing variable using superclass referenced variable and subclass referenced variable?

If we access a variable using superclass referenced variable it is accessed from superclass
If we access a variable using subclass referenced variable it is accessed from subclass

So we can differentiate superclass variables from subclass variables either by using

1. super keyword or
2. superclass referenced variable

What is the output from the below programs?

```
class A7 {
        int x = 10;
        int y = 20;
}

class B7 extends A7{

        int x = 30;
        int y = 40;

}
```

```
class TestAB7 {
        public static void main(String[] args) {

                B7 b = new B7();
                A7 a = new B7();

                System.out.println(b.x + " ... " + b.y);
                System.out.println(a.x + " ... " + a.y);

                b.x = 5;
                a.x = 6;

                System.out.println(b.x + " ... " + b.y);
                System.out.println(a.x + " ... " + a.y);
        }
}
```

```
class A8 {

        static int a = 10;
        int x = 20;

}
class B8 extends A8{
        static int a = 30;
        int x = 40;
}
```

```
class TestAB8_1 {
        public static void main(String[] args) {

                B8 b1 = new B8();
                A8 a1 = new B8();

                System.out.println(b1.a + "..." + b1.x);
                System.out.println(a1.a + "..." + a1.x);
                System.out.println(B8.a + "..." + A8.a);

                b1.a = 5;
                a1.a = 6;

                System.out.println(b1.a + "..." + b1.x);
                System.out.println(a1.a + "..." + a1.x);
                System.out.println(B8.a + "..." + A8.a);
        }
}
```

```
class TestAB8_2 {
        public static void main(String[] args) {

                B8 b = new B8();
                A8 a = b;

                System.out.println("a: "+b.a);
                System.out.println("a: "+a.a);

                System.out.println("x: "+b.x);
                System.out.println("x: "+a.x);

        }
}
```

Identify which class memory region variables are modified and read in above program, and finally what is output? In this example you will learn usage of *this.member*

```
class A9{
        int x = 10;
        int y = 20;

        void m1(){
                System.out.println(x);
                System.out.println(y);
        }
}
```

```
class B9 extends A9{

        int x = 30;
        int y = 40;

        void m2(){
                System.out.println(x);
                System.out.println(y);
        }

        public static void main(String[] args){
                B9 b1 = new B9();
                B9 b2 = new B9();

                b2.x = 50;
                b2.y = 60;

                b1.m1();
                b1.m2();

                System.out.println();
                b2.m1();
                b2.m2();
        }
}
```

Identify which class memory region variables are modified and read in above program, and finally what is output? In this example you will learn usage of *super.member*

```
class A10{
        int x = 10;
        int y = 20;
        void m1(){
                System.out.println(x);
                System.out.println(y);
        }
}

class B10 extends A10{

        int x = 30;
        int y = 40;
        void m2(){
                System.out.println(x);
                System.out.println(y);

                System.out.println(super.x);
                System.out.println(super.y);
        }
        void m3(){
                x = 50;
                y = 60;

                super.x = 70;
                super.y = 80;
        }
}
```

```
Class TestAB10{
        public static void main(String[] args){

                B10 b1 = new B10();
                A10 a1 = b1;

                B10 b2 = new B10();
                A10 a2 = b2;

                b1.m3();

                Sopln(b1.x +"..."+b1.y);
                Sopln(a1.x+"..."+ a1.y);

                Sopln(b2.x+"..."+b2.y);
                Sopln(a2.x+"..."+a2.y);
                System.out.println();

                b1.m1();  b2.m1();
                System.out.println();

                b1.m2();  b2.m2();
                System.out.println();
        }
}
```

How many times a superclass is loaded into JVM?

A class is loaded into JVM only once. So a superclass is loaded into JVM only once when its first subclass is loaded into JVM. When we load its next subclass superclass is not loaded again, only the current subclass is loaded.

How superclass static and non-static variables are shared to all its subclasses?

- static variable value is shared to all its subclasses and
- non-static variable is created in every subclass object separately

What is the output from the below program?

```
class A11 {
        static int a = 10;
        int x = 20;

        static {
                Sopl("A is loaded");
        }
}

class B11 extends A11 {
        static int b = 30;
        int y = 40;

        static {
                Sopln("B is loaded");
        }
}

class C11 extends A11{
        static int c = 50;
        int z = 60;

        static {
                Sopln("C is loaded");
        }
}
```

```
class TestABC11{

        static {
                Sopln("TestABC11 is loaded");
        }

        public static void main(String[] args) {

                B11 b1 = new B11();
                C11 c1 = new C11();

                b1.a = 15;
                b1.x = 16;

                System.out.println("b1.a: "+b1.a);
                System.out.println("c1.a: "+c1.a);

                System.out.println("b1.x: "+b1.x);
                System.out.println("c1.x: "+c1.x);

        }
}
```

Method Hiding, Overriding, and Overloading:

- Redefining super class *static method* in subclass with same prototype is called *"method hiding"*.
- Redefining super class *non-static method* in subclass with same prototype is called *"method overriding"*.
- Defining new method with the existed method name but with different parameters type | list | order is called "Method Overloading".

Rule: We cannot hide or override a method in the same class because we cannot create two methods with the same signature in a class. We can only do it in subclass. But we can overload method in the same class or in its subclass because overloading method has different signature as we are changing parameters.

Identity hiding, overriding and overloading methods from the below program

```
//A12.java
class A12 {
        static void m1(){
                Sopln("A m1");
        }
        void m2(){
                Sopln("A m2");
        }
        static void m3(){
                Sopln("A m3 no-arg");
        }
        int m3(String s){
                Sopln("A m3 String-arg");
                return 50;
        }
}
```

```
//B12.java
class B12 extends A12 {
        static void m1(){
                Sopln("B m1");
        }
        void m2(){
                Sopln("B m2");
        }
        void m3(float f, int x){
          Sopln("B m3 float, int arg");
        }
}
```

Execution control flow: Hiding & overloading methods are executed from referenced variable type class; whereas overriding method is executed from current object's class.

What is the output from the below program, find out is there any compile time errors?

```
//TestAB.java
class TestAB {
        public static void main(String[] args){
                B b = new B();
                b.m1();
                b.m2();
                b.m3();
                b.m3("abc");
                b.m3(45.3, 67);
                b.m3(45.3f, 67);

                A a = new B();
                a.m1();
                a.m2();
                a.m3();
                a.m3("abc");
                a.m3(45.3f, 67);
        }
}
```

Let us first understand Method overriding feature completely then we will learn overloading:

Method Overriding execution control flow:

In object oriented programming method overriding is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its super classes. The implementation in the subclass overrides (replaces) the implementation in the super class. So, *overridden method is always executed from the object whose object is stored in referenced variable.* Super class method is called overridden method, and subclass method is called overriding method. *What is the output from the below program?*

```
//Example.java
class Example
{
    void add(int a , int b)
    {
        Sopln("Example add(int , int): "+(a + b));
    }

    void sub(int a, int b)
    {
        Sopln("Example sub(int , int): "+(a - b));
    }
}
```

overridden method

```
//Sample.java
class Sample  extends Example
{
    void add(int a , int b)
    {
        Sopln("add(int , int) in Sample");
        Sopln("The addition of " + a + " and " + b + " is: " + (a + b));
    }

    public static void main(String[] args)
    {
        Sample s = new Sample();

        s.add(10, 20);

        s.sub(10, 20);
    }
}
```

Overriding method

When a method must be overridden?

If the super class method logic is not fulfilling subclass business requirements, sub class should override that method with required business logic. Usually in super class, methods are defined with generic logic which is common for all subclasses.

For instance, if we take Shape type of classes all shape classes, like Rectangle, Square, Circle etc ... should have methods area() and parameter(). Usually these methods are defined in super class, Shape, with generic logic but in subclasses Rectangle, Square, Circle etc ... these methods must have logic based on their specific logic. Hence in Shape subclasses these two methods must be overridden with subclass specific logic.

The *best example* that you knew already is *overriding toString()* method. When you print object, if you want to print your own message, then what you have to do? You must override it in your class to override the default logic that is given in Object class.

When a subclass method is treated as overriding method?

If a method in sub class contains signature same as super class *non private method*, sub class method is treated as overriding method and super class method is treated as overridden method. In the above example add(int , int) in Sample is overriding add(int, int) in Example, because both methods have same signature and it is not private in Example. So Example class add() method is called overridden method, Sample class add() method is called overriding method.

Rules in method overriding/hiding:

If a method is said to be overriding method, then it should satisfy below rules

1. retrun type should be same as super class method

```
class Example{
    void m1(){}
}
```

```
class Sample extends Example{
    void m1(){}
}
```

```
class Sample extends Example{
    int m1(){ return 10; }
}
```

2. static modifier should not be removed or added

```
class Example{
    void m1(){}
}
```

```
class Sample extends Example{
    void m1(){}
}
```

```
class Sample extends Example{
    static void m1(){}
}
```

```
class Example{
    static void m1(){}
}
```

```
class Sample extends Example{
    void m1(){}
}
```

```
class Sample extends Example{
    static void m1(){}
}
```

3. Accessibility modifier should be same as super class method or it can be increased, but should not be decreased.

```
class Example{
    void m1(){}
}
```

```
class Sample extends Example{
    private void m1(){}
}
```

```
class Sample extends Example{
    public void m1(){}
}
```

Focus: We cannot override *private method* because it is not inherited to subclass. If we override it in subclass no CE, and it is not considered as overriding method, it is just considered as subclass own method. So this method can have its own return type, NAM and AM.

Identify is below method is overriding method and is it there any compile time error.

```
class Example{
    private void m1(){}
}
```

```
class Sample extends Example{
    private void m1(){}
}
```

```
class Sample extends Example{
    static int m1(){return 10;}
}
```

The below table shows the allowed accessibility modifiers for the sub class overriding method based on super class overridden method's accessibility modifier.

Super class method	Sub class method
private	private, default, protected, public
default	default, protected, public
protected	protected, public
public	Public

4. throws clause should not be added with checked exception if super class method does not contain it. If super class method contain throws clause, sub class overriding method should contain throws clause with same exception class or its sub class or it can be removed. For more details on programming refer Exception Handling chapter.

When we call superclass method using subclass object what is its current object?
The same subclass object, then the method called in this method is executed from subclass whose object is stored in *this* variable if it is overridden, else its is executed from the current superclass. *What is the output from the below program?*

```
class A13 {
        void m1(){
                System.out.println("A m1");
        }
        void m2() {
                System.out.println("A m2");
                m1();
        }
}
```

```
class B13 extends A13{
        void m1(){
                System.out.println("B m1");
        }
        public static void main(String[] args) {
                B13 b = new B13();
                b.m1();  b.m2();

                A13 a = new B13();
                a.m1();  a.m2();
        }
}
```

Ouptput

```
class A13 {
        private void m1(){
                System.out.println("A m1");
        }
        void m2() {
                System.out.println("A m2");
                m1();
        }
}
```

```
class B13 extends A13{
        void m1(){
                System.out.println("B m1");
        }
        public static void main(String[] args) {
                B13 b = new B13();
                b.m1();  b.m2();

                A13 a = new B13();
                a.m1();  a.m2();
        }
}
```

Ouptput

In executing static method call we should not consider object type rather we must consider only referenced variable type. *What is the output from the below program?*

```
class A14 {
        static void m1(){
                System.out.println("A m1");
        }
        void m2() {
                System.out.println("A m2");
                m1();
        }
}
```

```
class B14 extends A14{
        static void m1(){
                System.out.println("B m1");
        }
        public static void main(String[] args) {
                B14 b = new B14();
                b.m1();  b.m2();

                A14 a = new B14();
                a.m1();  a.m2();
        }
}
```

Ouptput

How can we execute super class method if it is overridden in sub class?

Using ***super*** keyword. Below program explains method overriding and its accessing

```
class Example
{
    void m1()
    {
        System.out.println(" Example m1");
    }
    void m2()
    {
        System.out.println("Example m2");
    }
    void m3()
    {
        System.out.println("Example m3");
    }
}
```

```
class Sample extends Example
{
    void m1()
    {
        System.out.println(" Sample m1");
    }

    void m2()
    {
        super.m2();
        System.out.println(" Sample m2");
    }
    public static void main(String[] args)
    {
        Sample s = new Sample();
        s.m1();  s.m2();  s.m3();
    }
}
```

Q) How can we execute super class overridden static method from subclass?

By using super class name.

```
class Example{
    static void m1(){
        Sopln("Example m1");
    }
}
```

```
class Sample extends Example{
    static void m1(){
        Example.m1();
        Sopln("Sample m1");
    }
}
```

Q) If subclass does not have main method, will it leads to RE: *java.lang.NoSuchMethodError?*

No, if main method definition is also not available in any one of its super classes, then only above RE is thrown. What is the output of the below program:

```
class Example{
    public static void main(String[] args){
        Sopln("Example main");
    }
}
```

```
class Sample extends Example
{

}
```

Q) Can we override main method in subclass, if so are both methods executed?

We can override main method in subclass, and it is only executed from sub class. To execute super class main method, we must call it explicitly from subclass main method as shown below

```
class Example{
    public static void main(String[] args){
        Sopln("Example main");
    }
}
```

```
class Sample extends Example{
    public static void main(String[] args){
        Example.main(new String[0]);
        Sopln("Sample main");
    }
}
```

What is the output from the below program?

```
class A15{
        void m1(){
                System.out.println("A m1");
        }
        void m2(){
                System.out.println("A m2");
                m1();
        }
}

class B15 extends A15{
        void m1(){
                System.out.println("B m1");
        }
        void m3(){
                System.out.println("B m3");
                m1();
                super.m2();
        }
}

class C15 extends B15{
        void m2(){
                System.out.println("C m2");
                m4();
        }
}

class D15 extends C15{
        void m1(){
                System.out.println("D m1");
        }
        void m2(){
                System.out.println("D m2");
        }
        void m4(){
                System.out.println("D m4");
        }
        public static void main(String[] args){
                D15 d = new D15();
                d.m1();
                d.m2();
                d.m3();
                d.m4();
        }
}
```

```
class A16{
        static void m1(){
                System.out.println("A m1");
        }
        static void m2(){
                System.out.println("A m2");
                m1();
        }
}

class B16 extends A16{
        static void m1(){
                System.out.println("B m1");
        }
        void m3(){
                System.out.println("B m3");
                m1();
                super.m2();
        }
}

class C16 extends B16{
        static void m2(){
                System.out.println("C m2");
                m4();
        }
}

class D16 extends C16{
        static void m1(){
                System.out.println("D m1");
        }
        static void m2(){
                System.out.println("D m2");
        }
        static void m4(){
                System.out.println("D m4");
        }
        public static void main(String[] args){
                D16 d = new D16();
                d.m1();
                d.m2();
                d.m3();
                d.m4();
        }
}
```

```
//A17.java
class A17{

        static int a = 10;
        int x = 20;

        static void m1(){
                System.out.println("A m1");
        }
        void m2(){
                System.out.println("A m2");
        }

        void m3(){
                System.out.println("A m3");

                System.out.println("A a: "+a);
                System.out.println("A x: "+x);
                m1();
                m2();
        }
}

//B17.java
class B17 extends A17{

        static int a = 50;
        int x = 60;

        static void m1(){
                System.out.println("B m1");
        }
        void m2(){
                System.out.println("B m2");

                System.out.println("B a: "+a);
                System.out.println("B x: "+x);
        }
        void m4(){
                super.a = a - 10;
                super.x = x - 10;
        }
}
```

```
Class TestAB17{
        public static void main(String[] args){
                B17 b1 = new B17();
                B17 b2 = new B17();
                A17 a1 = new B17();

                b1.a = 15;  b1.x = 16;
                b2.a = 18;   b2.x = 19;

                b1.m4();  b2.m4();

                b1.m3();
                System.out.println();
                b2.m3();

                System.out.println();

                System.out.println(b1.a);
                System.out.println(a1.a);

                System.out.println(b1.x);
                System.out.println(a1.x);

        }
}
```

Understanding Method Overloading

It is a process of defining multiple methods with the same name but with different parameter types | list | order is called method overloading.

Q) When should we overload methods?

To execute same logic with different type of arguments we should overload methods.
For example to add two integers, two floats, and two Strings we should define three methods with same name as shown below:

```
class Addition{
  void add(int a1, int a2){
        System.out.println(a1 + a2);
  }
  void add(float f1, float f2) {
        System.out.println( f1 + f2);
  }
  void add(String s1, String s2){
        System.out.println(s1 + s2);
  }
}
```

Q) What is the advantage in overloading or what is the disadvantage if we define methods with different names?

A) If we overload method, user of our application gets comfort feeling in using method with an impression that he/she calling only one method by passing different type of values.

The best example for you is "println" method. It is overloaded method, not single method taking different type of values.

Q) When a method is considered as overloaded method?

If two methods have same method name, those methods are considered as overloaded methods. Then the rule we should check is both methods must have different parameter types | lists | order. But there is no rule on return type, Non-Accessibility Modifier, and Accessibility Modifier means overloading method can have its own return type, Non-Accessibility Modifier, and Accessibility Modifier because overloading methods are different methods.

Find out valid overloading methods from the below list of methods

```
void m1(){}
int m1(){return 50;}
int m1(String s){ return 50;}
static void m1(String s){}

void m1(int i1){}
void m1(int i2){}
String m1(float f1){return "";}
```

```
void m1(int i1, int i2){}
void m1(int i1, int i2, int i3){}

void m1(int i1, float f1){}
void m1(float f1, int i1){}

void m1(float f1, double d1)
void m1(float f1, long L1)
```

Q) Can we overload methods in same class?

Yes it is possible No CE and No RE. Methods can be overloaded in same or in super and sub classes, because *overloaded methods are different methods*. But we cannot override method in same class it leads to CE: "*method is already defined*" because *overriding methods are same methods with different implementation*.

Overloaded methods invocation control flow:

Compiler always checks for the called method definition in referenced variable type class with the given argument's type parameters. So in searching and executing a method definition we must consider both referenced variable type and argument type

1. referenced variable type for deciding from which class method should be bind
2. argument type for deciding which overloaded method should be bind

For example:

```
B b = new B();
A a = new B();

b.m1(50); => b.m1(int);
```

- In this method call, we should search m1() method definition in B class with int parameter in compilation.

```
a.m1(50); => a.m1(int);
```

- In this method call, we should search m1() method definition in A class with int parameter, not in B class even though object is B.

JVM always executes the called method definition from the class which is linked by compiler provided it is not overridden in current object class. If it is overridden method then only it is executed from current objects subclass.

This means:

In case of b.m1(50) method call, we should execute m1(int) method from B class because referenced variable type and object type both are same B type.

But in case of a.m1(50) method call, we should execute m1(int) method from B class if is overriding in B class else it means it is overloading or static we should execute it from A class because referenced variable type is A.

Note: Private methods are not overridden methods because they are not inherited to subclass. So, private methods are executed from referenced variable type class.

Also static methods are not overridden methods because they are hidden methods as they do not need object for executing their logic, so static methods are also executed from referenced variable type class.

It means

- If m1(int) is not overridden method or static method, overloaded method it is executed from A class.

- If m1(int) is a non-static method and if it is overridden in B class, it is executed from B class not from A class.

4 points to be considered to link and execute overloaded methods

Point #1: Overloading method executed based on given argument type

Point #2: *Widening, Autoboxing, var-arg*: If the given argument type parameter is not found then compiler search for widening type | AB type | Var-arg type parameter method of the given argument

Point #3: *Ambiguous error*: When same argument type parameter method is not found, if parameters of two overloaded methods are matched with given argument type then we get ambiguous error.

Point #4: *method overloading with inheritance*: with the given argument type parameter method we should first search in current referenced variable's class, if not found then we should search it in next parent class, grandparent class till root class Object, if it is not found we must repeat the searching with widening, AB, Var-arg, if this searching also failed then should throw CE: cannot find symbol

Let us understand all above 4 points practically

Point #1: Which overloaded form is executed when overloaded method is invoked?

Overloaded method is executed based on argument type passed in its invocation.

What is the output from the below program?

```
class Example
{
  void add(){
        System.out.println("no-arg add");
  }
  void add(int a) {
        System.out.println("int-arg add");
  }
  void add(String str){
        System.out.println("String-arg add");
  }
  /* int add(String s){
        System.out.println("String-arg add");
        return 10;
  }*/
```

```
  public static void main(String[] args)
  {
        Example e = new Example();

        e.add();
        e.add(10);
        e.add("abc");
   }
}
```

O/P:

```
===
no-arg add
int-arg add
String-arg add
```

We can pass argument in 5 ways

1. value directly
2. variable
3. value/variable with cast operator
4. expression
5. non-void method

1. **If we pass value directly we should bind and execute value type parameter method**
   Q) Find out which parameter method is executed for below method calls.

```
m1(50);      => executes: m1(int);
m1("abc");   => executes: m1(String);
```

2. **If we pass variable as argument we should bind and execute variable type parameter method but not its value type parameter method**
   Q) Find out which parameter method is executed for below method calls.

```
float f = 'a';
m1(f);  => executes: m1(float);

Object obj = "a";
m1(obj);     => executes: m1(Object)
m1("a");     => executes: m1(String)

String s = "a";
m1(s);  => executes: m1(String)
```

3. **If we use cast operator in argument we should bind and execute cast operator type parameter method.**
   Q) Find out which parameter method is executed for below method calls.

```
byte b = 'a';
m1(b);          => execute: m1(byte)
m1((char)b);    => execute: m1(char)
m1('a');        => execute: m1(char)
m1((byte)'a');  => execute: m1(byte)
m1((byte) true); => CE:inconvrtible types

Object obj = "a";
m1(obj)              => execute: m1(Object)
m1((String)obj)      => execute: m1(String)
m1((Integer)obj)     => RE: CCE
m1("a")              => execute: m1(String)
m1((Object)"a")      => execute: m1(Object)
```

4. **If we pass expression as argument we should bind and execute expression result type parameter method**
   Q) Find out which parameter method is executed for below method calls.

```
m1(10 + 20);      => execute: m1(int)
m1('a' + 'b');    => execute: m1(int)
m1("a" + "b");    => execute: m1(String)
```

5. **If we pass method calls as argument we should bind and execute method return type parameter method**
   Q) Find out which parameter method is executed for below method calls.

```
int m2(){ return 'a'; }
m1( m2() );   => execute: m1( int );

Object m3(){ return "a"; }
m1( m3() );    => execute: m1(Object)
```

What is the output from the below program?

```
class MOL1{
    void m1(int a ){
        System.out.println("int-arg");
    }
    void m1(char ch){
        System.out.println("char-arg");
    }
    public static void main(String[] args){
        MOL1 a1 = new MOL1();
        a1.m1(99);
        a1.m1('c');
        a1.m1((char)100);
        a1.m1((int)'d');

        System.out.println();
        int i1 = 97,
        int i2 = 'a';
        char ch1 = 98,
        char ch2 = 'b';

        a1.m1(i1);
        a1.m1(i2);
        a1.m1(ch1);
        a1.m1(ch2);

        System.out.println();
        a1.m1((char) i1);
        a1.m1((int) ch1);

        System.out.println();
        a1.m1( i1 + i2);
        a1.m1( ch1 + ch2);

        System.out.println();
        a1.m1( 10 + 'a');
        a1.m1( 'a' + 'b');
        a1.m1( (char)('a' + 'b'));
        a1.m1( (char)'a' + 'b');

        System.out.println();
        a1.m1( m2() );
        a1.m1( m3() );
    }

    static int m2(){
        return 'a';
    }
    static char m3(){
        return 97;
    }
}
```

We can also overload method with reference types.

Given:

```
class CalleeImpl{
        public void foo(Object o) {
                System.out.println("Object parameter");
        }
        public void foo(String s){
                System.out.println("String parameter");
        }
         public void foo(Integer i){
                System.out.println("Integer parameter");
         }
}
```

What is the output from the below program?

```
public class MOL2OverloadingMystery{
        public static void main(String[] bdsvf){
                CalleeImpl cl = new CalleeImpl();

                  Object ob1   = new Object();
                  Object ob2   = "HariKrishna";
                  Object ob3   = new Integer(7279);

                 cl.foo(ob1);
                 cl.foo(ob2);
                 cl.foo(ob3);

                 System.out.println();
                 cl.foo((String)ob2);
                 cl.foo((Integer)ob3);

                 System.out.println();
                 cl.foo((String)ob1);
                 cl.foo((Integer)ob1);
                 cl.foo((String)ob3);
                 cl.foo((Integer)ob2);
        }
}
```

Point #2: If passed argument type parameter method is not found, does compiler throw CE?

No, it does not throw compiler time error.

It searches overloaded method

1. First searches with widening type of the given argument, if not found
2. Second searches with Auto Boxing type of the given argument, if not found
3. Third searches with Var-arg type of the given argument, if this is also not found
4. At last Compiler throws CE: cannot find symbol

So the order of searching for the overloading method definition is

Same Type -> Widening -> AB -> Var-arg -> CE

Very important point to be remembered - If the method call is matched with widening | auto boxing | var-arg type parameter methods then compiler changes the given argument type to the matched methods parameter types. Then, JVM further executes the method definition based on the changed argument value type not with the actual given argument type in the source file. So we can say that JVM executes overloaded method definition which is linked by compiler not with given argument value type as it is appeared in source code.

What is the output from below program?

```
class MOL3 {

        static void m1(int a){
                System.out.println("int-arg");
        }

        static void m1(float f){
                System.out.println("float-arg");
        }

        public static void main(String[] args) {
                m1(10);

                m1('a');

                m1(50L);

                long L = 50;
                m1(L);

                m1(50.34);
        }
}
```

Find out CE in the below program?

```
class MOL4 {

        static void m1(byte b){
                System.out.println("byte-arg");
        }

        public static void main(String[] args) {
                m1(50);

                m1( (byte)50 );

                byte b = 50;
                m1( b );
        }
}
```

In below list which method calls leads to CE

void m1(float f) { }	void m1(char ch) { }	void m1(double d){ }
1. m1(10); 2. m1('a'); 3. m1(45L); 4. m1(4.5);	1. m1(10); 2. m1('a'); 3. m1("a"); 4. m1((char)10);	1. m1(10); 2. m1('a'); 3. m1(54L); 4. m1(46.043f); 5. m1(568.954); 6. m1(true);

What is the output from below program

```
class Example
{
  void m1(int i)
  {
    SopIn("int-arg");
  }

  void m1(byte b)
  {
    SopIn("byte-arg");
  }
}
```

```
class Sample
{
    public static void main(String[] args)
    {
       byte  b  = 10;
       short s  = 15;
       char  ch = 'a';
       int   i  = 20;

       Example e = new Example();

       e.m1(b); e.m1(s); e.m1(ch); e.m1(i);

       e.m1(10); e.m1(15); e.m1('a'); e.m1(20);

       e.m1((byte)10); e.m1(15); e.m1('a'); e.m1(20);

    }
}
```

Reference type widening:

When we invoke an overloaded method by passing an object first compiler checks for same argument type parameter method, if same type parameter method is not available then it search for its immediate super class parameter type method. It repeats this searching till it found java.lang.Object class parameter method. If no method is found with this passed object matched parameter, compiler throws CE: cannot find symbol.

The referenced type parameter method can be called by passing either

- same type of object or
- its subclass object or
- null

For example: assume "Sample" is a subclass of "Example"

Given:

void m1(Example e){ }

Find out CE from below list of method calls

```
m1( new Example() );
m1( new Sample() );
m1( new Test() );
m1( "abc" );
m1( null );
```

Q) What should be the method parameter to accept all types of objects as argument?

A) *java.lang.Object* class

Point #4: What does happened if the passed argument matched parameter is not found and if it is matched with widening parameters with different order?

A) It leads to CE: ambiguous error

Is below program compiled, if not what is the CE?

```
class Example
{
  void m1(int i , float f)
  {
    SopIn("int, float method");
  }

  void m1(float f , int i)
  {
    SopIn("float, int method");
  }
}
```

```
class Sample
{
  public static void main(String[] args)
  {
    Example e = new Example();

    e.m1(10, 20.345f);
    e.m1(20.345f, 10 );

    e.m1(10, 20);
  }
}
```

In the above program, ***e.m1(10, 20)*** method call leads to CE: ambiguous error, because the parameters of both methods are matched with this method call arguments, so compiler cannot decide which method definition must be bind for this method call, hence it throws CE.

Three important cases with referenced type overloaded methods

Case #1: If we overload methods with super and sub class types as parameter, and if the passed argument is matched, subclass parameter method is executed. In this case compiler and JVM give first priority to subclass method parameter method.

For example:

```
class A{
        void m1(Example e){
          SopIn("Example arg");
        }

        void m1(Sample s){
          SopIn("Sample arg");
        }

        public static void main(String[] args){
                A a = new A();
                a.m1(new Example());
                a.m1(new Sample());
                a.m1("abc");
                a.m1(null);

                Example e1 = new Example();
                Example e2 = new Sample();
                Sample  s1 = new Sample();
                Example e3 = null;
                Sample  s2 = null;

                a.m1(e1);     a.m1(e2);     a.m1(s1);
                a.m1(e3);     a.m1(s2);
        }
}
```

Case #2: When a method is overload with siblings, and if passed argument is matched with both parameters, compiler throws CE: ambiguous error.

For example:

```
class A{
        void m1(Example e){
          Sopln("Example arg");
        }

        void m1(Test s){
          Sopln("Test arg");
        }
                                        public static void main(String[] args){
                                                A a = new A();
                                                a.m1(new Example());
                                                a.m1(new Sample());
                                                a.m1("abc");
                                                a.m1(new Test());

                                                a.m1(null);
                                                a.m1( (Sample)null );
                                                a.m1( (Test)null );
                                        }
                                }
```

Case #3: When a method is overloaded with siblings parameters along with super and subclass parameters, if we pass null directly it leads CE: ambiguous error.

For example:

```
class A{
        void m1(Object obj){
                Sopln("Object arg");
        }

        void m1(Example e){
                Sopln("Example arg");
        }

        void m1(Sample s){
                Sopln("Sample arg");
        }

        void m1(Test t){
                Sopln("Test arg");
        }
                                        public static void main(String[] args){
                                                A a = new A();
                                                a.m1(new Example());
                                                a.m1(new Sample());
                                                a.m1("abc");
                                                a.m1(new Test());

                                                a.m1(null);
                                                a.m1((Sample) null);
                                                a.m1((Test) null);
                                        }
                                }
```

If "m1(Test t)" method definition is not available

- m1(new Test()) method call binds with m1(Object)
- m1(null) method call does not leads to CE, and is bind with m1(Sample s)

Q) What is the output from below programs?

Case #1: method overloading with siblings parameters

What is the output from below program?

```
class Test{}

class Example
{
  void m1(String s)
  {
    SopIn("String-arg");
  }

  void m1(Test t)
  {
    SopIn("Test-arg");
  }
}

class Sample
{
  public static void main(String[] args)
  {
    Example e = new Example();

    e.m1("abc");
    e.m1(new Test());

    e.m1(null);
  }
}
```

Case #2: method overloading with super and subclasses parameters

What is the output from below program?

```
class Test{}

class Ex
{
  void m1(Object obj)
  {
    SopIn("Obj-arg");
  }

  void m1(Test t)
  {
    SopIn("Test-arg");
  }
}

class Sa
{
  p s v main(S[] args)
  {
    Ex e = new Ex();

    e.m1("abc");
    e.m1(new Test());
    e.m1(new Object());

    e.m1(null);
  }
}
```

In first case, ***e.m1(null)*** method call leads to CE: ambiguous error, since the argument is matched with both methods parameter.

In the second case, this method call ***does not leads to CE,*** because subclass parameter type method is bind for this method call.

Point #5: Method overloading with inheritance

Methods can be overloaded in same class or in super and sub classes. In sub class if we have method with super class method name but with different parameters type | list | order then sub class method is called overloading method not overriding method.

Error: Methods cannot be overridden in same class they should be overridden in sub classes.

For example: Find out which is overloading and overriding?

```
class A{
        void m1(int a){}
}

class B extends A{
        void m1(float f){}
}
```

```
class A{
        void m1(int a){}
}

class B extends A{
        void m1(int a){}
}
```

Execution control flow with inheritance:

Compiler and JVM execute overloaded method with inheritance as shown below:

1. First search for the method with given argument type in subclass, if not found search it in superclass with same argument type, if not found
2. Then next it searches with Widening, if not found Auto boxing, if not found var-arg in subclass first, if not found in subclass, then searches in superclass, if method is not found with these options also
3. Finally compiler throws CE: cannot find error.

What is the output from below program?

```
class Example{
        void add(int a , int b)  {
                Sopln("Example int, int");
        }

        void add(String a, float b){
                Sopln("Example String, float");
        }

        int add(String s1 , String s2)  {
                Sopln("Example String, String");
                return 10;
        }
}
```

```
class Sample extends Example{
        void add(int x , int y)  {
                Sopln("Sample int, int");
        }
        float add(float a , int b){
                Sopln("Sample float, int");
                return a + b;
        }

        String add(String s1 , double d){
                Sopln("Sample String, double");
                return s1 + d;
        }
}
```

```
class Test{
        public static void main(String[] args) {
                Sample s = new Sample();

                s.add(10, 20);
                s.add("abc", 20);
                s.add("abc", "xyz");
                s.add("10", 20.0);
                s.add(10, 20.0f);
        }
}
```

Identify valid *overriding* and *overloading* methods from the below methods list

```
class Example
{
  void m1(int a , int b){}

  void m2(){}

  private void m3(String s){}

  protected void m4(double d, float f){}

  public void m5(){}

  public void m6(int x , float f){}

  static int m7(){return 10;}

  void m8(){}

 }
```

```
class Sample  extends Example
{
  void m1(int x , int y){}

  public void m2(){}

  public int m3(String s){return 10;}

  public float m4(double d, float f){return 20.34f;}

  void m5(){}

  public static void m6(float f , int x){}

  int m7(){return 30;}

  static void m8(){}

 }
```

What is the output from the below program?

```
class A{
        void m1(int a){
                System.out.println("A int-arg");
        }
};
class B extends A{
        void m1(float f){
                System.out.println("B float-arg");
        }
};

class MOL5WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1(50);
                b.m1('a');
                b.m1(50L);

                System.out.println();
                A a = new B();
                a.m1(50);
                a.m1('a');
                a.m1(50L);
        }
}
```

```
class A{
        void m1(float f){
                System.out.println("B int-arg");
        }
};
class B extends A{
        void m1(int a){
                System.out.println("A float-arg");
        }
};

class MOL6WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1(50);
                b.m1('a');
                b.m1(50L);

                System.out.println();
                A a = new B();
                a.m1(50);
                a.m1('a');
                a.m1(50L);
        }
}
```

```
class A{
        void m1(int a){
                System.out.println("A int-arg");
        }
};
class B extends A{
        void m1(float f){
                System.out.println("B float-arg");
        }
        void m1(char ch){
                System.out.println("B char-arg");
        }
};
```

```
class MOL5_1WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1(50);
                b.m1('a');
                b.m1(50L);

                System.out.println();
                A a = new B();
                a.m1(50);
                a.m1('a');
                //a.m1(50L);
        }
}
```

```
class A{
        void m1(int a){
                System.out.println("A int-arg");
        }
        void m1(char ch){
                System.out.println("A char-arg");
        }
};
class B extends A{
        void m1(float f){
                System.out.println("B float-arg");
        }
        void m1(char ch){
                System.out.println("B char-arg");
        }
};
```

```
class MOL5_2WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1(50);
                b.m1('a');
                b.m1(50L);

                System.out.println();
                A a = new B();
                a.m1(50);
                a.m1('a');
                a.m1(50L);
        }
}
```

```
class A{
        void m1(float f){
                System.out.println("A float-arg");
        }
};
class B extends A{
        void m1(int a){
                System.out.println("B int-arg");
        }
        void m1(long l){
                System.out.println("B long-arg");
        }
        void m1(float f){
                System.out.println("B float-arg");
        }
};
```

```
class MOL7WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1(50);
                b.m1('a');
                b.m1(50L);

                System.out.println();
                A a = new B();
                a.m1(50);
                a.m1('a');
                a.m1(50L);

        }
}
```

```
class A{
        void m1(Object obj){
                Sopln("A Object-arg");
        }
};
class B extends A{
        void m1(String s){
                Sopln("B String-arg");
        }
};
```

```
class MOL8WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1("a");
                b.m1(10);

                System.out.println();
                A a = new B();
                a.m1("a");
                a.m1(10);
        }
}
```

```
class A{
        void m1(String s){
                Sopln("A String-arg");
        }
};
class B extends A{
        void m1(Object obj){
                Sopln("B Object-arg");
        }
};
```

```
class MOL9WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1("a");
                b.m1(10);

                System.out.println();
                A a = new B();
                a.m1("a");
                a.m1(10);
        }
}
```

```
class A{
        void m1(String s){
                Sopln("A String-arg");
        }
        void m1(Integer io){
                Sopln("A Integer-arg");
        }
};
class B extends A{
        void m1(Object obj){
                Sopln("B Object-arg");
        }
        void m1(String s){
                Sopln("B String-arg");
        }
        void m1(Integer io){
                Sopln("B Integer-arg");
        }
};
```

```
class MOL10WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1("a");
                b.m1(10);

                System.out.println();
                A a = new B();
                a.m1("a");
                a.m1(10);
        }
}
```

```
class A{
        void m1(Object obj){
                Sopln("A Object-arg");
        }
};
class B extends A{
        void m1(Object obj){
                Sopln("B Object-arg");
        }
        void m1(String s){
                Sopln("B String-arg");
        }
};
```

```
class MOL11WithInheritance {
        public static void main(String[] args) {
                B b = new B();
                b.m1("a");
                b.m1(10);

                System.out.println();
                A a = new B();
                a.m1("a");
                a.m1(10);
        }
}
```

What is the output from the below program?

```
class A {
        int x = m1();
        int m1(){
                System.out.println("A m1");
                return 50;
        }
}

class B extends A {
        int y = m1();
        int m1(){
                System.out.println("B m1");
                return 60;
        }
}
class TestAB {
        public static void main(String[] args) {
                B b = new B();

                System.out.println("x: "+b.x);
                System.out.println("y: "+b.y);
        }
}
```

Output

```
class A {
        int x = m1();
        static int m1(){
                System.out.println("A m1");
                return 50;
        }
}

class B extends A {
        int y = m1();
        static int m1(){
                System.out.println("B m1");
                return 60;
        }
}
class TestAB {
        public static void main(String[] args) {
                B b = new B();

                System.out.println("x: "+b.x);
                System.out.println("y: "+b.y);
        }
}
```

Output

Constructor overloading

Defining multiple constructors in a class with different parameters type | list | order is called constructor overloading. Like methods constructors can also be overloaded, but they cannot be overridden because constructors are not inherited, also we cannot create constructor in subclass with super class name.

Execution: To execute overloaded constructor we must pass that constructor parameter type argument in the object creation statement as show below.

For example

```
class Example {
        Example(){
            SopIn("Ex No-arg constructor);
        }
        Example(int a){
            SopIn("Ex int-arg constructor);
        }
        Example(String str){
            SopIn("Ex String-arg constructor);
        }
```

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

      Example e1 = new Example();

      Example e2 = new Example(10);

      Example e3 = new Example("abc");
  }
}
```

Why should we overload constructor?

In the below two cases we should overload constructor

1. To define different object initialization logic
2. To execute the same initialization logic by taking input values in different types.

For example:

If we want to initialize an object with static values or with user given values we should overload constructors they are no-arg & parameterized constructors

If we want to initialize an object variable say "int x" by taking its value either as int type or as Integer type we must overload constructors with int parameter and integer parameter.

```
class Example {
        int x;

        Example(){
            x = 10;
        }
        Example(int a){
            x = a;
        }
}
```

```
class Example {
        int x;

        Example(int a){
            x = a;
        }
        Example(String s){
            x = Integer.parseInt(s);
        }
}
```

How can we call super class overloaded constructors from subclass constructors?
By using *super()* . We must place super() call in subclass constructor by passing argument same as super class overloaded constructor parameter type

What is the output from the below program?

```
class Sample extends Example{
        Sample(){
           Sopln("Sa No-arg constructor);
        }
        Sample(String str) {
           super(10);
            Sopln("Sa String-arg constructor);
        }
```

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

        Sample s1 = new Sample();

        Sample s2 = new Sample("abc");
    }
}
```

Understanding this():
How can we call overloaded **constructors from other constructors of same class without creating other new object?**
We must use this() call by passing the overloaded constructor parameter type argument

What is the output from the below program?

```
class Example {

        Example(){
                this(10);
                Sopln("No-arg constructor);
        }

        Example(int a){
                this("abc");
                Sopln("int-arg constructor);
        }

        Example(String str){

           Sopln("String-arg constructor);
        }
```

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

      Example e1 = new Example();

      Example e2 = new Example(10);

      Example e3 = new Example("abc");
   }
}
```

Output

Note: In the above program only one object is created per new keyword execution even though three constructors are executed. So in turn non-static variables and non-static blocks are executed only once per object creation.

this() execution control flow:

Due to this() call, control is sent to respective parameter constructor that is matched with the given argument. Control is sent to another constructor without creating object and also without executing current constructor logic. Object is created from the constructor that has *super()* call, because to create sub class object super class memory should be created and should be included in subclass object. After *super()* execution sub class object is initialized and NSBs are executed, and then *constructors logic is executed in reverse constructors call stack without creating new objects.*

Q) Can we have both super() and this() calls in the same constructor?

No, *we* should not place super() and this() calls in the same constructor, because in subclass object superclass memory is included more than once. So compiler does not place *super()* call in constructor if has *this()* call.

Q) How can we prove object is created or not created?

Print some message from *non-static block*.

Below program explains this() execution control flow, this program also proves that object is created only once per new keyword even though more than one constructor is executed.

```
class Example{
        int x = m1();

        {
                Sopln("NSB");
        }

        int m1(){
                Sopln("m1 : x");
                retrun 10;
        }
        Example(){
                this(10);
                x = 50;
                Sopln("No-arg constructor");
        }
        Example(int a){
                this("abc");
                x = 60;
                Sopln("int-arg constructor");
        }
        Example(String str){
                x = 70;
                Sopln("String-arg constructor");
        }
```

```
  public static void main(String[] args)
  {
      Example e1 = new Example();
          Sopln("e1.x: "+e1.x);

      Example e2 = new Example(10);
          Sopln("e2.x: "+e2.x);

      Example e3 = new Example("abc");
          Sopln("e3.x: "+e3.x);

  }
}
```

Constructors chaining

Calling one constructor from other constructor by using *super()* and *this()* is called constructor chaining.

Subclass constructors are chained with superclass constructors by using *super()* and Subclass overloaded constructors are chained by using *this()*.

The below program demonstrates the process of constructor chaining using this() and super() methods. What is the output from the below program?

```
class SuperClass
{
        SuperClass()
        {
                this(10);
                 Sopln("superclass no-arg");
        }

        SuperClass(int a)
        {
                this("abc");
                Sopln ("superclass int-arg");
        }

        SuperClass(String s)
        {
                Sopln ("superclass String-arg");
        }
}
```

```
class SubClass extends SuperClass
{
        SubClass()
        {
                this(10);
                Sopln("subclass no-arg");
        }

        SubClass(int a)
        {
                 this("abc");
                Sopln ("subclass int-arg");
        }

        SubClass(String str)
        {
                Sopln ("subclass string-arg");
        }
}
```

```
class ThisSuperDemo{
        public static void main(String[] args) {
                new SubClass();

                System.out.println();
                new SubClass(10);

                System.out.println();
                new SubClass("abc");
        }
}
```

Rules on this():

Rule #1: Like super(), *this()* call also must be placed as first statement only in the constructor, else it leads to CE: **"call to this must be first statement in a constructor"**

```
class ThisRule1
{
        ThisRule1()
        {
                this(10);
                System.out.println("No-arg");
        }

        ThisRule1(int a)
        {
                System.out.println("int-arg");
                this("abc");
        }

        ThisRule1(String s)
        {
                System.out.println("String-arg");
        }

        void m1()
        {
                this("abc");
                System.out.println("m1");
        }
}
```

Rule #2: *this()* and *super()* calls cannot be placed in the same constructor, because both must be placed as first statement in constructor.

```
class ThisRule2
{
        ThisRule2(){
                this(10);
                System.out.println("No-arg");
        }
        ThisRule2(int a){
                this("abc");
                super();
                System.out.println("int-arg");
                }

        ThisRule2(String s){
                System.out.println("String-arg");
        }
}
```

Hence, based on current rule we can conclude that compiler places super() call only if there is no this() or super() calls explicitly.

Rule #3: Also, multiple this() or super() calls cannot be placed in same constructor, violation leads to same CE.

```
class ThisRule3 {
        ThisRule3(){
                this("abc");
                this("abc");

                System.out.println("No-arg");
        }
```

```
        ThisRule3(String s){
                super();
                super();

                System.out.println("String-arg");
        }
}
```

Rule #4: In an object creation we are not allowed to call same constructor more than once, violation leads to CE: ***"recursive constructor invocation"***. But we are allowed to call it again in another object creation

Case #1: So this() cannot be placed in all constructors, at least one constructor must be left with super() call in order to call super class constructor when subclass object is created. Else it leads to CE: ***"recursive constructor invocation"***.

```
class ThisRule4{
        ThisRule4(){
                this(10);
                System.out.println("No-arg");
        }
        ThisRule4(int a){
                this("abc");
                System.out.println("int-arg");
        }
        ThisRule4(String s){
                this();
                System.out.println("String-arg");
        }
}
```

```
class ThisRule4{
        ThisRule4(){
                this(10);
                System.out.println("No-arg");
        }
        ThisRule4(int a){
                this("abc");
                System.out.println("int-arg");
        }
        ThisRule4(String s){

                System.out.println("String-arg");
        }
}
```

Case #2: Also in a constructor its own this() call cannot be placed, violation leads to CE: ***"recursive constructor invocation"***.

```
class ThisRule5{
        ThisRule5(){
                this();
                System.out.println("No-arg");
        }
}
```

Special case: But it is possible to create object in a constructor with same constructor. In this case compiler cannot identify recursive constructor call, but it is identified by JVM and terminates program execution with RE: *"java.lang.StackOverflowError"*.

```
class ThisRule5{
        ThisRule5(){
                new ThisRule5();
                System.out.println("No-arg");
        }
}
```

Q) Why we cannot place this() call in the same constructor Or why we can't call a constructor *recursively* with this()?

A) The Rule is: in an object creation we are not allowed to call same constructor more than once. It can be called again in another object creation. This the reason compiler allows calling constructor in creating new object in the same constructor but not with this() as this() does not create new object.

Understanding Final methods and Final classes

Final Method

A method which has final keyword in its definition is called final method: Rule is it cannot be overridden in subclass. But it is inherited to subclass, and we can invoke it to execute its logic.

When a method should be defined as final?

If we do not want allow subclass to override super class method and to ensure that all sub classes uses the same super class method logic then that method should be declare as final method.

Rule: Final method cannot be overridden in sub class, violation leads to CE

Below program explains final method

```
class Example
{
    void m1()
    {
        System.out.println(" Example m1");
    }

    final void m2()
    {
        System.out.println("Example m2");
    }

    void m3() {
        System.out.println("Example m3");
    }
}
```

```
class Sample extends Example
{
    void m1()
    {
        System.out.println(" Sample m1");
    }

    void m2()
    {
        System.out.println(" Sample m2");

    }
    public static void main(String[] args)
    {
        Sample s = new Sample();
        s.m1();   s.m2();   s.m3();
    }
}
```

What is the difference between private and final methods?

private method is not inherited, where as final method is inherited but cannot be overridden. So private method cannot be called from subclass, where as final method can be called from subclass. The same private method can be defined in subclass, but it does not lead to CE.

Can we *overload* final method in subclass?

Yes, we can overload final method in subclass

```
class Ex{
        final void m1(int x){}
}
```

```
class Sa extends Ex{
        void m1(float f){}
}
```

Can we *override* non final method as final in subclass?

Yes, it is possible

```
class Ex{
        void m1(){}
}
```

```
class Sa extends Ex{
        final void m1(){}
}
```

Can we declare main method as final?
Yes, it is possible.

For example:

```
class Example{
        public static final void main(String[] args){
                System.out.println("Ex main");
        }
}
```

Then, can subclass of *Example* have its own main method (can we override main method)?
No, it is not possible because final method cannot be overridden.

Then how can we execute subclass static and non-static members by subclass itself?
By using *static* block

```
class Sample extends Example{

        static{
                System.out.println ("SB");
                m1();

                Sample s = new Sample();
                s.m2();
        }

        static void m1(){
                System.out.println("m1");
        }

        void m2(){
                System.out.println("m2");
        }

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

O/P:

```
>java Sample
SB
m1
m2
Ex main
```

Final class

A class which has final keyword in its definition is called final class and it cannot be extended.

When a class should be defined as final?

In the below situations we must define class as final

- If we do not want override all methods our class in subclass
- If we do not want extend our class functionality.

Rule: Sub class cannot be defined from final class, violation leads to CE:

```
final class Example{
        int x = 10;
        final int y = 20;

        void m1() {
          System.out.println(" Example m1");
        }
}
```

```
class Sample extends Example
{
}
```

Q) If class is declared as final, are all its members final?

A) No, final class members are not final until they are declared as final members explicitly by using final keyword.

Q) Can we instantiate final class?

Yes, we can instantiate final class means we can create object for final class.

Find out compile time error in the below program.

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

                Example e = new Example();
                e.x = 50;
                e.y = 60;

                e.m1();
        }
}
```

Understanding Abstract methods and Abstract classes

A method that does not have body is called abstract method, and the class that is declared as abstract using *abstract* keyword is called abstract class. If a class contains abstract method, it must be declared as abstract.

Procedure to create Abstract Methods in a class

Create a method without body. A method created without body is called abstract method. Rule of abstract method is it should contain abstract keyword and should ends with semicolon.

Ex: abstract void m1();

Q) Why method should has abstract keyword if it does not have body?

In a class we are allowed ONLY to define methods with body. Since we are changing its default property – means removing its body – it must has abstract keyword in its prototype.

We should follow below rules in defining abstract method and abstract class.

Rule: 1 If method does not have body it should be declared as abstract using abstract modifier keyword else leads to CE

```
class Example
{
   void m1();   X  CE: missing method body, or declare abstract
}
```

Rule: 2 If a class have abstract method, it should be declared as abstract class using abstract keyword else it leads to CE

```
class Example   X  CE: Example is not abstract and does not override abstract method m1() in Example
{
   abstract void m1();   ✓
}
```

```
abstract class Example   ✓
{
      abstract void m1();   ✓
}
```

Rule 3: If class is declared as abstract it can not be instantiated violation to CE

```
abstract class Example
{
   abstract void m1();
   p s v main(String[] args)
   {
         Example e = new Example();   X  CE: Example is abstract; cannot be instantiated
   }
}
```

Q) Why abstract class cannot be instantiated?

Because is not fully implemented class so its abstract method cannot be executed.

If compiler allows us to create object for abstract class, we can invoke abstract method using that object which cannot be execute by JVM at runtime. Hence to restrict calling abstract methods compiler does not allow us to instantiate abstract class.

Q) Who will provide implementation (body) for abstract methods?

Sub class developers provide implementation for abstract methods according to their business requirement. Basically in projects abstract methods (method prototypes) are defined by super class developer, and they are implemented (method body and logic) by sub class developer.

While deriving a sub class from an abstract class we should satisfy below rule

Rule 4: The sub class of an abstract class should override all abstract methods or it should be declared as abstract else it leads to CE

Ex:

```
abstract class Example
{
 abstract void m1();
 abstract void m2();
}
```

```
class Sample extends Example   X CE:
{
   void m1()
   {
     SopIn("m1");
   }
}
```

Solution: Declare class as abstract

```
abstract  class Sample extends Example   ✓
{
   void m1()
   {
     SopIn("m1");
   }
}
```

Or

Override both methods

```
class Sample extends Example   ✓
{
   void m1()              void m2()
   {                      {
     SopIn("m1");           SopIn("m2");
   }                      }
}
```

Q) What type of members we can define in an abstract class?

We can define all static and non-static members including constructors plus abstract method. So in an abstract class we can define ***9 types of members***.

Q) Will abstract class memory be created when subclass object is created?

Yes, its non-static members get memory when its *concrete subclass object* is created.

Q) How can we execute static and non-static concrete members of abstract class?

Static members can be executed directly from its main method and its non-static members are executed by using its *concrete sub class* object.

Below program shows executing abstract class static members.

what is the output from the below program?

```
abstract class Example
{
        abstract void m1();

        static int a = 10;
        int x = 20;

        static
        {
                System.out.println("Example SB");
        }

        {
                System.out.println("Example NSB");
        }

        Example()
        {
                System.out.println("Example Constructor");
        }

        static void m2()
        {
                System.out.println("Example SM");
        }
        void m3()
        {
                System.out.println("Example NSM");
        }
        public static void main(String[] args)
        {
                System.out.println("Example main");

                System.out.println("a: "+a);
                m2();
                Example e = new Example();
                e.m3();
        }
}
```

Below program shows executing abstract class non-static members

```
class Sample extends Example
{
        static int b = 30;
        int y = 40;

        static
        {
                System.out.println("Sample SB");
        }

        {
                System.out.println("Sample NSB");
        }

        Sample()
        {
                System.out.println("Sample Constructor");
        }

        static void m4()
        {
                System.out.println("Sample SM");
        }
        void m5()
        {
                System.out.println("Sample NSM");
        }

        void m1()
        {
                System.out.println("m1 in Sample");
        }

        public static void main(String[] args)
        {
                System.out.println("\nSample main");

                System.out.println("a: "+a);
                System.out.println("b: "+b);
                m2();
                m4();

                System.out.println();

                Sample s = new Sample();
                s.m1();
                s.m3();
                s.m5();
        }
}
```

What is the output from the above program?

>javac Example.java
>java Example

>javac Sample.java
>java Sample

Q) Can we declare abstract method as static?

No, we are not allowed to declare abstract method as static. It leads to compile time error *illegal combination of modifiers abstract and static*

If compiler allows us to declare it as static, it can be invoked directly which cannot be executed by JVM at runtime. Hence to restrict in calling abstract methods compiler does not allow us to declare abstract method as static.

***For Example*:**

```
abstract class Example{
   static abstract void m1();  CE: illegal combination of modifiers abstract and static
}
```

Q) Can we declare abstract method as final?

No, because it should be allowed to overridden in sub class. It leads to compile time error *illegal combination of modifiers abstract and final*

***For Example*:**

```
abstract class Example{
    final  abstract void m1();  CE: illegal combination of modifiers abstract and final
}
```

Q) Can we declare abstract method as private?

No, because it should be inherited to sub class. It leads to compile time error *illegal combination of modifiers abstract and private*

***For Example*:**

```
abstract class Example{
  private abstract void m1();  CE: illegal combination of modifiers abstract and private
}
```

What are the legal modifiers allowed in combination with abstract modifier?

Only two modifiers are allowed with abstract
They are

1. protected
2. public

If we use any other 8 modifiers it leads to
CE: illegal combination of modifiers

If we user abstract modifier again it leads to
CE: repeated modifier
abstract abstract void m1();

Q) Identify valid abstract method declarations from the below list?

```
abstract class Example
{
        abstract void m1();

        abstract int m2();

        static abstract void m3();

        final abstract void m4();

        private abstract void m5();

        protected abstract void m6();

        public abstract void m7();
}
```

Write down service user and then subclass classes for the above abstract class

Q) Can we declare concrete class as abstract?

Yes it is allowed, Defining a class as abstract is a way of preventing someone from instantiating a class that is supposed to be extended first.

Requirement to declare concrete class as abstract:

To ensure our class *non-static members* are only accessible via sub class object we should declare concrete class as abstract.

Predefined concrete abstract classes are

- java.awt.Component
- javax.servlet.http.HttpServlet

Concrete class declared as abstract

```
abstract class Example
{
        static void m1(){
                System.out.println("Example m1");
        }

        void m2(){
                System.out.println("Example m2");
        }
}
```

Calling concrete abstract class members from normal class

```
class Sample {
        public static void main(String[] args)
        {
                Example.m1();

                Example e = new Example();
                e.m2();
        }
}
```

Calling concrete abstract class members from sub class

```
class Sample extends Example{
        public static void main(String[] args)
        {
                m1();

                Sample e = new Sample();
                e.m2();
        }
}
```

Understating interface

Definition and need of interface

Interface is a fully un-implemented class used for declaring set of operations of an object for developing a loosely coupled runtime polymorphic object user class to use these operations from different subclasses of this interface.

So it is a pure abstract class allows us to define only "public static final variables and public abstract methods" for declaring a object operations.

Basically it is used for developing a specification / contract document between service user and service provider to access that objects operations by service user those are implemented by a service provider.

Actually it is introduced in Java to support developing *multiple inheritance*.

Q) What is the need of interface when we have abstract class to define abstract methods?

A) Java does not support multiple inheritance with classes. So we must use interface as super class to develop abstraction for supporting multiple inheritance. If we define abstract class in place of interface, a service provider cannot implement multiple specifications so that service provider cannot have multiple businesses. For example Tata, Reliance, NareshTechnologies cannot have multiple businesses in reality if we use abstract class for developing specification.

Syntax to create interface

Using the keyword "interface" programmer can develop a class of type interface.

```
interface Shape {
  double PI = 3.14;

  void findArea();
  void findParameter();
}
```

Save above interface in a file name Shape.java. We can compile interface but we cannot execute, because it does not have main method.

```
Compile it with javac command to get its .class file
>javac Shape.java
        |->Shape.class

>java Shape
Exception in thread "main" java.lang.NoSuchMethodError: main
```

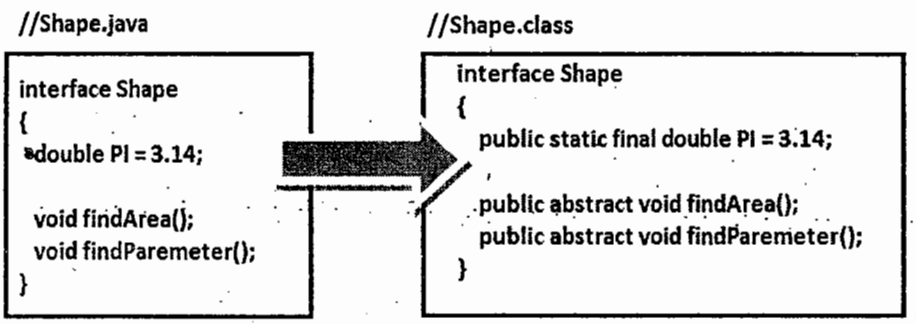
In developing, using and deriving a subclass from an interface we must follow below rules:

Rule #1 interface cannot have concrete methods, violation leads to compile time error

```
interface Shape{
  void findArea()  {
    System.out.println("shape area"); //CE: interface methods cannot have body
  }
}
```

Rule #2: In interface we can have only static final variables, even if we create variable as non-static, non-final variable compiler convert it as static, final variable as shown below.

Check below diagram for compiler code change in interface



Conclusion:

==========

1. Inteface does not have any super class / interface
2. Default access specifier of inteface is package
3. Default access specifier of inteface members is public
4. By default interface members are
=> public static final variables
=> public abstract methods

If we do not provide these keywords, compiler will place all the above keywords

Rule #3: We cannot declare interface members as private or protected members, violation leads to CE: modifier is not allowed here.

```
//Shape.java
interface Shape
{
        private void findArea();          X CE: modifier private not allowed here
        protected void findParemeter();   X CE: modifier protected not allowed here

        void print();                     ✓
        public void display();            ✓
}
```

Note: print() method accessibility modifier is not default, it is public.
complier will add public and abstract keywords as shown in the below diagram

```
//Shape.class
interface Shape
{
        public abstract void print();
        public abstract void display();
}
```

Rule #4: Interface variables should be initialized at the time of creation because they are final, else it leads compile time error "= expected".

```
interface Shape{
  //double PI;  X CE: = expected
}
```

Rule #5: Interface cannot be instantiated, but its reference variable can be created for storing its subclass objects references to develop *loosely coupled user application* to get *Runtime Polymorphism* for executing method from its different subclasses.

```
class Example {
    public static void main(String[] args)  {
        //Shape s = new Shape();X CE: Shape is abstract; cannot be instantiated

        Shape s;  ✓          Shape s = null;  ✓
    }                        Shape s = new Rectangle();✓
}                                        //assume Rectangle is subclass of Shape
```

Rule #6: We cannot declare interface as final, it leads to compile time error
Because it should contain sub class.

```
final interface Shape{  X CE: illegal combination of modifiers: interface and final
}
```

Q) Can we apply abstract keyword to interface?
Yes, it is optional and at compilation time it is removed by compiler.

```
abstract interface Shape{
        void findArea();
}
```

```
interface Shape{
        void findArea();
}
```

Q) Can we create empty interface?
Yes, it is possible.

```
interface Shape{

}
```

Q) What is the purpose of empty interface?
A) To create marker interface
For more details on marker interface check *IOStreams* chapter

Q) How can we write logic in interface?
Using Inner class we can provide logic in interface.

```
interface I  {
        class A{
                void m1(){
                        System.out.println("In inner class");
                }
        }
}
```

For more details check *Inner classes* chapter.

Inheritance with interface

Using "*implements*" keyword we can derive a class from interface.

Rule #7: The class derived from interface should implement all abstract methods of interface, otherwise it should be declared as abstract else it leads to CE.

Below program shows implementing inheritance with interface

```
interface Vehicle{
        void engine();
        void breaks();
}
```

```
abstract class Bus implements Vehicle{
        public void breaks(){
                Sopln("Bus has two breaks");
        }
}
```

At Bus level the method breaks() can only be implemented because it is common for all Bus subclasses. The method engine() will be implemented in subclasses as shown below. So this class must be declared as abstract.

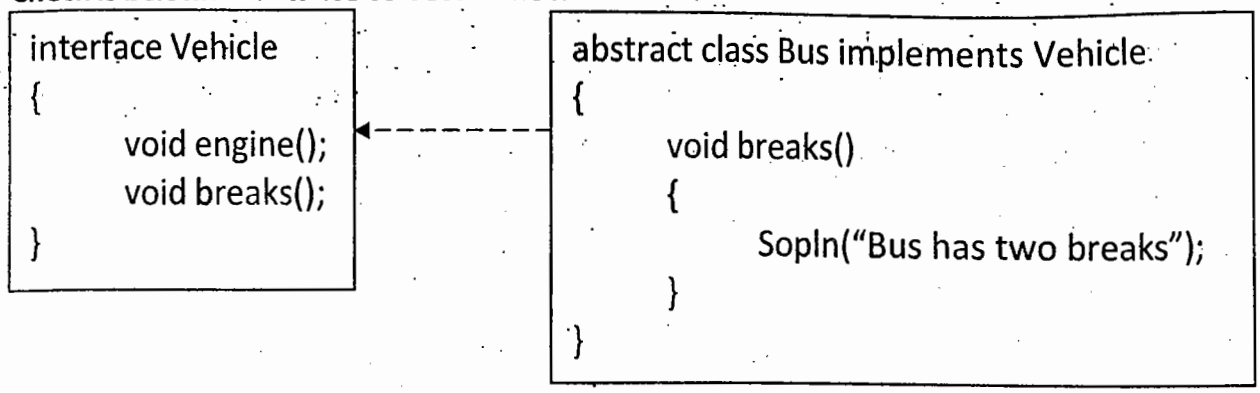
```
class RedBus extends Bus{
        public void engine(){
                Sopln("RedBus engine capacity is 40 kmph");
        }
}
```

```
class Volvo extends Bus{
        public void engine(){
                Sopln("Volvo engine capacity is 110 kmph");
        }
}
```

User program

```
class Driver{
        public static void main(String[] args){
                Vehicle v;

                v = new RedBus();               v = new Volvo();
                v.engine();                     v.engine();
                v.breaks();                     v.breaks();
        }
}
```

Check is below inheritance correct or not?



Rule #8: Sub class should implement interface method with public keyword, because interface method's default accessibility modifier is public. Below is the correct implementation.

```
abstract class Bus implements Vehicle
{
    public void breaks()
    {
        Sopln("Bus has two breaks");
    }
}
```

The similarities and differences between abstract class and interface:

Main differences to be answered in interview

Interface is a fully unimplemented class used for declaring operations of a object. Abstract class is a partially implemented class. It implements some of the operations of the object those are declared in its interface. These implemented operations are common for all next level subclasses. Remaining operations are implemented by the next level subclasses according to their requirement.

Interface allows us to develop multiple inheritance so we must start object design with interface, where as abstract class does not support multiple inheritance so it always comes next to interface in object development graph.

Similarities

1. Both interface and abstract class cannot be instantiated, but abstract class can be instantiated indirectly via sub class.
2. Reference variable can be created for both interface and abstract class
3. Sub class should implement all abstract methods
4. Both cannot be declared as final.

Other important Differences

Abstract class	Interface
1. It is a partial abstract class, it allows us to define both concrete and abstract methods	It is pure abstract class, it allows us to define only abstract methods
2. It provides both reusability and forcibility	It provides only forcibility
3. It should be declare as abstract using abstract keyword, abstract methods also should contain abstract keyword	Declaring it as abstract is optional, also declaring its methods as abstract is optional. Compiler places abstract keyword at compilation time.
4. Its default accessibly modifier is package, can be changed to public.	Its default accessibly modifier is also package, can be changed to public.
5. Its members default accessibly modifier is package, can be changed to any of other three accessibility modifiers.	Its members default accessibly modifier is public, cannot be changed.
6. Its variables are not by default static and final.	Its variables are by default static and final.
7. Static final variables no need to be initialized at the time of creation, but should be initialized in any one of the static blocks.	In interface, variables should be initialized at the time of creation as there are no static blocks.
8. Inheritance relation is established using "extends" keyword	Inheritance relation is established using "implements" keyword.
9. It can have inner class	It can also have inner class

Design a Zoo class to execute the same invoked methods from different animal classes?

To develop this class below things should be satisfied

1. Method parameter type should be *super class* of all animal classes, because it must take all types of animal class objects as arguments.
2. To satisfy above point, method parameter type can be **Object** type, but here the problem is - user can pass any type of object, because compiler only checks argument is sub class of Object or not.
3. Hence to solve above problem and to allow user to send only animal type objects, and also to let compiler to know whether the objects are of type animal or not, we should define a special class as super class to all animal type of classes. Assume that class we define with name "**Animal**". So method prototype should be define as like below

```
class Example{
        void m1(Animal a){

        }
}
```

```
class Example{
        void m1(Object obj)
        {

        }
}
```

Wrong design

Answer my questions:

Q) What are the methods you can call from this method?
A) Only methods those are available in Animal class.

Q) Ok, then what are the methods you can define in Animal class?
A) The methods those are common for all animals, because it is generic class for all Animal type classes. (***Generalization***)

Q) Alright, what are those methods?
A) Assume eat(), sleep() ☺.

Q) Well☺, What is the logic you can implement for these two methods in Animal class?
A) ☹, No Logic.

> We cannot define any common logic for these two methods in Animal class. These two methods logic should be implemented specific to each and every animal separately, because every animal has its own food habits and sleeping places.

Q) Nice thinking, then how will you define these two methods in Animal class – as *concrete methods* or as *abstract methods*?
A) Hhhhh, as abstract methods.

You are genius,

Q) But what is the problem if we declare them as empty concrete methods?

> If we develop these two methods as concrete methods, we cannot guarantee those methods are implemented in sub classes.
>
> To force sub class developers to implement these two methods we should create these two methods as abstract methods. So that, user program developer does not face problems at runtime when methods are being executed, since all Animal subclasses develop logic with same method prototype.

Q) Good design, ok now you tell, what type of class is Animal?
A) Okay, it should be an interface, because we are creating only abstract methods and also for supporting multiple inheritance.

Below you have complete design and development

- Animal class definition,
- Its sub classes and
- User class as per your requirement.

```
public class Lion implements Animal
{
        public void eat()
        {
            Sopln("I, Lion, eat non-veg");
        }
        public void sleep()
        {
           Sopln("I, Lion, sleep in caves");
        }
}
```

```
public interface Animal
{
        public void eat();
        public void sleep();
}
```

```
public class Rabbit implements Animal
{
        public void eat()
        {
            Sopln("I, Rabbit, eat veg");
        }
        public void sleep()
        {
           Sopln("I, Rabbit, sleep in bushes");
        }
}
```

```
class Zoo
{
     public static void sendAnimal(Animal a)
     {
        a.eat();
        a.sleep();
     }
}
```

```
class AnimalUser
{
     public static void main(String[] args)
     {
        Zoo.sendAnimal(new Lion());
        Zoo.sendAnimal(new Rabit());
     }
}
```

Understanding enum:

Definition and need of enum

enum is a one type of class which is final. It is created by using the keyword "enum". It is used for defining set of named constants those represents a menu kind of items.

For example:

Hotel Menu,
Bar Menu,
Naresh Technologies course menu,
etc ...

Before Java 5 these menu items are created by using class. class has a problem in accessing these menu items, that is it cannot return or print item name as it is declared instead it returns or prints its value.

For example:

```
//Months.java
class Months{
        static final int JAN    = 1;
        static final int FEB    = 2;
}

//Year.java
class Year{
        public static void main(String[] args){
                System.out.println(Months.JAN);
                System.out.println(Months.FEB);
        }
}
```

Prints:

```
1
2
```

enum is introduced to solve above problem, it means to print menu item name not value.

Syntax: to create enum data type in Java 5 a new keyword "enum" is introduced.

```
enum <enumname>{
        //menu items names with "," seperator
                ;
        //normal all 8 members which you defined in a normal class
}
```

Note: For creating named constants use only names do not use any datatype

For example

```
//Months.java
enum Months{
        JAN, FEB
}

//Year.java
class Year{
        public static void main(String[] args){
                System.out.println( MonthsEnum.JAN );
                System.out.println( MonthsEnum.FEB );
        }
}
```

Compilation:

```
>javac Year.java
>java Year
```

Prints:

```
JAN
FEB
```

More examples:

1. Define an enum for storing Flowers
2. Define an enum for storing NareshTechnologies courses

```
//Flowers.java
enum Flowers{
        ROSE, LILLY
}

//Courses.java
enum Courses{
        C, CPP, ORACLE, JAVA, DOTNET
}
```

What is the datatype of named constants in enum?

Its data type is current enum type. It is added by compiler automatically.

For example

In the enum Months, its named constants are converted as shown below

```
public static final Months JAN;
public static final Months FEB;
```

Compiler changed code for the enum Months (Months.class)

```
final class Months extends java.lang.Enum{

  public static final Months JAN;
  public static final Months FEB;

  private static final Months[] $VALUES;

  public static Months[] values(){
    return (Months[])$VALUES.clone();
  }

  public static Months valueOf(String s){
    return (Months)Enum.valueOf(Months, s);
  }

  private Months(String s, int i){
    super(s, i);
  }

  static {
    JAN      = new Months("JAN",  0);
    FEB      = new Months("FEB",  1);

    $VALUES = new Months[]{ JAN, FEB };
  }
}
```

Conclusion from above program

1. "enum" is a "final class"
2. It is subclass of java.lang.Enum.
3. It is a abstract class which is the default super class for every enum type classes. Also it is implementing from Comparable and Serializable interfaces
4. Since every enum type is Comparable so we can add its objects to collection TreeSet object, also it can be stored in file as it is Serializable type.
5. All named constants created inside enum are referenced type the current enum type. Compiler adds the missing code, and
6. These enum type variables are initialized in static block with this enum class object by using (String, int) parameter constructor
   Here
   - String parameter value is named constant name exactly as declared in its enum declaration
   - int parameter value is its position in the list. The position number starts from "ZERO"

7. In every "enum" type class compiler places private (String, int) parameter constructor with "super(s, i);" statement to call java.lang.Enum class constructor. The string value "named constant name" and its ordinal value, "its position in the list". These two values are stored in java.lang.Enum class non-static variables "name" and "ordinal" respectively, which are created in our enum class object separately for every enum variable's object.
8. Also below two additional methods are add
   1. To return all enum variable's objects
      public static <enumtype>[] values()
      For example:
      public static Months[] values()

   2. To return only the given enum object
      public static <enumtype> valueOf(String s)
      - here parameter is named constant name in String form
      For example:
      public static Months valueOf(String s)

      Months m = Months.valueOf("JAN")

9. It is also inheriting methods from java.lang.Enum class. The important methods are
   1. public final String name()
      returns name of the current enum exactly as declared in its enum declaration

   2. public final int ordinal()
      returns current enum position

Q) Why compiler places no-arg constructor with no-arg super call in normal classes and parameterized constructor with (string,int) arg super class in enum?
A) For normal classes super class is java.lang.Object and it has no-arg constructor so compiler places default constructor without parameters with no-arg super call.

But for enum the super class is java.lang.Enum, it has String, int parameter constructor. So compiler places default constructor with (String, int) parameter with super call with string, int argument.

enum type class object JVM Architecture:
As we said in above points every enum type class object has minimum 2 variables
1. name *and*
2. ordinal

Every named constant variable defined inside enum type is an objects referenced variable that holds current enum class object. So, in the above Months enum class JAN, FEB are objects. Below is the Months enum class objects structure

Write a program to print name and ordinal of all enum objects of enum Months

```
// MonthsNameAndOrinal.java
class MonthsNameAndOrinal{
        public static void main(String[] args){
                Months[] months = Months.values();

                for (Months month : months){
                        System.out.println(month.name());
                        System.out.println("...");
                        System.out.println(month.ordinal());
                }
        }
}
```

Q) How can we assign prices (values) to Menu items in enum? Is below syntax is valid?

```
enum Months{
        JAN = 1, FEB = 2
}
```

It is a wrong syntax, it leads to CE because named constants are not of type "int" they are of type "Months". So we can not assign values to named constants directly using "=" operator.

Q) Then how should we assign?

Syntax:

```
namedconstant(value)
```

For example;

```
JAN(1), FEB(2)
```

Rule: To assign values to names constants as shown in the above syntax, enum must have a parameterized constructor with the passed argument type. Else it leads to CE.

Q) Where these values 1, 2 are stored?

A) We must create a non-static int type variable to store these values.
So, to store these named constants values we must follow below 3 rules
In enum class

1. We must create non-static variable in enum with the passed argument type
2. Also we must define parameterized constructor with argument type
3. Named constants must end with ";" to place normal variables, methods, constructors explicitly. It acts as separator for compiler to differentiate named constants and general members.

Below code shows defining enum constants with values

```
//Months.java
enum Months{
        JAN(1), FEB(2)
                ;
        private int num;

        Months(int num){
                this.num = num;
        }
        public int getNum(){
                return num;
        }
        public void setNum(int num){
                this.num = num;
        }
}
```

Compilation:

```
>javac Months.java
        |->Months.class
```

Compiler changed code

In .class file developer given constructor code is merged with enum default constructor code that is given by compiler, as shown below

//Months.class

```
final class Months extends Enum{

  public static final Months JAN;
  public static final Months FEB;
  private int num;
  private static final Months $VALUES[];

  public static Months[] values(){
    return (Months[])$VALUES.clone();
  }
  public static Months valueOf(String s){
    return (Months)
        Enum.valueOf(Months, s);
  }

  private Months(String s, int i, int price){
    super(s, i);
    this.num = num;
  }

  public int getnum() {
    return num;
  }

  public void setnum(int num){
    this.num= num;
  }

  static{
    JAN = new Months("JAN", 0, 1);
    FEB = new Months("FEB", 1, 2);
    $VALUES = (new Months[]{ JAN, FEB});
  }
}
```

Write a program to print above Months as how the real Months items are appeared.

Expected Output:
1. JAN
2. FEB

//Year.java

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

                Months[] menuItems = Months.values();

                for(Months menuItem : menuItems){
                   System.out.print( menuItem.getNum() + ". " );
                   System.out.println( menuItem.name() );
                }
        }
}
```

JVM Architecture:

SCJP Questions (written test questions)
enum has below set of rules

Rule #1: on enum subclass
We cannot derive a subclass from enum, because it is final class

For example
```
enum Flowers{}
enum Rose extends Flowers{} CE: '{' expected
```

Rule #2: on enum object
We cannot instantiate enum using new and contructor, it leads to CE: enum types may not be instantiated, but it is instantiated by compiler in .class file.

```
enum Months{
        ;
        public static void main(String[] args){
                Months m1 =  new Months("JAN", 0);
        }
}
```

Rule #3: on ";"

- Named constants must be separated with comma "," but not with semicolon ";"
  For example
```
enum Color{
        RED ; BLUE ; CE: <identifier expected>
}
```
  here BLUE is not considered as named constant,
  because ; represents end of all named constants.

- But we can place ; at end of all named constants, here in enum it is act as separator and also tells end of all named constants.

```
enum Color{
        RED, BLUE ;
}
```

- After named constants semicolon is mandatory to place normal members (static and non-static members) to enum definition
  For example

```
enum Color{
        RED, BLUE

        int x = 10; ✗ CE: ',' , '}', or ';' expected
}
```

```
enum Color{
        RED, BLUE
            ;
        int x = 10; ✓
}
```

Rule #4: on "enum members"

Inside enum we can define

1. Named constants
2. All static members- SV, SB, SM, MM
3. All Non-static members except abstract methods- NSV, NSB, NSM, Constructor
1. Rule: Abstract method is not allowed because we cannot declare enum as abstract as enum object should be created to initialized named constants
2. inner class, interface, enum

For example:

```
enum Color{
        RED(15), BLUE(25), GREEN
                ;
        static int a = 10;
               int x = 20;

        static void m1(){
                System.out.println("SM");
        }
        void m2(){
                System.out.println("NSM");
        }

        static{
                System.out.println("SB");
        }
        {
                System.out.println("NSB");
        }

        Color(){
                System.out.println("no-arg constructor");
                this.x = 50;
        }
        Color(int x){
                System.out.println("int-arg constructor");
                this.x = x;
        }

        //abstract void m3(); CE:
        public static void main(String[] args){
                System.out.println("Color main");
        }

        class A{}
}// Color close
```

Mandatory

Q) How can we access enum members?

1. *named constants,* we can access named constants by using enum name (Color), because they are static
2. *static members,* we access static members by using enum name (color)
3. *non-static members,* we access non-static members by using enum objects (named constants RED, BLUE)

Syntax:

enumname.namedconstant.NSM

Below application shows accessing static and non-static members of an enum

```
class Test{
    public static void main(String[] args){
        System.out.println("Test main");

        //accessing named constants
        System.out.println(Color.RED);
        System.out.println(Color.BLUE);

        //accessing static members
        System.out.println(Color.a);
        Color.m1();

        //accessing non-static members
        System.out.println(Color.RED.x);
        System.out.println(Color.BLUE.x);
        System.out.println(Color.GREEN.x);

        Color.RED.m2();
        Color.BLUE.m2();
    }// main close
}// Test close
```

Compilation:

```
>javac Test.java
```

> java Color

```
NSB
int-arg Constructor
NSB
int-arg Constructor
NSB
no-arg Constructor
SB
Color main
```

>java Test

```
Test main

NSB
int-arg Constructor
NSB
int-arg Constructor
NSB
no-arg Constructor
SB
```

```
RED
BLUE

10
SM
15
25
50
NSM
NSM
```

Rule #5: On enum members order

named constants must be the first members in enum and they must be separated with *comma*

For example

Case #1: enum with Named Constant (NC), and Non-static member (NM)

```
enum Color{
        RED, BLUE
                ;
        int a = 10;
}
```

Case #2: enum with NC, NM, and NC

```
enum Color{
        RED;
        int a = 10;
        //BLUE; CE: <identifier> expected
}
```

Case #3: enum with NM and NC

```
enum Color{
        int a = 10;
        RED,BLUE;
}
```

Case #4: empty enum

```
enum Color{
}
```

Case #5: enum with only NMs

```
enum Color{
        int a = 10;
}
```

Rule: To define enum only with NMs it must starts with ";" to tell to compiler it is not NC.

```
enum Color{
        ; int a = 10;
}
```

Rule #6: On constructor

We cannot declare explicit constructor as protected or public, because enum constructor is by default private.

Find out compile time errors

```
enum Color{
        private Color(){}
}
enum Color{
        Color(){}
}
enum Color{
        protected Color(){}
}
enum Color{
        public Color(){}
}
```

If we do not apply Accessibility Modifier to constructor compiler changes it to private.

Q) Hello where is ";" how is above enum compiled without starting with " ; "?

A) ";" is optional if enum has only no-arg constructor.

Check below bits

Case #1: ";" is mandatory before constructor if we place constructor along with NCs

```
enum Months{
        JAN, FEB
            ;
        Months(){}
}
```

Case #2: Also ";" is mandatory if we write constructor with parameters

```
enum Months{
            ;
        Months(int a){}
}
```

Case #3: Also ";" is mandatory if we write no-arg constructor along with normal members

```
enum Months{
            ;
        Months(){}
        int a= 10;
}
```

Case #4: Also ";" is mandatory if we write no-arg constructor as private

```
enum Months{
            ;
        private Months(){}
}
```

Q) Can we overload constructor in enum?

We can overload constructors in enum to initialized named constant objects with different type of values.

For Example:

```
public enum Months{
        JAN(1), FEB(2L), MAR("3")
                ;
        private int number;
        private Months(int num){
                this.number = num;
        }
        private Months(long num){
                this.number = (int)num;
        }
        private Months(String num){
                this.number = Integer.parseInt(num);
        }
}
```

Compiler changed code in the above program:
All three constructors are updated with name, ordinal parameters, and in body compiler places "super(s, i);" as first statement.

check below code:

```
public final class Months extends java.lang.Enum{

        public static final Months JAN;
        public static final Months FEB;
        public static final Months MAR;

        private int number;
        private static final Months $VALUES[];

        public static Months[] values(){
                return (Months[])$VALUES.clone();
        }
        public static Months valueOf(String s){
                return (Months)Enum.valueOf(Months, s);
        }

        private Months(String s, int i, int num){
                super(s, i);
                this.number = num;
        }
        private Months(String s, int i, long num){
                super(s, i);
                this.number = (int)num;
        }
        private Months(String s, int i, String num){
                super(s, i);
                this.number = Integer.parseInt(num);
        }
        static{
                JAN= new Months ("JAN", 0, 1);
                FEB = new Months ("FEB", 1, 2);
                MAR= new Months ("MAR", 1, 3);
                $VALUES = (new Months []{ JAN, FEB, MAR});
        }
}
```

Rule #7: On Named constants declaration
Named constants must be declared according to that enum's available constructors

For example:

Case #1: If we do not define constructor, it has default constructor. So, NCs must be declared without arguments

Find out compile time error

```
enum Months{
        JAN, FEB, MAR(3)
}
```

Case #2: If we define constructor with int parameter, then it does not have default constructor. So, all NCs must be declared with only int arguments

Find out compile time error

```
enum Months{
        JAN, FEB, MAR(3)
                ;
        Months(int i){}
}
```

Case #3: if we define constructors with no-arg, int and String parameters, all NCs must be declared either without argument or with int argument or with String argument

Find out compile time error

```
enum Months{
        JAN, FEB("2"), MAR(3), APR('a'), MAY(5L)
                ;
        Months(){}
        Months(int i){}
        Months(String s){}
}
```

Rule #8: allowed modifiers to enum

Only public, strictfp modifiers are allowed for enum

Find out CEs in the below list

```
private enum Color{}
protected enum Color{}
public enum Color{}
static enum Color{}
final enum Color{}
abstract enum Color{}
strictfp enum Color{}
```

- private, protected, static are not allowed as they are only applicable to class members
- final is not allowed as it is already final
- abstract is not allowed as it should be instantiated and more over it is final

Rule #9: inner enum

We can define enum inside a class but we cannot define it inside a method

```
class A{
        enum Color{}  ✓
        void m1(){
                enum Color{} ✗
        }   CE: enum types must not be local
}
```

- private, protected, public, static, strictfp are allowed for inner enum.
- final, abstract modifiers are not allowed also for inner enum.

Chapter 16

Inner Classes

- In this chapter, You will learn
  - Definition of inner class
  - Need of inner class
  - Syntax to define inner class
  - Types of inner classes
  - Inner class name change after compilation
  - Allowed modifiers
  - Type of members allowed in inner class
  - Accessing outer class members from inner class
  - Accessing inner class members from outer class and outside of outer class.
  - Syntax to differentiate outerclass members from innerclass members if both have same name.

- By the end of this chapter- you will identify the correct usage and appropriate situation to use all types of inner classes.

Interview Questions

By the end of this chapter you answer all below interview questions

- What is an inner class?
- What is the need of inner classes?
- What are the advantages of Inner classes?
- What are disadvantages of using inner classes?
- What are the different types of inner classes?
- If you compile a file containing inner class how many .class files are created and what are all of them accessible in usual way?
- Can we create inner class name with outerclass name or package name?
- What is static member class?
- What are non static inner classes?
- Which modifiers can be applied to the inner class?
- Does a static nested class have access to the enclosing class' non-static methods or instance variables?
- How to access the inner class from code within the outer class?
- How to create an inner class instance from outside the outer class instance code?
- How to refer to the outer this i.e. outer class's current instance from inside the inner class?
- Can the method local inner class object access method's local variables?
- Can a method local inner class access the local final variables? Why?
- Which modifiers can be applied to the method local inner class?
- Can a local class declared inside a static method have access to the instance members of the outer class?
- Can a method which is not in the definition of the super class of an anonymous class be invoked on that anonymous class reference?
- What are different types of anonymous classes?
- Can an anonymous class define method of its own?
- Can an anonymous class implement multiple interfaces directly?
- Can an anonymous class implement an interface and also extend a class at the same time?
- How can we access local class or anonymous inner class specific members from outer class?
- Is it allowed to create inner class in interface?
- Is it allowed to create inner interface in class?
- Is it allowed to extend inner class from other class or interfaces?

Definition: The class defined inside another class or interface is called inner class.
We can also create an interface in another class or interface.

For example

```
class Example{
        class Sample{}
}
```

```
class Example{
        interface Sample{}
}
```

```
interface Example{
        class Sample{}
}
```

Q) From which Java version innerclass concept is available?
From Java 1.1 version innerclass concept is introduced for supporting AWT, and Swing components event handling.

Need of innerclass:
Innerclass is used for creating an object logically inside another object with clear separation of properties region. So that we know that object belongs to which object.

For example:

```
class Student{
        int sno;
        String sname;

        class Address{
                int hno;
                String city;
        }
}
```

If these Address properties are needed for more than one outer object then we must create Address class as outer class, and we should create this class object in those outer object classes with HAS-A relation.

Creating an innerclass is a way of logically grouping classes that are only used in one place.
It increases encapsulation. Nested classes can lead to more readable and maintainable code.

Inner classes are introduced to enclose logic of a class using another class in that same class. Basically innerclass concept is introduced to solve the problems of AWT components event handling logic development. For more details check AWT programming.

Types of inner classes:
We have 4 types of inner classes

1. Nested class (static inner class)
2. Inner class (non-static inner class)
3. Method local class (local inner class)
4. Anonymous class (argument inner class)

Syntax to create all above 4 types of inner classes:

//Example.java

```
class Example{

        static class A{ }

                class B{ }

        void m1(){
            class C{}

            m2( new Thread(){} );

        }
}
```

Compilation

- javac Example.java
  - |->Example.class
  - |->Example$A.class
  - |->Example$B.class
  - |->Example$1C.class
  - |->Example$1.class

Compiler generates .class file separately for every inner class as shown above. So that the inner class logic is completely separated from outer class and stored in another .class file.

The naming convention innerclass .class file is

- for static and non-static inner class
  - outerclassname$innerclassname.class
    - Example$A.class
    - Example$B.class
- for method local inner class
  - outerclassname$<n>innerclassname.class
    where n is an index number starts with 1, increased by 1 only if class is repeated in another method.
    - Example$1C.class
- for anonymous class
  - outerclassname$<n>.class
    where n is an index number stats with 1, incremented by 1 for every anonymous class definition.
    - Example$1.class

Find out ".class" file names for below inner classes

```
class Example{
        static class A{} ;
        class B{} ;

        void m1(){
                class C{} ;

                new Thread(){} ;

                class D{}
        }

        void m2(){
                class C{}
                class E{}

                new Thread(){};
        }
}
```

- **javac Example.java**
  - |-> Example.class
  - |-> Example$A.class
  - |-> Example$B.class
  - |-> Example$1C.class
  - |-> Example$1.class
  - |-> Example$1D.class
  - |-> Example$2C.class
  - |-> Example$1E.class
  - |-> Example$2.class

Common interview questions on inner classes

1. Syntax to define inner class
2. Allowed modifiers
3. Type of members are allowed in inner class
4. Accessing outer class members from inner class
5. Accessing inner class members from outer class and outside of outer class
6. Differentiating outer class members from interclass members if both have same name
7. Can we create innerclass with outerclass name, and can we create innerclass with other innerclass name?
8. Can innerclass derived from other classes/interfaces if so from how many?

Let us understand static inner class

The inner class defined at class level with static keyword is called static inner class.

1. *Syntax*:

```
class Example{
  static class A{}
}
```

2. *Allowed modifiers*

   private, protected, public, final, abstract, strictfp.

3. *Types of members allowed*

   All 8 types of static and non-static members are allowed

   1. static variable
   2. static block
   3. static method
   4. main method
   5. non-static variable
   6. non-static block
   7. non-static method
   8. constructor

4. *Accessing outer class members from inner class*

   We can access outer class all members from its static inner class including private members. In accessing outer class members you just consider "static inner class" as a "static method" of outer class. So, we can access static members directly by their name from static inner class. But we cannot access non-static members directly by their name we must access them only by creating outer class object.

Find out compile time errors in the below program?

```
class Example {

        static int a = 10;
                int x = 20;
        private int y = 30;

        static class A{

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

                        Example e = new Example();
                        System.out.println(e.a);
                        System.out.println(e.x);
                        System.out.println(e.y);
                }
        }
}
```

Q) Can we call outer class members using innerclass referenced variable?

No, it leads to CE: cannot find symbol

Find out CE in the below program?

```
class Example{
   int x = 10;

   static class A{
     static void m1(){
        A a = new A();
        Sopln(a.x);

        Example e = new Example();
        Sopln(e.x);

     }
   }
}
```

Q) Why cannot we call non-static members from static innerclass?
Static inner class members do not get memory in outer class object. So we must create outer class object explicitly to access it non-static members. But in case of static members call compiler places outer class name in accessing static member's definition.

5. Accessing inner class members from outer class and outside of outer class

Accessing from outer class:

Including private members we can access all members of static inner class's from outer class as shown below

1. Static members by using inner class name
2. Non-static members by using its object

Below program shows accessing inner class members from outer class

```
class Example{
        static class A{
                private int y = 20;
                static void m1(){  System.out.println("inner class SM m1"); }
                void m2(){ System.out.println("inner class NSM m2"); }
        }//inner class close
        public static void main(String[] args){
                A.m1();

                A a = new A();
                System.out.println( a.y );
                a.m2();
        }
}
```

Accessing from outside of the outer class

The syntax for accessing inner class from outside outer class is

"outerclassname.innerclassname"

***Rule**: Like outer class private members, inner class private members are also cannot be accessed from* outside of outer class.

Given:

```
class Example{
    static class A{
        static int a = 10;
        int x = 20;
        private int y = 30;

    }
}
```

What is the output from the below program?

```
class Sample{
    public static void main(String[] args) {
        System.out.println( "a: "+ Example.A.a );

        Example.A a1 = new Example.A();
        System.out.println("x: "+a1.x);
        //System.out.println("x: "+a1.y);
    }
}
```

CE: y has private access in Example.A

Now let us learn executing a class with static innerclass

Given:

```
class Example {

    static class A{

        public static void main(String[] args){
                Sopln("Inner class main method");
        }
    }

    public static void main(String[] args){
        Sopln("Outer class main method");
    }

}
```

Compilation: First let us compile above program. It creates two .class files

```
>javac Example. Java
        |-> Example.class
        |-> Example$A.class
```

Execution

Outer class execution

>java Example

Outer class main method

Inner class execution By just executing outer class, static inner class members are not executed automatically. So we must executed static inner class separately as shown below

>java Example$A

Inner class main method

Very important point to be remembered

When we load outer class inner class is not loaded and in the same way

When we load inner class outer class is not loaded. *What is the output from the below program*

```
class Example {
        static{
                System.out.println("outer class is loaded");
        }

        static class A{
                static{
                        System.out.println("inner class is loaded");
                }

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

```
>java Example


>java Example$A
```

Focus: If we access outer class members from inner class or inner class members from outer class then both classes are loaded into JVM. *What is the output come from the below program?*

```
class Example {
        static{
                System.out.println("Outer class is loaded");
        }
        Example(){
                System.out.println("Outer class constructor");
        }
        static class A{
                static{
                   System.out.println("Inner class is loaded");
                }
                A(){
                   System.out.println("Inner class constructor");
                }
                static void m1(){
                  System.out.println("Inner class SM");
                }
                void m2(){
                  System.out.println("Inner class NSM");
                }
                public static void main(String[] args){
                        System.out.println("inner class main");
                        Example.m1();
                        Example e = new Example();
                        e.m1();
                }
        }

        static void m3(){
                System.out.println("Outer class SM");
        }
        void m4(){
                System.out.println("Outer class NSM");
        }
        public static void main(String[] args){
                System.out.println("outer class main");
                A.m3();
                A a = new A();
                a.m4();
        }
}
```

```
>java Example

>java Example$A
```

Q) Can we define outer class members in inner class again?
A) Yes, it is possible

Q) Then how can we differentiate both members in inner class?
A) By using their class name or object name

```
class A{
        static int a = 10;
                int x = 20;

        static class B{

                static int a = 50;
                        int x = 60;

                void m1(){
                        System.out.println(a);
                        System.out.println(x);

                        A a = new A();
                        System.out.println(A.a);
                        System.out.println(a.x);

                }//m1 close
        }//inner class close

        void m2(){
                        System.out.println(a);
                        System.out.println(x);

                        B b = new B();
                        System.out.println(B.a);
                        System.out.println(b.x);
        }

        public static void main(String[] args){
                A a = new A();
                a.m2();

                B b = new B();
                b.m1();
        }
}
```

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

        A a1 = new A();
        A.B b1 = new A.B();

        System.out.println(A.a);
        System.out.println(A.B.a);

        System.out.println(b1.x);
        System.out.println(b1.x);
   }
}
```

Interview points on static inner class:

1. The inner class created at class level with static keyword is called static inner class.
2. A separate ".class" file is created to store its bytecodes with "outerclassname$innerclassname"
3. All 8 types of members are allowed in static inner class.
4. All outer class members including private members are allowed to access from inner class directly if they are static and if it is non-static by using outer class object.
5. In the same way, all inner class members including private are allowed to access from outer class only by using its name or object.
6. All non-private inner class members are allowed to access from outside outer class by using below syntax "outerclassname.innerclassname.membername"
7. We can execute static inner class alone with the syntax
   "java outerclassName.innerclassname"
8. If we load outer class, static inner class is not loaded. It is loaded only if one of its member is called from outer class.
9. Outer class and inner class members are differentiated by using their class names or objects name.

Let us understand non-static inner class

The inner class that is definition at class level without static keyword is called non-static inner class. We must develop non-static inner class to create a separate object in outer class object. So that when outer class is instantiated this inner class members are provided memory as part of every outer class instance.

1. *Syntax*:

```
class Sample{
    class B{}
}
```

Q) After compiling outer class with a non-static inner class how many .class files are generated?

A) Two .class files are generated
one for outer class and another for innerclass as shown below

```
>javac Sample.java
        |-> Sample.class
        |-> Sample$B.class
```

2. *Allowed modifiers*
   private, protected, public, final, abstract, strictfp.

3. *Types of members allowed*
   Only non-static members are allowed they are:
   1. non-static variable
   2. non-static block
   3. non-static method
   4. constructor

   Rule: We should not declare static members in non-static inner class
   It leads to CE: "inner classes cannot have static declarations"
   Find out compile time errors in the below program

```
class Sample{
        class B{
                static int a = 10;
                int x = 20;
        }
}
```

4. *Accessing outer class members from inner class*
   We can access outer class all members from its non-static inner class including private members by their name directly. *Find out compile time errors in the below program?*

```
class Example {

        static int a = 10;
                int x = 20;
        private int y = 30;

        class A{

                void m1{
                  System.out.println(a);
                  System.out.println(x);
                  System.out.println(y);
                }
        }
}
```

5. Accessing non-static inner class members from outer class and outside outer class
   Q) Then how can we execute non-static innerclass members?
   A) We must call non-static inner class members from outer class main method or from outside outer class main method by using its object.

Q) What is the syntax to create non-static inner class object?

There are two syntaxes

1. *Old syntax*
   B b = new B();

2. *New Syntax* <outerclassobject.innerclassobject>
   Outerclassname.innerclassname refvar =
   new outerclassconstructor().new innerclassconstructor();

   Sample.B b = new Sample().new B();

We must use *first syntax* to access non-static innerclass members from non-static members of its outer class. And we must use *second syntax* to access innerclass members from static members of outer class and also to access from outside outer class.

Q) Why we must create outer class object to create non-static innerclass object?

A) Because this inner class is a non-static member of outer class. As how you are accessing other non-static members in the same we must create outer class object to create inner class object and further to access its members.

Below program shows accessing non-inner class member from outer class and outside outer class

```
class Sample{
   class B{
        void m1(){
          System.out.println("B m1");
        }
   }

   void m2(){
        B b = new B();          //compiler changed code: B b = this.new B();
        b.m1();
   }

   public static void main(String[] args){

        //B b = new B(); CE: non-static variable this cannot be referenced from a static context

        Sample.B b = new Sample().new B();
        b.m1();
   }
}
```

```
class Test{
   public static void main(String[] args){
        //B b = new B(); CE: Cannot file symbol class B

        Sample.B b1 = new Sample().new B();
        b1.m1();

        //other way of creation
        Sample s = new Sample();
        Sample.B b2 = s.new B();
        b2.m1();
   }
}
```

Now let us learn executing a class with static innerclass

Q) Can we define static block in non-static inner class?

No, it leads to compile time error.

Q) Can we define main method in non-static inner class?

No, it leads to compile time error.

Q) Can we execute non-static inner class members by using "java" command as
>java Sample$B

A) No, it leads to exception NSME, because it does not have main method.

So to run non-static inner class members we must execute its outer class.
Run above applications *Sample* and *Test* classes.

>java Sample
>java Test

Outer class object's memory Structure:

Q) Can we define outer class members in inner class again?
A) Yes, it is possible

Q) Then how can we differentiate both members in inner class?
A) By using *"outerclassname.this"*

```
class A{
        int x = 20;

        class B{
                int x = 50;

                void m1(){
                        System.out.println(x);
                        System.out.println(this.x);
                        System.out.println(A.this.x);
                }
                void m2(){
                        int x = 60;
                        System.out.println(x);
                        System.out.println(this.x);
                        System.out.println(A.this.x);
                }
        }

        void m3(){
                System.out.println(x);

                B b = new B();
                System.out.println(b.x);
        }

        public static void main(String[] args){
                A a = new A();
                a.m3();

                A.B b = new A().new B();
                b.m1();
                b.m2();
        }
}
```

```
class Test{
    public static void main(String[] args){
        A a = new A();
        a.m3();

        A.B b = new A().new B();
        b.m1();
        b.m2();
    }
}
```

Interview points on non-static inner class:

1. The inner class created at class level without static keyword is called static inner class.
2. A separate ".class" file is created to store its bytecodes with "outerclassname$innerclassname"
3. Only non-static members are allowed in non-static inner class.
4. All outer class members including private members are allowed to access from inner class directly.
5. In the same way, all inner class members including private are allowed to access from outer class only by using its name or object.
6. All non-private non-static inner class members are allowed to access from outside outer class by using its object "new A().new B()". A is outer class, B is innerclass
7. Non-static innerclass bytecodes are stored in every outer class instance.
8. Outer class member are differentiated from inner class members are by using the syntax: "outerclassname.*this*.non-staticmember".

Method local innerclass

The inner class defined inside a method of outer class is called method inner class.

1. *Syntax*:

```
class A{
    void m1(){
            class B{}
    }
}
```

Q) After compiling outer class with a local inner class how many .class files are generated?

A) Two .class files are generated

one for outer class and another for innerclass as shown below

>javac A.java

|-> A.class

|-> A$1B.class

Where "1" is the index number representing B is the first local inner class created in class A

2. *Allowed modifiers*

Only final, abstract, strictfp,

Accessibility modifiers are not allowed

3. *Types of members allowed in local class*

Only non-static members are allowed they are:

1. non-static variable
2. non-static block
3. non-static method
4. constructor

Rule: We should not declare static members in local inner class

It leads to CE: "inner classes cannot have static declarations"

Find out compile time errors in the below program

```
class A{
        void m1(){
            class B{
                static int a = 10;
                int x = 20;
            }
        }
}
```

4. *Accessing outer class members from local inner class*
   1. If local class is defined in static method outer class static members are accessible directly and non-static methods are accessible only through outer class object.
   2. If local class is defined in non-static method outer class all static and non-static members are accessible directly by their names and they are executed from the current object of that method.
   3. *Rule:* Its methods parameter and local variables are accessible from local inner class only if they are final, if we access its method's non-final variables it leads to CE: *"local variable is accessed from within inner class; needs to be declared final"*

Find out compile time errors in the below program?

```
class A {
    static int a = 10;
    int b = 10;

    static void m1(){
            final int c = 10;
            int d = 10;
            class B{
                    void m1(final int e, int f){
                            System.out.println(a);
                            System.out.println(b);
                            System.out.println(c);
                            System.out.println(d);
                            System.out.println(e);
                            System.out.println(f);
                    }
            }
```

```
            void m2(){
                class C{
                    void m2(){
                        System.out.println(a);
                        System.out.println(b);
                    }
                }
            }
        }
        public static void main(String[] args) {
            A a1 = new A();          A a2 = new A();
            a1.a = 5;                a2.a = 7;
            a1.b = 6;                a2.b = 8;
            a1.m1();                 a2.m1();
            a1.m2();                 a2.m2();

        }
}
```

5. Accessing local inner class members from outer class and outside outer class
   We cannot access inner class members from outer class, because it local to only the method in which it is defined. Just you treat local inner class is a local variable.

Q) Then how can we access and execute method local inner class members?
A) Method local inner class members are allowed to access only inside its enclosing method after its definition. This rule is same like local variable rule that is local variable is accessible only inside the method after its creation statement. ***Rule***: If we access method local inner class before its definition or from outside methods it leads CE: "cannot find symbol"

Find out compile time error from the below program

```
class A{
    static void m1(){
        B b1 = new B();

        class B{  int x = 10; }

        B b2 = new B();
        System.out.println(b2.x);
    }
    static void m2(){
        B b = new B();
    }
    public static void main(String[] args){
        m1();
    }
}
```

Q) Then how can we access method local inner class members from outside of the method?
A) We must follow either of the below two approaches

1. We must return local inner class object from its enclosing method or
2. We must send local inner class object as argument to outer class method

In maximum cases in project we *return* local inner class object from its enclosing method.

Procedure for returning or sending local inner class object
we must use its super class name as return type and parameter of the method.

Why should we use super class name?
Because we cannot use the local inner class name before its declaration or outside of its enclosing method.

For example:

```
class A{
    ⇨ B m1(){
            class B{
                void m3(){
                    System.out.println("B m3");
                }
            }

            B b = new B();

            m2(b);
            return b;
        }
            ⇩
        void m2(B b){
            b.m3();
        }
}
```

Above program leads to CE: "cannot find symbol class B" at arrow marked places.

Q) What is the default super class of all inner classes, is it java.lang.Object?
A) Yes, for every outerclass, and innerclass java.lang.Object is the superclass.

Q) So can we use it as return type and parameter for sending local inner class object?
A) Yes, but we cannot call methods of this local innerclass, because compiler searches method definition in Object not in local innerclass.

Find out CE in the below program:

```
class A{
        Object m1(){
                class B{
                        void m3(){
                                System.out.println("B m3");
                        }
                }
                B b = new B();
                m2(b);
                return b;
        }

        void m2(Object obj){
                obj.m3(); ⇦
        }
}
```

Above program leads to CE: "cannot find symbol method m3() in class java.lang.Object" at arrow marked place.

Solution: We must define a special super class with m3() method declaration. So this super class should be an interface type.

Below is the correct code to send local inner class object to outside of its enclosing method

```
interface Example{
        void m3();
}
class A{
        Example m1(){
                class B implements Example{
                        public void m3(){
                                System.out.println("B m3");
                        }
                }
                B b = new B();
                m2(b);
                return b;
        }

        void m2(Example e){
                e.m3();
        }
}
```

As per runtime polymorphism m3() method is executed from B class as the current object is B class object.

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

                A a = new A();
                Example e = a.m1();
                e.m3();
        }
}
```

Q) How can we differentiate outer class method variables from inner class method variable if both have same name? A) We cannot differentiate outer class method variable from inner class method local variable if both have same name. *What is the output from below program?*

```
class A {
        void m1(){
                final int x = 2;
                class B{
                        void m2(){
                                System.out.println("In B m2 x: "+x);
                                int x = 4;
                                System.out.println("In B m2 x: "+x);
                        }
                }
                B b = new B();
                b.m2();
        }
        public static void main(String[] args){
                A a = new A();
                a.m1();
        }
}
```

What is the output from the below program?

```
class A {
        int x = 1;

        void m1(){
                final int x = 2;
                class B{
                        int x = 3;

                        void m2(){
                                SopIn(x);
                                int x = 4;

                                SopIn(x);
                                SopIn(this.x);
                                SopIn(B.this.x);
                                SopIn(A.this.x);
                        }
                }
//continuation to m2() method

                        B b = new B();
                        b.m2();


                        SopIn(x);
                        SopIn(b.x);
                        SopIn(this.x);
        }

        public static void main(String[] args){
                A a = new A();
                a.m1();
        }
}
```

Interview points on local inner class:

1. The inner class created inside outer class method is called local inner class.
2. A separate ".class" file is created to store its bytecodes with "outerclassname$<n>innerclassname". Here <n> is number stats with 1 and incremented by 1 if local name is defined with same name in another method.
3. Only non-static members are allowed in local inner class.
4. All outer class members including private members are allowed to access from inner class directly if it is defined in non-static method. If it is defined in static method outer class non-static methods must be called using outer class object.
5. Only final local variables and final parameters of its enclosing method's are accessible from its local inner class.
6. Inner class members are not allowed to call from outer class because it is local to a method. We are allowed to call local inner class members only within its enclosing method that to after its definition with its object. We can also access its private members.
7. To access local inner class members from outer class or outside outer class we must pass its objects as return type or argument by using its super class name.
8. We cannot differentiate outerclass method local variable from inner class variable if both have same name.

Anonymous inner class

Anonymous class is one type of inner class. It is a nameless subclass of a class/interface. Like other inner classes it is not individual class, it is a subclass of some other existed class or interface. Using anonymous inner class we can do three things at a time

1. Inner class creation as a subclass of outer class
2. Overriding outer class method
3. Creating and sending its object as argument or return type to another method

Syntax to create anonymous inner class

```
new outerclassname(){
        //overriding outerclass methods
}
```

For Example: below statement creates anonymous class for Example class,

new Example(){} *<= you watch it closely it is not* Example class object creation

Below statement creates *Example class object* not anonymous class

```
new Example();
```

Below statement shows anonymous inner class creation for the interface A

```
interface A{}
new A(){}      <= It is not A interface object, it is A subclass object
```

Rule: When we develop anonymous class from an interface, we must implement all its methods in anonymous class as shown below:

```
interface A{
        void m1();
}
new A(){
        public void m1(){
                System.out.println("In anonymous class");
        }
}
```

Two ways to create anonymous class

1. As method argument - shown above
2. With destination variable - shown below *For example:* A a = new A(){};

Q) Where should we define anonymous class in outer class?

A) Anywhere in the outer class, most of the times it is created as an argument of a method.

Rule: Anonymous inner class must created with destination variable at class level, but at method level we can create it without destination variable.

Find out Compile time errors in the below program

```
class Example {

        new A(){};
        static A a1 = new A(){};
        A a2 = new A(){};

        m1(new A(){});

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

                m1( new A(){} );
                Aa3 = new A(){};
                m1(a3);
        }
        static void m1(A a){
                a.m1();
        }
}
```

Check below code for more understanding.

```
interface A{}

class B{

        static A a1 = new A(){};
        A a2 = new A(){};

        static void m1(){
                A a3 = new A(){};
        }
        void m3(){
                A a4 = new A(){};
        }
        void m4(A a){}

        public static void main(String[] args) {
                A a5 = new A(){};
                new A(){};

                m4( a5 );

                m4( new A(){} );
        }
}
```

Q) How many anonymous inner classes are created for the interface A in this B class?
A) 7

Q) After compiling B class how many *.class* files are generated?
A) 9 .class files are generated by compiler
=> *A.class + 7 innerclasses + B.class*

Q) What is the naming conventions of inner classes? A) *outerclassname$<n>.class*
Where <n> is an integer number starts with 1 and incremented by 1 for every new anonymous inner class. Below are the .class files

|-> A.class
|-> B$1.class
|-> B$2.class
|-> B$3.class
|-> B$4.class
|-> B$5.class
|-> B$6.class
|-> B$7.class
|-> B.class

Q) Write down anonymous class for calling below method?

```
class Example{
        void m1(Runnable r){
                r.run();
        }
}
class Test{
        public static void main(String[] args){
                Example e = new Example ();
                e.m1( new Runnable(){
                        public void run(){
                                System.out.println("run");
                        }
                });
        }
}
```

Runnable is a predefined interface given for creating custom thread with a method
public void run();

Explanation:

In c.m1(new Runnable(){-----}); method call compiler generates a new class as a subclass of Runnable interface with the name "Test$1" and places run() method logic in this class's ".class" file, and then it places "Test$1" class object as argument.

JVM executes run() method from anonymous class Test$1

Q) Find out which statement creates anonymous class for Thread class?

Thread class is a predefined class it is sub class of Runnable interface.

```
class Test{
        public static void main(String[] args){
                Thread th1 = new Thread();
                Thread th2 = new Thread(){};

                e.m1( new Thread(){} );

                e.m1( new Thread() );

                e.m1( new Thread(){
                        public void run(){
                                System.out.println("Hi");
                        }
                });
        }
}
```

In Thread class run() method implemented without logic. So if we pass Thread class object directly or its subclass object without overriding run() method to Example class m1() method we do not see any output on console. So in the above program in first two times m1() method calls we not do see output on console and in third time m1() method call we get output *Hi*

Q) Can we store anonymous class object in its own referenced variable?

For example

```
Eample$4 e4 = new Thread(){};
```

A) No, not possible compiler throws below CE: Cannot find symbol

Q) What type of members are allowed in anonymous class?

A) In anonymous class

1. We can only define NSVs, NSBs, NSMs.
2. We cannot define constructor because class name is unknown.
3. We cannot define static members it leads to CE: inner classes cannot have static declarations

For example

```
new Thread(){
        //static int a = 10;
        int a = 10;
        //static void m1(){}
        void m2(){}
        //static{}
        {}
};
```

But compiler places constructor in anonymous class when its .class file is generated.

For more information decompile anonymous class's .class file by using "javap" tool.

Q) Is below program compiled?

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

                new Thread(){
                        System.out.println("Hi");
                };
        }
}
```

A) No, Sopln() statement leads to CE: <identifier> expected, because it is placed at class level, anonymous class is not a method, is it subclass.

The statements placed directly in {} of anonymous class are placed at class. So the only allowed statements are variable creation and method definitions

Q) Is below program compiled?

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

                new Thread(){
                        void m1(){
                                System.out.println("Hi");
                        }
                };
        }
}
```

A) Yes, this program compiled fine

Q) Is below program compiled?

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

                new Thread(){
                        int a = 10;
                        void m1(){
                                System.out.println("Hi");
                        }
                };
        }
}
```

A) Yes, this program compiled fine

Q)What is the output of the below program?

```
class Test{
        public static void main(String[] args){
                new Thread(){
                    void m1(){
                        System.out.println("Hi");
                    }
                };
                System.out.println("Hello");
        }
}
```

Accessing outer class and enclosing method members from anonymous class

Outer class all members are accessible, and like as local inner class, from anonymous class also we cannot access non-final variables and parameters of enclosing method

Find out compile time error in the below program:

```
class Example{
        static int a = 10;
        int x = 20;

        void m1(){
                int p = 30;
                final int q = 40;

                new Thread(){
                        void m1(){
                                System.out.println(a);
                                System.out.println(x);
                                System.out.println(p);
                                System.out.println(q);
                        }
                };
        }
}
```

Here ; is mandatory, because it is an object creation statement

Accessing anonymous class members from outer class and outside outerclass

We cannot access anonymous class members from its enclosing method or from its outerclass as it does not have name. To access anonymous class members they must be called from its super class overriding members. *Find out compile time errors in the below program*

```
interface Example{
        void m1();
}
class Test{
        public static void main(String[] args){
                Example e = new Example(){
                        public void m1(){
                                System.out.println("Anonymous overriding method m1");
                                m2();
                        }
                        public void m2(){
                                System.out.println("Anonymous Own method m2");
                        }
                };
                e.m1();
                e.m2();
        }
}
```

Q) What are the differences between Local inner and anonymous class?

Local Inner Class	Anonymous Class
1. Local inner class has explicit name given by Developer	1. Anonymous class does not have explicit name, its name is given by compiler
2. Local inner class is individual class	2. Anonymous class is a subclass of the class mentioned after "new" keyword
3. Local inner class ending with ";" is optional	3. Anonymous class ending with ";" is mandatory.
4. Local inner class object should be created by developer	4. Anonymous class object is created automatically by JVM
5. Local inner class cannot be defined as method argument	5. Anonymous class can be defined as method argument, this is the main purpose of it
6. In local inner class we can define all four non-static members including constructor	6. In anonymous we cannot define constructor, it is automatically added by compiler
7. The only allowed modifiers are *final, abstract, strictfp*	7. No modifier is allowed
8. Local inner class members specific members are accessible from its enclosing method by using its referenced variable	8. We cannot access anonymous innerclass specific members as it does not have name.

Interview points on anonymous inner class:

1. It is a nameless class which is a subclass of other existed class/interface created at class level or outer class method level or as method argument.

2. Syntax to create anonymous class: new <outer classname/interface name>(){}

3. A separate ".class" file is created to store its bytecodes with "outerclassname$<n>". Here <n> is number stats with 1 and incremented by 1 for every anonymous class.

4. Most of the cases anonymous class is used to pass outer class subclass object by overriding one of its methods or all methods. Anonymous class is most useful in registering AWT, and Swing components events.

5. No modifier is allowed

6. Only Non-static varaibles, non-static blocks, non-static methods are allowed to define. We cannot define constructor in anonymous class as it is nameless class, but compiler provides constructor in anonymous class

7. All outer class members including private members are allowed to access from inner class directly if it is defined in non-static method. If it is defined in static method outer class non-static methods must be called using outer class object.

8. Only final local variables and final parameters of enclosing method's are accessible from anonymous inner class.

9. Inner class members are not allowed to call from outer class. We are allowed to call anonymous innerclass members only from its super class overriding methods.

10. We cannot differentiate outerclass method local variable from anonymous class variable if both have same name.

Below table shows all 8 interview questions answers on all 4 inner classes at a glance

Definition:

- The inner class created at class level with static keyword is called static inner class.
- The inner class created at class level without static keyword is called static inner class.
- The inner class created inside outer class method is called local inner class.
- It is a nameless class which is a subclass of other existed class/interface created at class level or outer class method level or as method argument.

Syntax:

Static innerclass:

```
class A{
        static class B{}
}
```

Non-static innerclass:

```
class A{
        class B{}
}
```

Method Local inner class:

```
class A{
        void m1(){
                class B{}
        }
}
```

Anonymous class of interface Example:

```
interface Example{}

class A{
        Example e1 = new Example(){};
        void m1(){
                Example e2 = new Example(){};
        }
        void m2(Example e){}
        public static void main(String[] args){
                A a = new A();
                a.m2( new Example(){});

        }
}
```

.class files name

- *Static innerclass*: outerclassname$innerclassname.class
- *Non-static innerclass*: outerclassname$innerclassname.class
- *Method local innerclass*: outerclassname$<n>innerclassname.class
- *Anonymous innerclass*: outerclassname$<n>.class

Allowed modifiers

- *Static innerclass*: private, protected, public, final, abstract, strictfp.
- *Non-static innerclass*: private, protected, public, final, abstract, strictfp
- *Method local innerclass*: final, abstract, strictfp
- *Anonymous innerclass*: no modifier

Q) If we declare non-static inner class as static, is it leads to CE?

A) No CE, it becomes static inner class

Q) If we declare method local inner class as static, is it leads to CE?

A) Yes CE, because static keyword is not allowed for local members.

Advantage of innerclass:

- In an object development, we can provide more separation of code within the same class
- We can access private members of an enclosing class directly. It reduces lot of code development in event handling.
- It provides quick implementation to an interface, creating and sending its object as an argument *or* return type of a method with less code.

Disadvantage of innerclass:

- Not readable for beginners, but once you understand the usage of innerclass you will enjoy using innerclass more rather than using outer classes.
- More .class files are created with different name, but it is not considerable.

Allowed members

- *Static innerclass*: All 8 static and non-static members, and abstract method.
- *Non-static innerclass*: Only all 4 non-static members, and abstract method.
- *Method local innerclass*: Only all 4 non-static members, and abstract method.
- *Anonymous innerclass*: Only non-static variable, block, and method.
  We cannot define explicit constructor as it is nameless

Accessing outer class members

- *In Static innerclass*: All outer class members including private members accessible
  - All static members are accessible directly by their name.
  - All non-static members are accessible only with outer class object
- *Non-static innerclass*: All outer class members including private members accessible directly by their name.
- *Method local innerclass*: All outer class members including private members accessible
  If it is defined in static method
  - All static members are accessible directly by their name.
  - All non-static members are accessible only with outer class object

  If it is defined in non-static method
  - All outer class members are accessible directly by their name

  Only final local variables and final parameters of enclosing method are accessible
- *Anonymous innerclass*: It is same as method local inner class.

Accessing innerclass members from outerclass and outside outerclass

- *Static innerclass:*
  From outerclass: all members including private members are accessible
  - All static members are accessible by using *innerclass name*
  - All non-static members are accessible by using *innerclass object*

  From outside outerclass: all non-private members are accessible
  - All static members are accessible by using

    outerclassname.innerclassname.membername

    For example: A.B.m1()

  - All non-static members are accessible by using *innerclass object*

    outerclassname.innerclassname var = new outerclassname.innerclassname();
    var.membername;

    For example:

    ```
    A.B b = new A.B();
    b.m1();
    ```

- *Non-static innerclass*: contains only non-static members so they are accessible only by using innerclass object.
  From outerclass:
  - All members including private members are accessible by using *innerclass object*

  From outside outerclass:
  - All non-private members are accessible by using *innerclass object*

    outerclassname.innerclassname var = outerclasobject.innerclassobject();
    var.membername;

    For example:

    ```
    A.B b = new A().new B();
    b.m1();
    ```

- *Method local innerclass:* Its members are not allowed to call from outer class because it is local to a method.
  Inside enclosing method
  - We are allowed to call local inner class members including private members only within its enclosing method that to after its definition with its object

  Outerclass and outside outerclass
  - We must pass its objects as return type or argument by using its super class name, and its specific members must be called in the superclass overriding method.
- *Anonymous innerclass:* We cannot call its members anywhere in the program directly, they are accessible from enclosing method, or outerclass and outside outer class only by calling them from only its overriding method.

Differentiating outerclass members from innerclass member if both have same name:

- *In Static innerclass*:
  - *Outerclass static members*: Outerclassname.membername
  - *Outerclass non-static members*: Outerclassnameobject.membername
- *In Non-Static innerclass*:
  - *Outerclass static members*: Outerclassname.membername
  - *Outerclass non-static members*: Outerclassname.*this*.membername
- *In method local innerclass*:
  If it is defined in static method
  - *Outerclass static members*: Outerclassname.membername
  - *Outerclass non-static members*: Outerclassnameobject.membername

  If it is defined in non-static method
  - *Outerclass static members*: Outerclassname.membername
  - *Outerclass non-static members*: Outerclassname.*this*.membername
- *In anonymous innerclass*: Same as method local innerclass

Q) Can we create inner class name with outerclass name?

A) No, it leads to compile time error. Because it is not possible to differentiate outer class members from innerclass members if both have same name.

```
1. class A {
2.       class A{}
3.       void m1(){
4.               class A{}
5.       }
6. }
```

At line# 2 and 4 leads to CE:
class A is already defined

Q) Can we create local innerclass name with other innerclass name?

A) Yes, there is no compile time error. Because there is not link between inner classes.

```
1. class A {
2.       class B{}
3.       void m1(){
4.               class B{}
5.       }
6. }
```

This program compiled fine, No CE
.class files name
A.class
A$B.class
A$1B.class

Q) Can innerclass derived from other classes/interfaces if so from how many?

- Static, non-static, and method local inner classes can extend from one super class and implement from multiple interfaces at a time as they have explicit name.
- But anonymous class can extend exactly one class or implement exactly one interface

Chapter 18

OOPS Fundamentals & Principles

- In this chapter, You will learn
  - Types of programming languages
  - Why OOPS?
  - OOPS definition
  - Building blocks of OOP Language
  - How a real-world object is represented using program?
  - How one object can communicate with another object?
  - Different OOP models
  - Why static and non-static variables
  - Class design by following OOP designs
  - Protecting data – *Encapsulation*
  - Reusing existed data – *Inheritance*
  - Types of inheritance
  - Providing multiple forms for a method – *Polymorphism*
  - Advantages of Runtime polymorphism
  - Hiding implementation details – *Abstraction*
  - What is a specification?
  - Implementing specification using interface?
  - Difference between interface and abstract class
  - Understanding Real world project design and its implementation using *ATMCard* and *ATMMachine* specifications.
- By the end of this chapter
  - You will learn designing all real world objects in Java with effective memory usage
  - You will be in a position to design and develop OOP based project.
  - You will become experienced Java Developer with fundamentals.

Interview Questions

By the end of this chapter you answer all below interview questions

- Types of programming languages

Creating real-world object in program

- Why OOPS?
- OOPS definition.
- Building blocks of OOP Language
- Procedure to create real-world object
- Syntaxes to represent object properties and behaviors
- Difference between throw, throws, return
- Object characteristics
- Definition of class, object, instance and their relationship.
- Different OOP models
- Why static and non-static variables?
- Designing a class by following OOP designs

Carrying object's data

- Types of datatypes
- Limitation PDTs, Need of RDTs
- Sending and returning single and multiple values from one method to another method as argument and return type
- Factory design pattern

Right design to initialize object's properties

- Creating a class with and without values, and their affect in creating instances
- Constructor and need of constructor
- this *and* super keywords

Right design for saving memory in storing object's properties value

- How many instances can we create for an object from a class?
- Variables and types of variables
- 2 Scopes
- Design to store object properties and object operation values
- What does happen when a method is called?
- What is Current Object and Argument Object, and where they are stored?

Establishing Relations between objects

- IS-A, HAS-S, USES-A relations
- extends, implements keywords

OOPS models to create real-world object

- Single Class -> Multiple instances
- Single Class -> Multiple Classes -> Multiple instances

OOPS Principles

1. Encapsulation
   - Definition of Encapsulation
   - How can we develop encapsulation in Java?
   - What is problem if we do not implement encapsulation in projects?
2. Inheritance
   - Definition of Inheritance
   - What happened if we develop inheritance?
   - What exactly we are doing through inheritance?
   - What should we do if superclass method functionality is not satisfying subclass requirement?
   - If we call overriding method using subclass object from which class is it executed?
   - UML notations and types of inheritance
   - Why Java does not support multiple inheritance?
3. Polymorphism
   - Definition of Polymorphism
   - Types of Polymorphisms
     - Compile-time polymorphism
     - Runtime polymorphism
4. Abstraction
   - Definition of abstraction
   - What is a specification?
   - What is service provider?
   - What is the right design to develop a class to call methods of an object to use its operations?
   - What is coupling, loose coupling, tight coupling?
5. Sample project implementation to understand.
   - Advantage of Runtime polymorphism
   - Need of Inheritance
   - When super class and sub class should be defined in project
6. Develop a project to use all types of ATMCard objects from a ATMMachine
7. Developing GSM Mobile functionality to understand how abstraction ensures all OOPS principles work together to give final shape to the project, and it gives real-time project development experience.
8. What is the right design to enhance the project to add more operations to an existed business object? *For example* what is the right design to add 3G operation to SIM object?
9. Final Project on oops to understand, when to use interface, abstract class, concrete class, final class, and abstract methods, concrete methods, final methods.

Types of programming languages

All available programming languages are divided into three types based on the features they are supporting.

1. Monolithic
2. Structured or Procedural
3. Object-Oriented programming language

Introduction to OOPS

OOPS stands for "Object Orient Programming System". It is a technique, not technology. It means, it does not provide any syntaxes or API, instead it provides suggestions to design and develop objects in programming languages.

Why OOPS?

OOPS is a methodology introduced to represents real world objects using a program for automating real-world business by achieving security because business need security.

All living and non-living things are considered as object. So the real-world objects such as Person, Animal, Bike, Computer, etc... can be created in OOP languages.

Q) Why do we need real-world objects in program?

Because real-world object is part a business. As we develop software for automating business we must also create that business related real-world objects in the project.

For example: To automate Bank business we must create real-world objects like - Customer, Manager, Clerk, OfficeAssistant, MarketingExecutive, Computer, Printer, Chair, Table, AC etc... Along with Bank object we must also create all above objects because without all above objects we cannot run Bank business. So technically we call above objects are business objects.

Definition of OOP:

OOP is a methodology that provides a way of modularizing a program by creating partitioned memory area for both data and methods that can be used as template for creating copies of such modules (objects) on demand.

Unlike procedural programming, here in the OOP programming model, programs are organized around objects and data rather than actions and logic.

Building blocks of OOP:

The building blocks of OOP are

- class
- object

Every Java program must start with a class, because using class only we can represent real-world objects like Person, Bike, Animal, etc ...

Definition of Class, Object and their relationship:

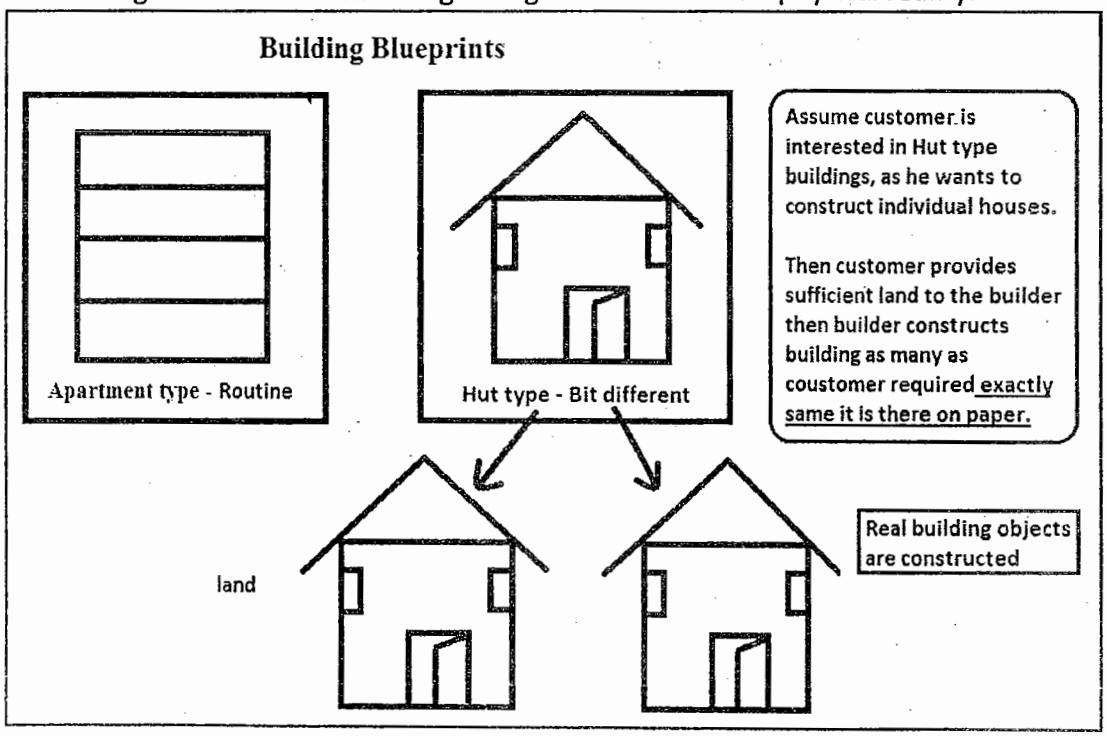
Definition of class

- A class is a specification or **blue print** or template of an object that defines what goes to make up a particular sort of object.
- Thus a class is a logical construct, an object has physical reality.
- A class defines the structure, state and Behaviour (data & code) that will be shared by a set of objects. Hence each object of a given class contains the structure, state and Behaviour defined by the class.
- When we create a class, we'll specify the data and code that constitute that class. Collectively these elements are called members of that class. Specifically, the data defined by that class are referred to as member variables or instance variables or attributes. The code that operates on that data is referred to as member methods or methods.

Definition of object

- Object is the physical reality of a class.
- Technically object can be defined as "It is an encapsulated form of all non-static variables and non-static methods of a particular class".
- An *instance of a class* is the other technical term of an object.
- The process of creating objects out of a class is called instantiation
- Two objects can be communicated by passing messages (arguments).

Below diagram shows the meaning of logical construct and physical reality.



More explanation of definition of object, class, instance and their relationship

Definition of object:

Any real-world living and non-living thing is called object

- For example Bike, Car, Dog, Computer, BankAccount

Definition of class:

A class is a logical construct of the object that defines/represents object logically with that objects properties and Behaviours.

- For example Bike{ bikeNumber, model, color, start(), move(), stop() }.

Definition of instance:

An instance is a single, unique memory allocation of a class that represents that object physically with specific values:

- For example Bike [8192, "Pulsar 180", Color.RED]

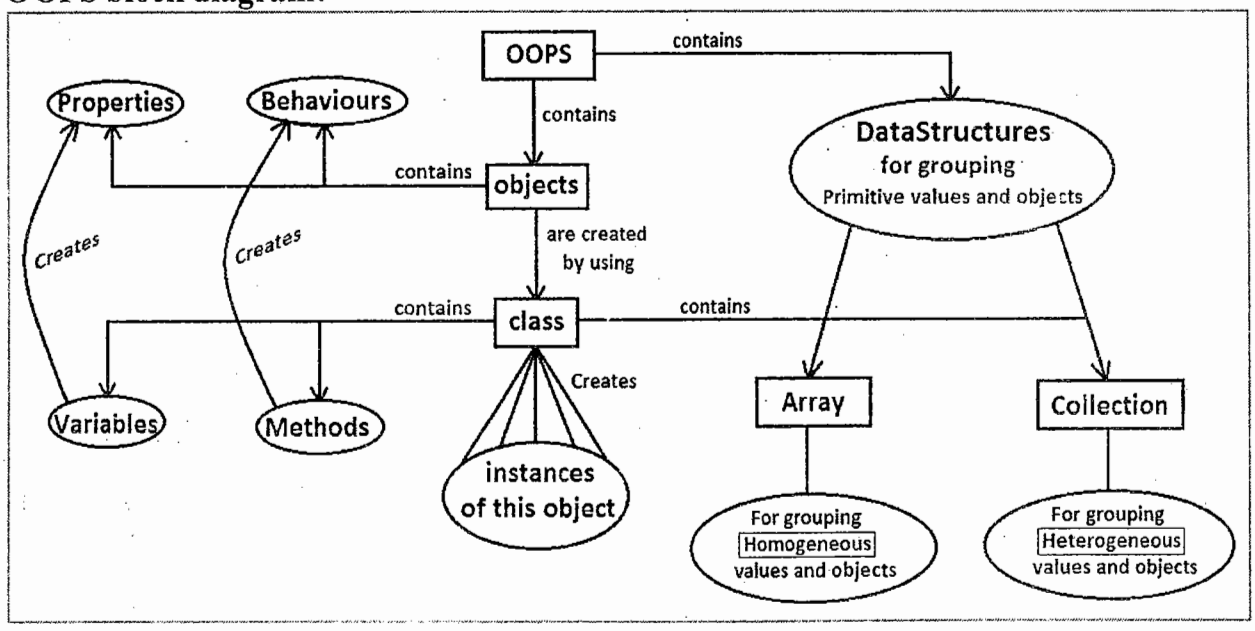
Technically speaking

- "class Bike{}" - creates Bike object logically
- "Bike b = new Bike()" - creates bike object physically, nothing but instance (memory)

Q) Is object and instance both are same?

No, memory allocated for creating object physically with specific values is called instance. This is the reason object is also defined as "instance of the class"

OOPS block diagram:



Object's characteristics:

A real world object contains below three characteristics

- State : describes the data stored in the object
- Behaviour: describes the methods in the object's interface by which the object can be used
- Identity : the property of an object that distinguishes it from other objects

- State is its properties - like name, height, weight, etc...
- Behaviour is its actions - like sleep, walk, run, eat, etc...
- Identity means identity - name

More explanation:

State

- Instance variables value is technically called as object state.
- An object's state will be changed if instance variables value is changed.
- A change in a state of an object must be a consequence of messages sent to the object.
- It means non-static variables must be changed only via setter methods not directly.

Behaviour

- Behaviour of an object is defined by instance methods.
- Behaviour of an object is depends on the messages passed to it.
- Message is passed to objects through methods.
- So an object Behaviour is depends on the instance methods.

Identity

- Identity is the hashcode of an object. It is used for differentiating one object from other object. It is implemented by JVM by converting the internal address of the object into an integer number. Since every object has different reference, each object will have a unique identity.

- The hashCode of the object is retrieved by using a method "hashCode" which is defined in java.lang.Object.

- *Hashcode is used when object is stored in Hashtable.* Refer Collections chapter.

Creating real world objects in OOPS:

By using "class" keyword we can create real-world object in OOP Languages and inside that class we can implement that object's operations/ Behaviours.

So to create real world object in OOP languages either in Java/.Net we must create a *class* specific to that object with that object name.

For example:

```
class Bank{}
class Customer{}
class Manager{}
```

As per OOPS, we can only represent real objects in programming. Representing means we can only store that object's properties and Behaviours but not structure.

Here,

- Properties means object's data/values
- Behaviours means object's actions (operations)

So inside a class we should create

- variables to store data (properties),
- methods to implement operations

OOPS terminology	PL terminology
object	Class
properties	variables
Behaviours/operations/actions	methods

Syntax to represent real-world object

```
class <object name>{
        ----------
        ----------      List of properties of
        ----------      this object
        ----------

        ---------
        ---------       List of Behaviours of this object
        ---------       i.e.; business operations
        ---------
}
```

Syntax to define object's properties and Behaviours (operations)

```
class <object name>{

        //property creation syntax
        <Accessibility Permissions> <datatype> <value type name> = <value>;

        //Behaviour creation syntax
        <Accessibility Permissions> <output> <operation name>(<input>) <failure errors> {
                Validations and Calculations
        }
}
```

Behaviour terminology	Method terminology
Operation name	Method name
Input	Parameters
Success Output	Return type
Failure Output	Exceptions
Validations and Calculations	Logic

Keywords for returning *success output* and *failure output*

OOP languages has two keywords

1. *return* - for returning success output
2. *throw* - for throwing failure output (exceptions)

Use of:

- **return type**: To inform to user programmer what type of output value coming out from the method.
- **method name**: to inform to user programmer what type of operation we are developing
- **parameters**: To inform to user programmer What type of and how many values must be passed to execute this method.
- **throws keyword**: To inform to user programmer what type of and how many exceptions (failed output) coming out from this method when method execution is terminated due to some wrong input.

Below application developing BankAccount object in java

```
//BankAccount.java
public class BankAccount{
    //BankAccount properties (values required to perform operations)
    private long accNo;
    private double balance;
    private String username;
    private String password;

    //Parameterized constructor to initialize instance
    public BankAccount(long accNo, double balance, username, String password){
        this.accNo      = accNo;
        this.balance    = balance;
        this.username = username;
        this.passwrod  = password;
    }
    //BankAccount Behaviours (an operation to complete a transaction)
    public void deposite(double amt) throws InvalidAmountException{

        if(amt <= 0){
            throw new InvalidAmountException();
        }

        balance = balance + amt;
    }
    public double withdraw( double amt ) throws InsuffientFundsException{

        if (balance < amt){
            throw new InsuffientFundsException();
        }

        balance = balance - amt;
        return amt;
    }
}
```

Steps to create real world objects in programming:

To create real world object in Java we must

1. Identifying the real- world object (say Bike)
2. Design that object as program using "class"
3. Find its properties and create them as variables inside that class
4. Find its Behaviours and create them as methods inside that class
5. Creating that object physically in memory using "new" keyword and constructor

We have developed BankAccount object by following above steps. We have finished first four steps. Now let us create new account instances from the above class BankAccount

//Cleark.java

```
public class Clerk{
        public static void main(String[] args){
                BankAccount acc1 = new BankAccount (1, 5000, "Hari", "Krishna");
                BankAccount acc2 = new BankAccount (2, 5000, "Rama", "Krishna");
        }
}
```

Different OOP models

We have below two design models to represent real world objects

They are:

Single class -> Multiple instances of the object

Single class -> Multiple classes -> multiple instances of the objects

Single class -> Multiple instances of the object

If multiple objects of same type have same properties and same Behaviours implementation then we develop by using this design. It means we develop a single class to represent that object and from this class we create new instance for each object's with its specific values.

Common and Individual properties:

In one category of objects like Person objects, Animal objects, Vehicle objects, Shape objects, we can find some common properties and individual properties

For Example, if we consider Person objects, every Person object has below properties

1. eyes
2. ears
3. hands
4. legs

5. name
6. height
7. weight

First four properties are common for all Person objects

Next three properties are individual to every Person objects.

Need of Static and non-static variables:

Q) Why do we have two types of class level variables- static & non-static?

- static variables are given to store common properties.
- non-static variables are given to store individual properties.

So, static variables have only one copy of memory location, and they are shared by all objects of that class. non-static variables have duplicate copies of memory one per each object separately.

In designing Person object,

- We create 1st four properties as static variables, so that only one copy of memory is created.
- We create next three variables as non-static variables, so that they have separate copies of memory for each Person object.

Q) What is the problem if we store common values using non-static variables?

A) No CE, No RE, it is waste of memory.

In above Person example, if we store first four properties using non-static variables, for every object creation 16 bytes of memory is consumed. Let us say, if we created 100 objects, 1600 bytes of memory is consumed to store same value, stupid design.

If we create those variables as static, only 16 bytes of memory is consumed for all objects. Static variables those are representing object properties must be declared as final to ensure their values are not modified.

So to create class with this design we must develop class as below

1. A class with the object name
   For example: Person, Computer, BankAccount
2. Static variables to store common values of all instances of this object
   For example: in case of Person object *eyes, ears, legs, hands*
3. Non-static variables to store instance specific values of each instance
   For example: in case of Person *name, height, weight*
4. A parameterized constructor to initialize on-static variables with instance specific values
5. Further we must create all required instances of this object from this same class

Below application shows creating single class for creating different Person objects

//Person.java

```
public class Person{

        static final int eyes    = 2;
        static final int ears    = 2;
        static final int hands   = 2;
        static final int legs    = 2;

        String name;
        double height;
        double weight;

        public Person(String name, double height, double weight){
                this.name    = name;
                this.height  = height;
                this.weight  = weight;
        }
}
```

//Test.java

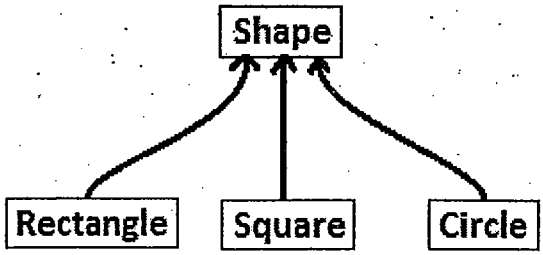
```
class Test{
        public static void main(String[] args){
                Person p1 = new Person("Dhoni", 5.9, 85);
                Person p2 = new Person("Yuvi", 6, 90);
        }
}
```

Single Class -> Multiple Classes -> Multiple Instances

If multiple objects of same type have different properties and different Behaviours implementation then we must develop this design. In this design we must define separate class for each object creation with a super class for grouping all these classes as one category. This implementation is called *INHERITANCE*.

For example: All shape type of objects have their own properties and Behaviours area() and perimeter() logic. So we must develop different classes to represent each shape like Rectangle, Square, Circle, etc... deriving from a super class that represents shape category let us say "Shape". This class Shape groups all these objects to treat them as shape type of classes.

Below architecture shows design Shape objects



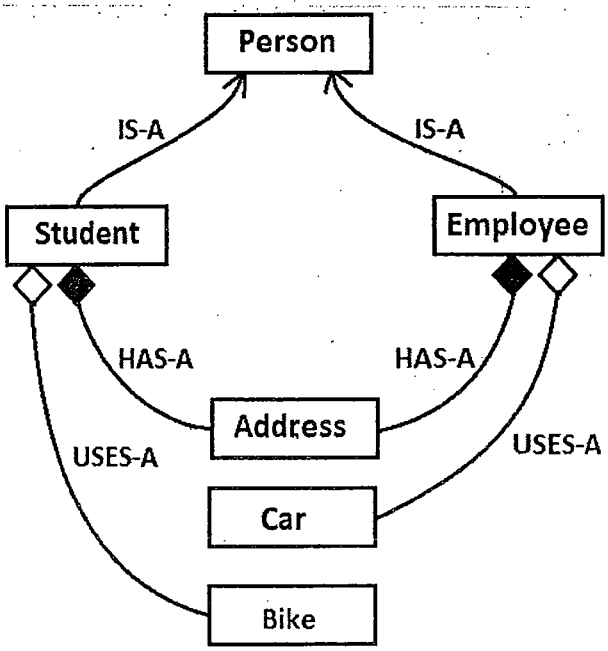
IS-A, HAS-A, USES-A relations

A business has many objects. All these objects must be related to each other.

Java supports we have three types of relations between objects

1. IS-A (Inheritance)
2. HAS-A (Composition)
3. USES-A (Aggregation)

Consider below objects and their relations UML diagram:



How should we read this relation?

Student IS-A Person
Employee IS-A Person

Student HAS-A Address
Employee HAS-A Address

Student USES-A Bike
Employee USES-A Car

Explanation:

1. IS-A:

We should establish IS-A relation between classes for grouping classes as a one category/family to establish child-parent relation. So that child classes objects can be used wherever parent class referenced variable is using.

In the above example: Student and Employee objects are grouped as one family of classes and they got the type Person. Then these two objects inherit the properties and Behaviours of Person. Now these objects are of type also Person, this is the reason we call them as

- Student IS-A Person
- Employee IS-A Person

How can we implement this IS-A relation in Java?

It is implemented by using below two keywords

1. extends
2. implements

- extends is used between two classes or two interfaces
- implements is used between a class and interface.

For example:

```
class Person{}
class Student extends Person{}
class Employee extends Person{}
```

2. HAS-A:

We should establish HAS-A relation between classes if one object cannot exist without another object. For example in the above example a Student or Employee objects cannot exist without Address. So we must establish HAS-A relation between these objects. This is the reason we call them as:

- Student HAS-A Address
- Employee HAS-A Address

How can we implement this IS-A relation in Java?

This relation is implemented by storing other object's instance using its non-static variable in our object's class. So we must create Address class type non-static variable to store its object in Student and Employee classes. Check below code:

For example:

```
class Address{}

class Student{
        Address add = new Address();
}
class Employee{
        Address add = new Address();
}
```

3. USES-A:

We should establish USES-A relation between classes if one object uses another object for performing one of its operations. For example in the above example Employee uses Car for traveling. So we must establish USES-A relation between these two objects. This is the reason we call them as:

- Employee USES-A Car for travelling

How can we implement this IS-A relation in Java?

This relation is implemented by storing other object's instance using parameter of the method in our object's class. So we must create *Bike* class type parameter in Student class in a method called *goingToCollegeBy* and *Car* class type parameter in Employee class in a method called *travel*. Check below code:

Below is the complete code of above relations:

```
class Bike{}
class Car{}

class Person{}
```

```
class Student  extends Person{
    Address add = new Address();
    Void goingToCollegeBy(Bike b)
}
```

```
class Employee extends Person{
    Address add = new Address();

    void travel(Car c){}
}
```

Develop an application to create Student and Bike objects for developing HAS-A relation between these two objects.

//Bike.java

```
class Bike{
        String  bikeNumber;
        String  bikeName;
        int     modelNumber;
        String  engineNumber;

        void engineStart(){}
        void move(){}
        void engineStop(){}
}
```

//Student.java

```
class Student{
        int     sno;
        String  sname;
        String  course;
        double fee;

        void goingToCollegeBy(Bike b){
                SopIn(this.sname + " is going to college by "+ b.bikeName + " bike");
        }
}
```

//Parent.java

```
class Parent{
        public static void main(String[] args){
                //buying bike (creating Bike object)
                Bike pulsar = new Bike();
                pulsar.bikeNumber    = "8192";
                pulsar.bikeName      = "Pulsar 180";
                pulsar.modelNumber = 2007;
                pulsar.engineNumber = "443322";

                //Creating student object
```

```
        Student hk = new Student();
        hk.sno      = 1;
        hk.sname    = "HariKrishna";
        hk.course   = "Java";
        hk.fee      = 5000;

        hk.goingToCollegeBy( pulsar );
    }
}
```

Practical explanation – self reading notes

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

Real-world objects share two characteristics: They all have state and Behaviour. Dogs have state (name, color, breed, hungry) and Behaviour (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and Behaviour (changing gear, changing pedal cadence, applying brakes). Identifying the state and Behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible Behaviour can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible Behaviours (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and Behaviour (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related Behaviour. An object stores its state in *fields* (variables in some programming languages) and exposes its Behaviour through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.
By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

- **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

- **Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- **Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- **Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

You will understand more about these benefits in the next section.

OOPS Principles

So far you have learn how to design a class to represent a real world object. Now we learn designing object behavior by protecting class data, reusing existed data, using services dynamically from different classes of same type. By the end of this chapter you will design and develop a real-world business based project yourself.

Definition of OOPS Principles

OOPS principles are *design patterns* those suggest how we should develop a program to organize and reuse it from other layers of the project effectively with high scalability.

Scalability means developing a project to accept future changes without doing major changes in the project, that small change also should be accepted from external files like property files or XML files. Scalability is achieved by developing classes by integrating them in loosely coupled way.

We should develop project with Scalability as there will be a growth in business, according to the growth in the business we must add required changes to the project with minimal modifications.

As a developer you must remember that in initial stage of business customer never make a significant investment. As business grows customer increase investment, according to the growth new requirements are added to the project. To add those new requirements we should not redesign the project entirely.

So we must design project by following OOPS principles strictly even they are not need at this state but for accepting future changes.

How many OOPS principles do we have?

We have **3** OOPS principles. They are:

1. Encapsulation
2. Inheritance
3. Polymorphism

Java also supports Abstraction, it is a supporting principle of OOPS that ensures all three principles are working together to give final shape of the project. Abstraction suggests a way to develop class with loose coupling.

What type of programming languages comes under OOP system?
The programming language that supports implementing all above 3 OOP principles by supporting abstraction is called *Object Oriented Programming* language.

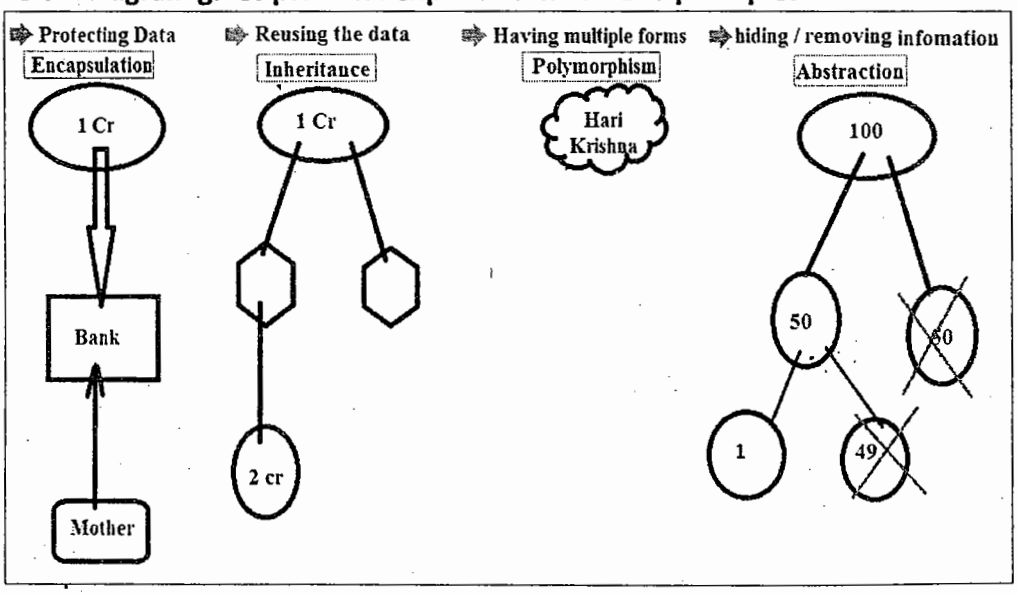
Encapsulation Definition:
The process of creating a class by Hiding internal data from the outside world; and accessing it only through publicly exposed methods is known as data **encapsulation/ data hiding.**

Inheritance Definition:
The process of creating a class to reuse exited class members *using our class name or object* is called inheritance. It can also be defined as it is a process of obtaining one object property to another object.

Polymorphism Definition:
It is a process of defining a class with multiple methods with same method name with different implementations is called polymorphism.

Abstraction Def:
The process of defining a class by providing necessary details to call object operation by hiding or removing its implementation details is called abstraction.

Below diagram gives pictorial explanation of *OOPS* principles



First diagram shows - we deposited money in bank to protect it and withdrawing it later with proper authentication - this is called encapsulation.

Second diagram shows - Reusing father's money in our own business to get more money or just spending it for enjoyment as if it was earned by us – this is called inheritance.

Third diagram shows – in this world there are many people having same name HariKrishna – this is called polymorphism.

Fourth diagram shows – There are 100 things among them we are taking only necessary one – this called abstraction.

In the next pages all above oops principles are explained with clear examples.

Encapsulation

The process of defining a class by hiding its data from direct access from the outside class members; and providing its access only through publicly accessible setter and getter methods with proper validations and authentications is called encapsulation.

How can we develop encapsulation in JAVA?

In Java, encapsulation is implemented

1. by declaring variables as **private,** to restrict it from direct access, and
2. by defining **one pair of public setter and getter methods** to access private variables.

- We declare variables as private to stop accessing them directly from outside class and
- We define public methods to access the same variable from outside class with proper validations, because if we provide variable access directly we cannot validate the data before storing it in the variable.

So by developing encapsulation we can protect or secure data

The below program explains developing a class by following encapsulation principle:-

//Bank.java

```
public class Bank {

        //hiding class data
        private double balance;

        public void setBalance(double balance){
                //add validation logic, to check data is correct or not
                this.balance = this.balance + balance;
        }
        public double getBalance(){
                //add validation logic, if needed
                return balance;
        }
}
```

//BankUser.java

```
class BankUser{
```

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

        Bank hdfcBank = new Bank();

        hdfcBank.setBalance(5000);
        System.out.println(icicibank.getBalance());
    }
}
```

What is the advantage in providing variable access via setter and getter methods?
We can validate user given data before it is storing in the variable. In the above program for balance variable ***–ve*** value is not allowed, so we can validate given amount value before storing it in *balance* variable. if we provide direct access to balance variable it is not possible to validate give amount value.

What is the problem if we do not follow encapsulation in designing a class?
We cannot validate user given data according to our business need and also future changes affect user programs. User programs must recompile and retest completely.

Consider an example "In a class we have variable that should be filled with a value by that class user" as shown in the below program. Assume in initial project requirement document customer did not mention that the application should not allow negative numbers to store. So we gave direct access to the variable and user can store any value as shown below.

```
class Example{
    int x;
}
```

```
class Sample{
    public static void main(String[] args){
        Example e = new Example();
        e.x = 50;
        Sopln(e.x);

        e.x = -10;
        Sopln(e.x);
    }
}
```

Assume in future customer wants application not allows negative numbers. Then we should validate user given value before storing it in variable.

Hence application architecture should as like below,

Declare variable as private, to stop direct access, and define setter method to take value from user. Then user invokes in this method to initialize variable. In this method we can validate the

passed value before storing it in variable. If it is **<0** we terminate assignment and informs the same to user, else we will store that data in variable.

Below program shows correct implementation of class by following encapsulation principle with required above validation logic.

```
class Example {
    private int a;

    public void setA(int a)    {
        if(a > 0){
            this.a = a;
        }
        else{
            Sopln("Do not pass negative number");
        }
    }
    public int getA() {
        return a;
    }
}
```

```
class Sample
{
        public static void main(String[] args)
        {
                Example e = new Example();
                //e.a = 50;  CE:
                //System.out.println(e.a); CE:

                e.setA(50);
                System.out.println(e.getA());

                e.setA(-6);
                System.out.println(e.getA());
        }
}
```

Now let us understand the problem

Since we have changed the code after application is used by several programmers, every one now should rebuild their applications to access variable through setter and getter methods. Otherwise their previous .class file code execution is failed with exception java.lang.IllegalAccessError.

It means since we are not followed encapsulation in developing class it leads lot of maintenance cost. Hence to avoid all these future problems we should always develop classes by flowing encapsulation principle, even though we do not have any validations before setting and getting variable, it is all required for future sake.

Advantage in designing classes by following encapsulation

In future user programs are no need to be recompiled due to the code change in setter and getter methods. Ultimately through encapsulation we are hiding implementation details from user.

Consider another example to understand need of encapsulation better, **For example**, Bike only has 5 gears, a method to change gears could reject any value that is less than 1 or greater than 5. If we provide direct access to variable we cannot reject storing a value <1 and >6.

Inheritance: As the name suggests, inheritance means to take something that is already made.

Inheritance is the process of defining a class to obtain other object members into our object to reuse or change that object members by using our class name or object as if they were defined in our class. The main purpose of inheritance is to obtain an existed class type to new classes so that new class objects can be using with existed class referenced variable.

It is the one of the most important features of Object Oriented Programming. It is the concept that is used for reusability and changeability purpose. Here changeability means overriding the existed functionality of the object or adding more functionality to the object.

Advantages of inheritance

Through Inheritance we can derive new classes from other classes

1. to obtain existed class type to new class
2. to reuse its members by using our class name or object
3. to change its one of the behaviors implementation by overriding method in our class.
4. to add more behaviors to that existed object by defining new methods in our class

For example

- Using the Bike as it is how you bought it from showroom is called reusing
- Changing that Bike as sports bike is called overriding existed functionality or adding new functionality

How inheritance can be implemented in JAVA?

Inheritance can be implemented in JAVA using below two keywords:

1. extends
2. implements

"extends" is used for developing inheritance between two classes or two interfaces, and *"implements"* is used for developing inheritance between interface, class.

Syntax:

```
class Example{}
class Sample extends Example{}
```

```
interface A{}
interface B extends A{}
class C implements A{}
```

The class followed by extends keyword is called *super class*, here Example is super class, and the class that follows extends keyword is called *sub class*, here Sample is sub class.

Super class is also called as Parent / Base class.
Sub class is also called as Child / Derived class

//The below program explains implementing inheritance between classes & Calling superclass members from subclass directly by their name and using subclass object.

```
//Example.java
class Example
{
    static int a = 10
    int x = 20;

    static void m1()
    {
        Sopln("Example m1");
    }

    void m2()
    {
        Sopln("Example m2");
    }
}
```

```
//Sample.java
class Sample  extends Example
{
    p s v main(String[] args)
    {
        Sopln(a);
        m1();

        Sample s = new Sample();
        Sopln(s.x);
        s.m2();

    }
}
```

In this program we accessed superclass

- static variable "a" and method m1() directly by their names and
- non-static variable "x" and method m2() with sub class object.

What exactly we are doing through inheritance?

By implementing inheritance we are establishing a relation between two classes and then extending one class scope to another class. So that other class members can be accessed directly from this class by their names as if they were developed in this class.

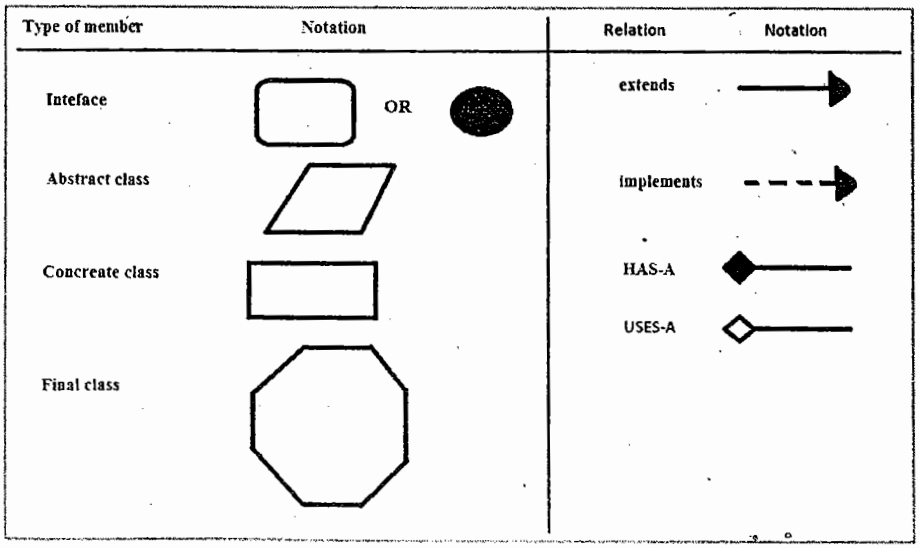
In the above program compiler and JVM first search for static variable "a" in main method, since it is not available here they search for it in Sample class, since it's not found in Sample class they search it in Example class, because Sample class scope is increased or extended to Example class.

Since Sample class scope is extended to Example class, we are able to access all Example class members directly from Sample as if they were defined in Sample class.

UML and types of inheritance:

UML stands for Unified Modeling Language. It is used for designing OOP based projects. It has different notations and diagrams to represent OOP members and their relations.

Below is the list of notations:



Types of Inheritance:

We have 5 types of inheritance

1. Single Level
2. Multi Level
3. Hierarchical
4. Hybrid
5. Multiple interfaces inheritance

Java does not support multiple inheritance, but it provides alternative to support multiple inheritance with interfaces.

Interview question

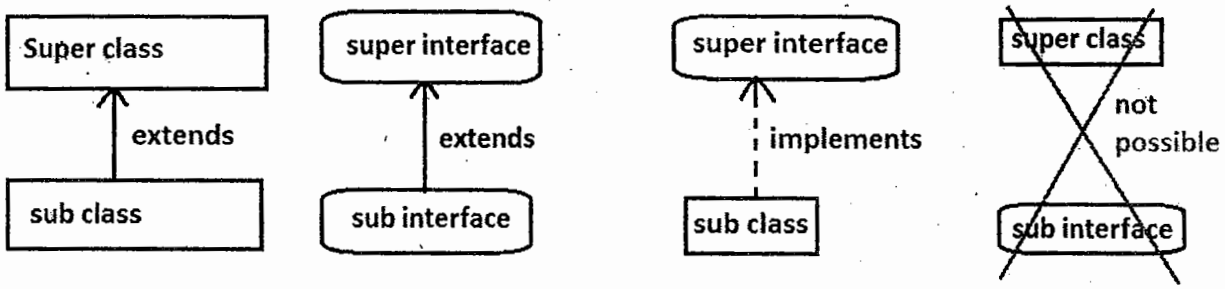
Q) Is Java supports multiple inheritance?

No

Single level inheritance:

If two classes or two interfaces or a class and interface participating in an inheritance, it is called single level.

Diagram:



- extends means reuse.
- implements means forcing to implement the body of abstract methods.
- so implements is not allowed between two interfaces, as interface cannot implement method.
- Also we cannot derive an interface from a class, because interface cannot have concrete methods.

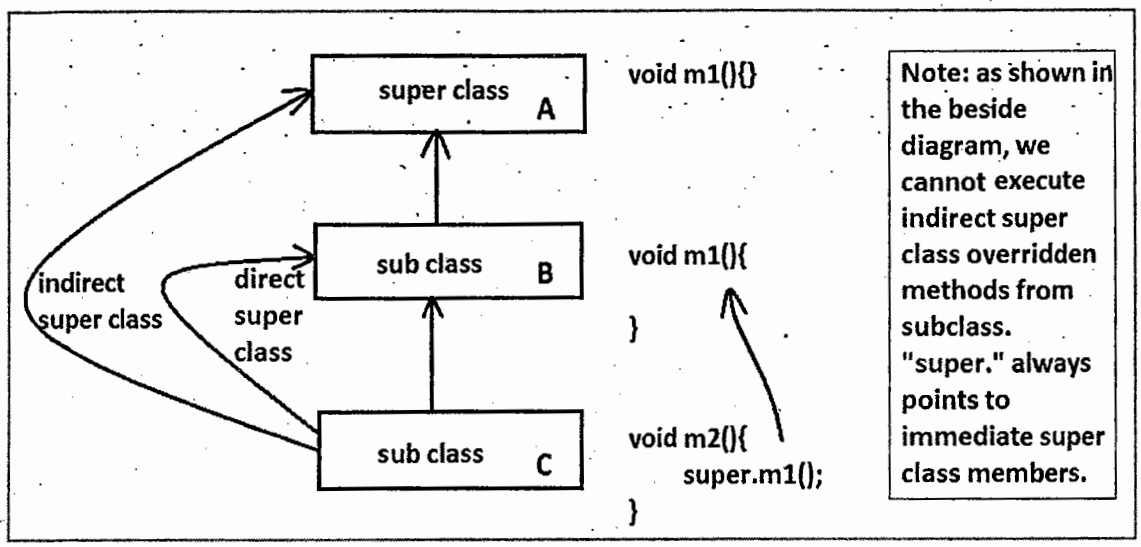
Q) In Java, can we develop a class without inheritance?

A) No, because by default every class is a subclass of java.lang.Object. So the default inheritance of Java is single level inheritance.

Multilevel inheritance:

If more than two classes are participating in inheritance relation vertically we called it as multilevel inheritance.

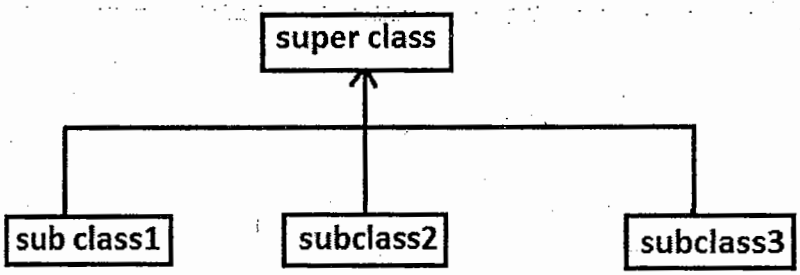
Diagram:



Hierarchical inheritance:

If we derive multiple subclasses from a single super class we call it as hierarchical inheritance.

Diagram:

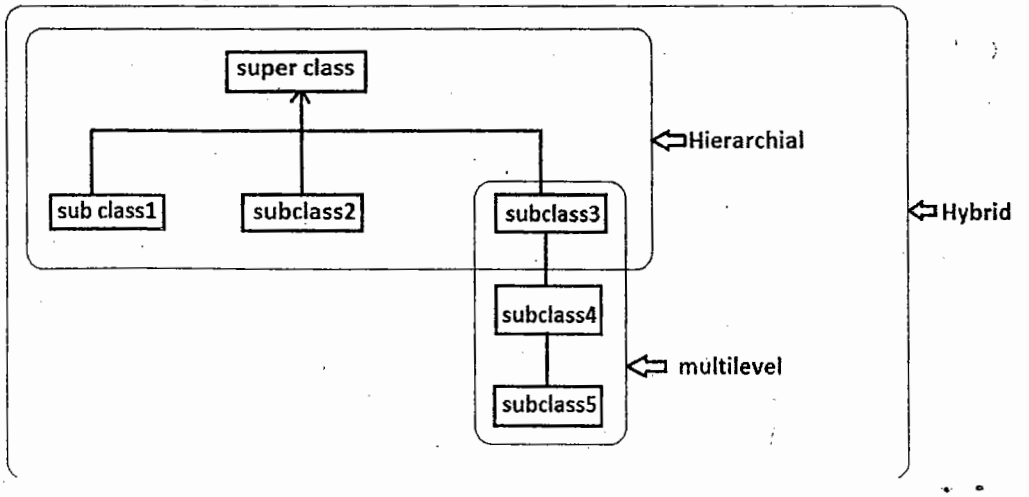


Q) As a subclass developer, can we stop creating a subclass from a non-final superclass in multilevel and hierarchical inheritances?

A) We can stop in multilevel inheritance by declaring our subclass as final class. But it is not possible to stop hierarchical inheritance because we cannot declare the superclass as final as it is developed by some other developer.

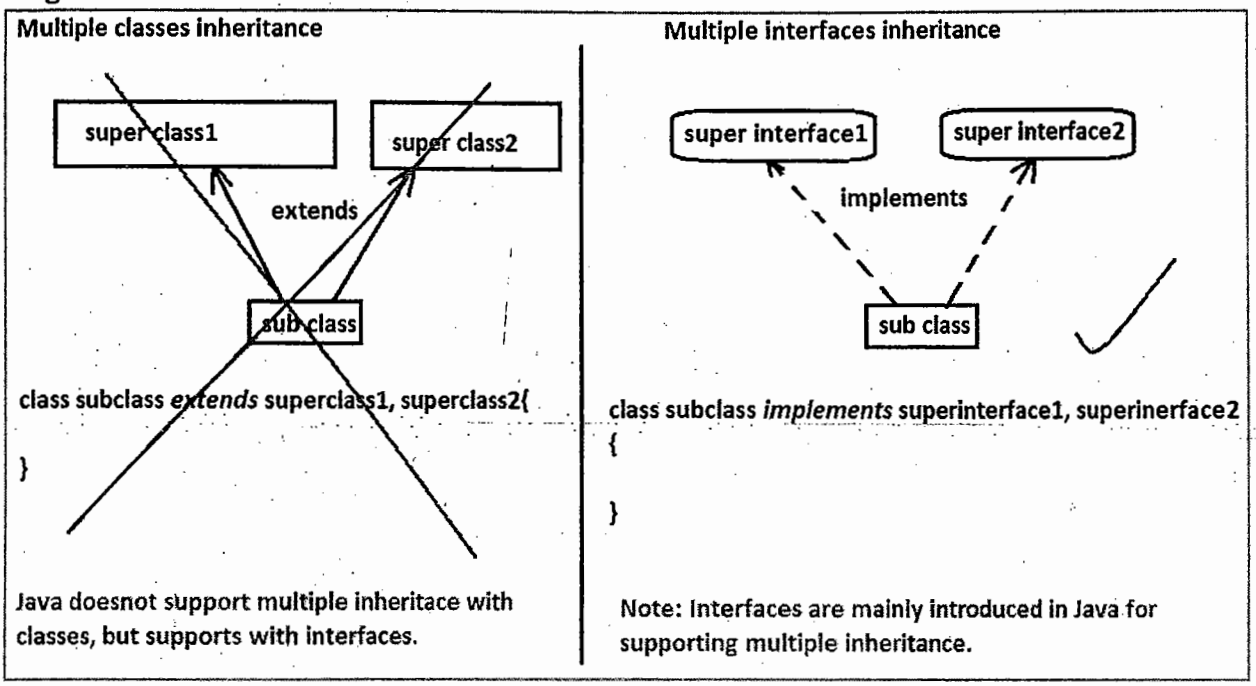
Hybrid inheritance:

Developing an inheritance by combining other types of inheritances is called Hybrid inheritance. Diagram:



Q) What is the default inheritance we develop in projects?
A) hybrid inheritance is the default inheritance we are developing in projects

Multiple inheritance:
Deriving a subclass from multiple superclasses is called multiple inheritance. Java does not support multiple inheritance with classes, but it supports with interfaces.
Diagram:



Also we can derive a class from single class and multiple interfaces.

```
class A{}
interface  B{}
class C extends A implements B {};
```

Is below syntax correct?

```
class C implements B extends A {};
```

A) No, it leads to CE, because in this syntax we must implement the method from interface then this implemented method becomes overriding method of superclass method if it contains the same method. So when we call it, it is executed from subclass, hence we are losing super class given functionality.

Q) Is a class deriving from any other class when it is only deriving from an interface?
A) Yes it is also deriving from java.lang.Object.
For example:

```
interface A{}
class B implements A{}

Changed by compiler as
class B extends Object implements A{}
```

Is this casting operation correct?

```
Object obj = new B();
A a = (A)obj;
```

Above two statements are correct, because every interface subclass is a subclass of java.lang.Object so compiler considering the coming object is of type A interface subclass, hence it compiles above casting statement. If the coming object from obj variable is not of type "A", JVM throws CCE.

Is below statement compiled and executed?

Case #1:

```
Object obj = "a";
A a = (A)obj;
```

A) No CE, but leads to CCE

Case #2:

```
String s = "a";
A a = (A)s;
```

A) It leads to CE: inconvertible types because String is not a subclass of A and more over there is no change of getting A subclass object from String referenced variable, because String is not default super class of all java classes.

Case #3:

Q) Can we call Object class methods by using interface referenced variable?

For example:

```
A a = new B();
a.toString();
```

A) Yes, we can call Object class methods by using interface, because that called method is executed from subclass of that interface, if not available it is executed from Object as it is the default superclass of every class.

Conclusion: Every interface subclass is a subclass of java.lang.Object class by default.
So we can cast Object class referenced variable to interface variable, and also we can call object class methods by using interface referenced variable.

So if we call a method by using interface referenced variable, compiler searches that method in interface first, if not available there it searches that method java.lang.Object classs, if that also not available, compiler throws CE: cannot find symbol

For example: identify compile time error

```
a.m1();
a.toString();
```

Identify in below cases is multiple inheritance possible or not?

Case #1: both interfaces has method with same prototype

Given:

```
inteface I1{ void m1(); }
inteface I2{ void m1(); }
```

A) Yes inheritance is possible, and subclass should implement m1() method only once.

For example:

```
class C implements I1, I2{
        public void m1(){}
}
```

Case #2: both interfaces has method with same name but with different parameters

Given:

```
inteface I1{ void m1(int a); }
inteface I2{ void m1(long l); }
```

A) inheritance is possible. Subclass should implement both methods because they are overloading methods, not overriding.

For example:

```
class C implements I1, I2{
        public void m1(int i){}
        public void m1(long l){}
}
```

Case #3: both interfaces have a method with same singnature, but with different return type.

Case #3.1: incompatible primitive return types

```
interface I1{ int m1(); }
interface I2{ long m1(); }
```

A) Inheritance is not possible because these two methods are considered as overriding methods in subclass, since they have different return types it is not possible to implement in subclass because violating overriding rule#2.

Case #3.2: compatible referenced return types (super and subclasses)

```
interface I1{ Object m1();}
interface I2{ String m1();}
```

A) Inheritance is possible and we should override String return type method as per covariant returns. If we override Object return type method it leads to

CE: m1() in C cannot override m1() in I2; incompatible return types
    found: Object
    required: String

```
class C implements I1, I2{
        //Object m1(){ return ""; } CE:
        String m1(){ return ""; }
}
```

Case# 3.3: incompatible referenced return types (siblings)

```
interface I1{ Integer m1();}
interface I2{ String m1();}
```

A) Inheritance is not possible.

Case #4: both interfaces have a method with same signature, but with different checked exceptions type.

Given:

```
interface I1{
        void m1() throws ClassNotFoundException ;
}
interface I2{
        void m1() throws InterruptedException ;
}
```

A) This case has three sub cases.
For more details check EH chapter

Case #5: both class and interface have method with same prototype.

Given:

```
class A{ public void m1() {} }
interface I1{ void m1(); }
```

A) Inheritance is possible and we no need to implement m1() method from interface as it is inheriting from A class with same prototype.

For example:

```
class C extends A implements I1{
}
```

Case #6: Both class and interface has a method with same signature but with different AM, i.e; class method is not public.

```
class A{ void m1(){} }
interface I1{ void m1(); }
```

A) Inheritance is possible but we must override m1() method with "public" keyword. Then to execute m1() method from superclass we must call it by using "super.m1();"

Case #7: if method in the class is declared static is inheritance possible?
A) Inheritance is not possible, because we cannot override static method as non-static or vice versa.

Q) How can we access interface variable from subclass?
A) by using its name directly, because it is a static variable in interface.

For example:

```
interface I1{ int a = 10; }
interface I2{ int b = 20; }

class C implements I1, I2{
        public static void main(String[] args){
                System.out.println("a: "+a);
                System.out.println("b: "+b);
        }
}
```

Case #8: If both interfaces have a variable with same name, then how can you solve ambiguous error.
A) We must call that variable by using interface name as shown below

```
interface I1 {
        int x = 10;
        int y = 30;
}

interface I2{
        double x = 20;
}

class C implements I1, I2{
        public static void main(String[] args){
                //System.out.println(x);
                //System.out.println(C.x);
                System.out.println(I1.x);
                System.out.println(I2.x);

                System.out.println(y);
                System.out.println(C.y);
                System.out.println(I1.y);
        }
}
```

Polymorphism – having multiple forms / behaviors
Defining a method in multiple classes with the same name with different implementations for exhibiting different behaviors of the object is called polymorphism.

We can develop polymorphism by using
- Method Overriding *or*
- Method Overloading

To develop polymorphism we must define method in all subclasses with the same name with the same the prototype as it is declared in the superclass.

Types of Polymorphisms
Java Supports two types of polymorphisms
1. Compile-time polymorphism
2. Run-time polymorphism

Compile time Polymorphism or **Static binding** or **Early binding**
When a method is invoked, if its method definition which is bind at compilation time by compiler is only executed by JVM at runtime, then it is called compile-time polymorphism or static binding or early binding.

Static methods, Overloaded methods and non-static methods which are not overridden in subclass are come under compile time polymorphism.

Runtime Polymorphism or **Dynamic binding** or **Late binding**
When a method is invoked, the method definition which is bind at compilation time is not executed at runtime, instead if it is executed from the subclass based on the object stored in the referenced variable is called runtime polymorphism.
Only non-static overridden methods are come under run-time polymorphism. private non-static methods and default non-static methods from outside package are not overridden. So these method call comes under compile time polymorphism.

Definition of Runtime polymorphism should be given in interview
The process of executing an invoked method from different subclasses based on the object stored in the referenced variable is called runtime polymorphism.

To develop runtime polymorphism we must invoke the method by using super class referenced variable. Then only we can store all subclasses objects to execute that method from the targeted subclass.

So runtime polymorphism is only implemented through
- Upcasting - to store subclass objects
- Method overriding

If a *method* is called without *upcasting* or if it *is not overridden* then that polymorphism is always compile time polymorphism, because the method definition which is bound at compilation time is only executed at runtime.

//Below program explains compile time and runtime polymorphism.

```
class Example
{
        void m1()
        {
                SopIn("Example m1");
        }
        void m2()
        {
                SopIn("Example m2");
        }
}
```

```
class Sample extends Example
{
        void m1()
        {
                SopIn("Sample m1");
        }
        public static void main(String[] args)
        {
                Sample s = new Sample();
                s.m1();
                s.m2();

                Example e = new Sample();
                e.m1();
                e.m2();
        }
}
```

Abstraction

It is a process of defining a class by providing necessary details to call object operations by removing or hiding its implementation details for developing loosely coupled runtime polymorphic object user class.

So, Abstraction is a fundamental principle of modeling. It is oops supporting principle that make sure all three OOPS principles are working together to provide final shape to project. A system model is created at different levels, starting at the higher levels and adding more levels with more details as more is understood about the system. When complete, the model can be viewed at several levels. So abstraction is about:

- Looking only at the information that is relevant at the time
- Hiding details so as not to confuse the bigger picture

It means abstraction principle telling us provide only method prototypes and hide method implementation details. Because as a user of that method we only expecting the information to call that method, such as method signature, return type, modifier- nothing but method prototype.

Answer my question:

Have you ever think of the logic of System.out.println() method?

You only think of how to use this method for printing data on console, you never think the internal implementation of this method because you're the user of Java API. Hence SUN has not given implementation details of this method. If you are really interested to know this method details you can check it in source code (java.io.PrintStream.java).

So abstraction has two forms

- removing non-essential details and further
- hiding non-essential details

Removing non-essential details can be developed using **abstract methods**.
Hiding non-essential details can be developed using **concrete methods**.

Basically Abstraction provides a contract between a service provider and its clients.

Q) How can you restrict subclass not to override super class method?
A) By declaring method as final.

Q) How can you force subclass to override super class method?
A) By creating method as abstract method.

//Below project explains above all points by implementing abstraction, inheritance, polymorphism, and implementing run-time polymorphism and its advantages.

//Shape.java

```
public interface Shape{

    void area();
    void perimeter ();

}
/*
Shape is created as interface as we can't decide the logic of above two methods. For these two methods logic must be implemented only at specific sub class level.

Note:
If we write them as concrete methods with some default logic, as a super class we can't guarantee that these methods are overridden in subclasses.
Hence to ensure and force subclass developer to override these methods with respect to their business logic we must define them as abstract methods in super class.
*/
```

```
//Rectangle.java
public class Rectangle implements Shape{
        private double l;
        private double b;

        public Rectangle( double l , double b){
                this.l = l;
                this.b = b;
        }

        public void area(){
                System.out.println("Rectangle area:"+(l*b));
        }

        public void perimeter(){
                System.out.println("Rectangle perimeter:"+(2*(l+b)));
        }

        public void printLB()  {
                System.out.println("l:"+l);
                System.out.println("b:"+b);
        }
}

//Square.java
public class Square implements Shape{
        private double s;

        public Square(double s){
                this.s = s;
        }

        public void area(){
                System.out.println("Square area:"+(s*s));
        }

        public void perimeter(){
                System.out.println("Square perimeter: "+(4 * s));
        }

        public void printS(){
                System.out.println("s:"+s);
        }
}
```

```
//Circle.java
public class Circle implements Shape {
        public static final float PI = 3.14f;
        private double radius;

        public Circle(double radius){
                this.radius = radius;
        }

        public void area(){
                System.out.println("Circle area:"+(PI*radius*radius));
        }
        public void perimeter(){
                System.out.println("Circle perimeter: "+(2*PI*radius));
        }

        public void printRadius(){
                System.out.println("radius:"+radius);
        }
}

//RP.java
public class RP{
        public static void main(String[] args) {
                //normal objects execution
                Rectangle r = new Rectangle(10 , 20);
                r.area();
                r.perimeter();
                r.printLB();

                Square sq = new Square(10);
                sq.area();
                sq.perimeter();
                sq.printS();

                Circle c = new Circle(10);
                c.area();
                c.perimeter();
                c.printRadius();

                /* In the above code we have design problem that is, all above three objects are
                referenced objects so they are not eligible for garbage collection automatically.
                Hence we must write extra code to make them eligible for garbage collection
                like as below.
                */
```

```
        r = null;       sq = null;      c = null;

        /*
        By implementing RP or Upcasting we can avoid above three lines of code, not
        only this we avoid creation of the three reference variables.
        */

        //upcasting
        Shape s;

        s = new Rectangle(10, 20);
        s.area();
        s.perimeter();
        //s.printLB(); //CE: cannot find symbol

        /* Using super class reference variable we cannot access subclass specific
        members. We can only invoke members which are defined in Super class.
        In this case we must implement down casting, like as below */

        ( (Rectangle)s ).printLB();

        s = new Square(10);
        s.area();
        s.perimeter();
        ((Square)s).printS();

        s = new Circle(10);
        s.area();
        s.perimeter();
        ((Circle)s).printRadius();

        /* Still we have one more code design issue that is; in above lines of code the
        method invocation statements are repeated for all three objects.

        Solution: To avoid this code redundancy we must write a separate method with
        super class parameter type.

        This is the actual implantation of runtime- polymorphism, executing methods
        based on objects passed at runtime. */

        callAP(new Rectangle(10,20));
        callAP(new Square(10));
        callAP(new Circle(10));
}
```

```
    static void callAP(Shape s)
    {
        s.area();
        s.paremeter();

        // implement downcasting to invoke subclass specific members.

        ((Rectangle)s).printLB();
        ((Square)s).printS();
        ((Circle)s).printRadius();

        /*
        Q) Think a minute, is above downcasting code executed?
        Ohhh, yes I got it. It leads to ClassCastException. Rectangle object
        cannot be cast to Square.

        When we write a method with super class parameter type, we should not down
        cast the super class reference variable directly into subclass type because we
        cannot guarantee the coming subclass object at runtime.

        If user passes other subclass object then it leads to RE: CCE, because subclasses
        are not compatible with each other.

        To solve CCE always we must use "instanceof" operator (keyword) to verify
        object type like as below.

        Below is the valid code.
        */

        if (s instanceof Rectangle){
            ((Rectangle)s).printLB();
        }
        else if (s instanceof Square){
            ((Square)s).printS();
        }
        else if (s instanceof Circle){
            ((Circle)s).printRadius();
        }
    }
}
```

Conclusion

Run-time polymorphism not only provides automatic garbage collection it also allows you to change the behavior of a class without changing even a single line of code in a class. Hence we can say Runtime polymorphism based application provides **Pluggability** nature, means **loose coupling.**

What is meant by loose coupling and tight coupling?
If a product is functioning correctly even after changing one of its parts with another company given part, it is called loosely coupled way of manufacturing.

Else it is called tightly coupled way of manufacturing.

Let us see some real world products manufactured by one company those have loose coupled with another company products.

1. **Switch board**
   We can use same socket to plug and operate different electronic machines, also we can replace current switch with any other company given switch. Still switch board works fine as expected.

2. **Plank**
   To the same plank we can attach and use any company tube light.

3. **Computer**
   We can replace current monitor with any other company monitor.

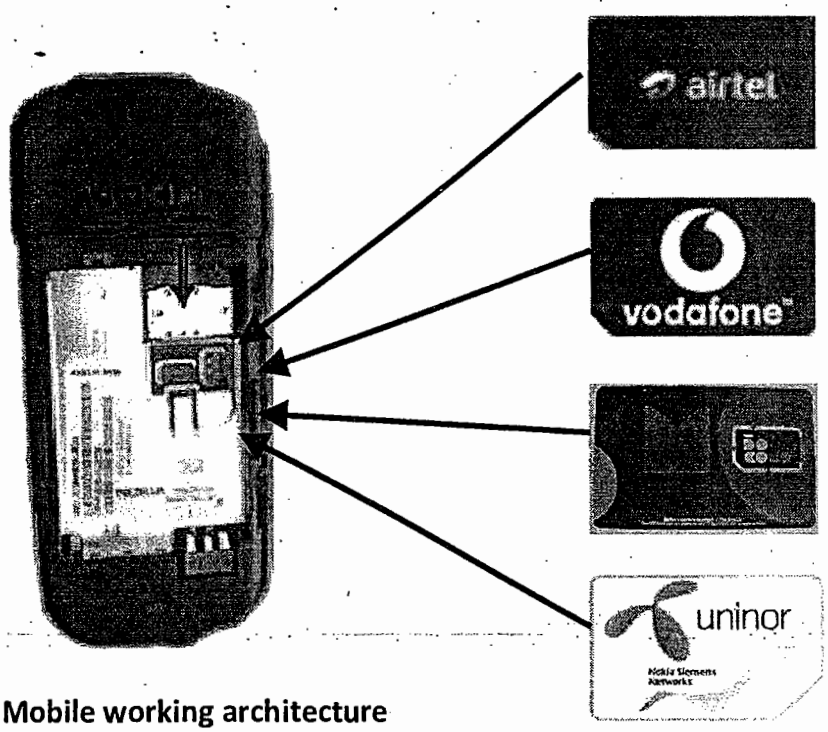
4. **Mobile**
   Mobile of one company can work with a SIM of all companies. As you know, these mobiles are called GSM mobiles, manufactured with loose coupling.

   Ex: **Nokia mobile** will work with Airtel, Vodafone, Docomo, even with the new company SIM Uninor.

   We also have mobiles those can only work with a particular type of SIM. What these mobiles are called, CDMA mobiles, those are manufactured with tight coupling.

   Ex: Tata, Reliance

GSM Mobile working architecture



CDMA Mobile working architecture

How different company manufactured products are working together?

In this example how mobile can work with different companies SIMs, even if SIM is changed mobile's original functionalities like - sendSMS, receiveSMS, dialCall, receiveCall - are not effecting. How it can automatically enables the current SIM manufacturer services?

It is possible because of **Specification,** a contract document.

When a super class must be defined for a set of same type of classes?

As you can noticed in the above program, runtime polymorphism achieved only with method overriding plus upcasting, hence we must define a super class for a set of same type of classes, for instance for all shape type classes like , Rectangle, Square, Circle etc we must define a super class like Shape.

Hence in below situations super class must be defied for a set of classes

- To have centralized change on a common data or logic that is being used in all related subclasses (Reusability).
- To have contract among or to force all subclasses to implement some sort of logic in a method with the same prototype (Forcibility).
    - Reusability is developed using concrete methods
    - Forcibility is developed using abstract methods.
- To store different objects of same type of subclasses at runtime using single reference variable for implementing runtime polymorphism.

When we must define subclass for a class?

Sub classes must be written only if

- We want to extend the functionality of the existed class
- To treat the new class as existed class type, for implementing runtime polymorphism.

Note: Don't implement subclasses unnecessarily, it cause memory location problem, because for every sub class object creation JVM creates memory of all super classes.

What are the benefits of inheritance?

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual super class. This also tends to result in a better organization of code and smaller, simpler compilation units.

Inheritance can also make application code more flexible to change because classes that inherit from a common super class can be used interchangeably. If the return type of a method is super class then the application can be adapted to return any class that is descended from that super class.

How can a user know what is the method should be called to get a service that is developed by another developer?

The developer should provide an interface with method prototype through which the user can execute method logic.

Here developer removing non-essential details relevant to user. Nothing but developer is supplying an interface with abstract methods.

In business terminology this interface is called contract document or specification, and method is called service.

So always projects development is starts with an interface to share services implementation method details to user. And further subclasses are developed by implementing those abstract methods defined in the interface according the company business requirement.

In projects Development we come across three types of persons

1. Service Specifier
2. Service Provider
3. Service User

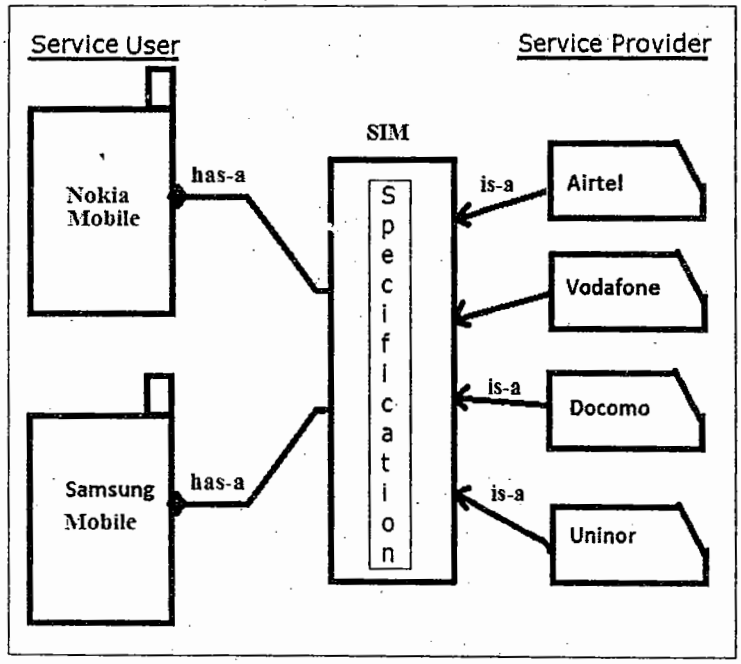
The person or company who develops specification is called service specifier.
The person or company who implements specification is called service provider.
The person or company who uses specification, further its implementation is called service user.

A specification can be implemented by more than one service provider, also can be used by more that on service user.

For example:
There are many SIM manufacturing companies providing implementation to SIM specification as per their business requirement. So SIM manufacturing companies are called service providers.

And there are many mobile manufacturing companies using SIM specification to supports all company SIMs working from the same mobile. So Mobile manufacturing companies are called service users.

Check below diagram



Q) What we achieve via abstraction in Java?
By implementing abstraction we can develop contract document between service providers, service users. This contract document is called specification, developed using interface in Java.

This specification is defined by a common company with all abstract methods.
Specification provides the below information to both service provider and service user.

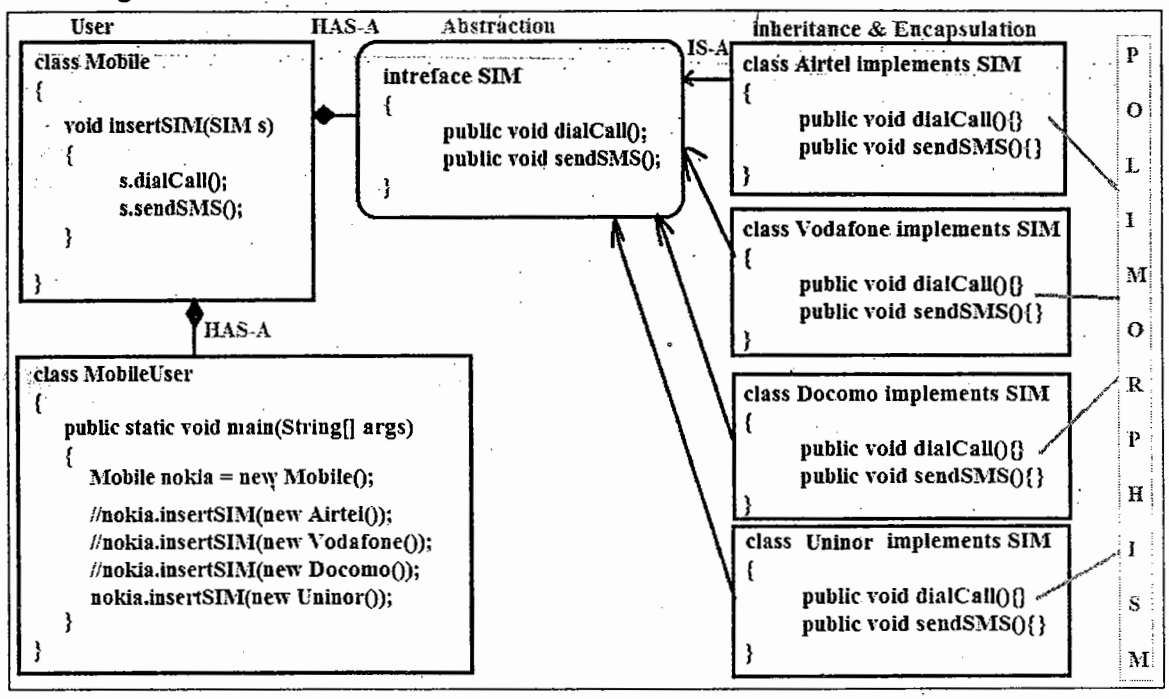
It tells to service provider – service user uses your services by calling this method. Hence Service provider will provide the service implementation logic with the method prototype given in specification.

It tells to service user - the required service is implemented with this method prototype by service provider. Hence Service user will invoke the service method with the method prototype given in specification.

Hence we can say, interface tells to

- ➔ service provider *what should implement but not how*.
- ➔ Service user *what to call* to get services

Below diagram shows the above information



In the above diagram SIM is a specification,
It provides information to Service provides – the service users, Mobile manufacturing companies, get services by calling methods dialCall and sendSMS methods.
It provides information to Service users – the service providers, SIM manufacturing companies, provides their service implementation in dialCall and sendSMS methods.

Then SIM manufacturing companies writes subclass from SIM interface and overrides dialCall and sendSMS methods as shown in Airtel, Vodafone, Docomo and Uninor classes.

And Mobile manufacturing companies will write a class by calling dialCall and sendSMS methods by using SIM reference variable as shown in Mobile class. These methods are called from the SIM that is inserted in the mobile as shown in MobileUser class.

Since Service provider, SIM manufacturing companies, and service users, Mobile manufacturing companies are following same SIM specification, Mobile users – means customers- will not face any problems in using different company SIMs with same mobile.

Hence if products of different companies want to work together, that products design and implementation must starts with specification, nothing but interface.

So, we can say abstraction provides a contract between a service provider and users.

Check below project development that shows how Mobile and SIM works together.
It also shows a sample project implementation, how in software industry real-time projects are developed by combining all these OOPS principles.

Below Program shows actual implementation with reflection API to implement runtime polymorphism.

//SIM.java
```
public interface SIM{
        public String sendSMS(String msg, long mobilenumber);
        public String dialCall(long mobilenumber);
}
```

//Airtel.java
```
public class Airtel implements SIM{
        public String sendSMS(String msg, long mobilenumber)      {
                return "Airtel : Your SMS Send successfully";
        }

        public String dialCall(long mobilenumber)    {
                return "Airtel: The number you are dialing is busy, please dial after some time";
        }
}
```

//Vodafone.java
```
public class Vodafone implements SIM {
        public String sendSMS(String msg, long mobilenumber)      {
                return "Vodafone : Your SMS Send successfully";
        }

        public String dialCall(long mobilenumber)    {
                return "Vodafone: The number you are dialing is not reachable,
                                                        please dial after some time";
        }
}
```

```
//Docomo.java
public class Docomo implements SIM {
        public String sendSMS(String msg, long mobilenumber)      {
                return "Docomo : Your SMS Send successfully";
        }

        public String dialCall(long mobilenumber)    {
                return "Docomo: The number you are dialing is switched off,
                                                        please dial after some time";
        }
}

//Mobile.java
public class Mobile {
        private SIM sim;

        public void insertSIM(String simName) throws Exception{
                //reflection api
                Class       simclass       = Class.forName(simName);
                Object    simobject    = simclass.newInstance();

                if (simobject instanceof SIM) {
                        sim      = (SIM) simobject;
                }
                else{
                        throw new Exception(" Invalid SIM ");
                }
        }

        public String sendSMS(String msg, long mobilenumber){
                return sim.sendSMS(msg, mobilenumber);
        }

        public String dialCall(long mobilenumber){
                return sim.dialCall(mobilenumber);
        }
}
```

```
//MobileUser.java => keypad logic
import java.io.*;
public class MobileUser {
	public static void main(String[] args) throws Exception{
		Mobile iphone= new Mobile();

		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

		System.out.print("Insert SIM: ");
		String simName	= br.readLine();

		iphone.insertSIM(simName);

		System.out.println();

		System.out.println("Type 1 and press <Enter> key  to Send SMS ");
		System.out.println("Type 2 and press <Enter> key  to Make Call\n ");

		System.out.print("Enter option:");
		String option   = br.readLine();

		String resp ;

		if(option.contains("1")){
			System.out.println("Type message and press <Enter> key:");
			String msg = br.readLine();

			System.out.println("Type mobile number and press <Enter> key:");
			String mobilenumber = br.readLine();

			resp = micromax.sendSMS(msg, Long.parseLong(mobilenumber));
			System.out.println(resp);
		}
		else if(option.contains("2")){
			System.out.println("Type mobile number and press <Enter> key:");
			String mobilenumber = br.readLine();

			resp = micromax.dialCall(Long.parseLong(mobilenumber));
			System.out.println(resp);
		}
		else{
			System.out.println("Invalid Option");
		}
	}
}
```

Assume Mobile is manufactured, release into market, user bought it, now new SIM Uninor is released into market with attractive services. Can your Mobile work with this new SIM?

It will work there will not be any problem with Mobile because its software is developed by following SIM specification, hence it supports loose coupling and Runtime polymorphism.

It's now Uninor SIM manufacturer responsibility to develop sim by flowing SIM specification to ensure its working with all GSM based mobiles. If it is developed as per SIM specification Mobile will accept Uninor also.

Now Practically think, insertSIM() method we are checking the inserted SIM class is of type SIM or not, so Uninor should be a subclass of SIM to be used with Mobile.

Below is the **valid Uninor SIM** implementation

```
//Uninor.java
public class Uninor implements SIM
{
	public String sendSMS(String msg, long mobilenumber)
	{
		return "Uninor: Your SMS Send successfully";
	}

	public String dialCall(long mobilenumber)
	{
		return "Uninor: Your calling is forwading to voice mail";
	}
}
```

So, ultimately who will suffer if specification is not followed in developing services?
Service Provider, the product will not have sales in the market.

Why projects or products development is starts with interface?
To develop loosely coupled based applications runtime polymorphism applications.

Loosely coupled or runtime polymorphism based application means it allows to replace an object with another object of same category without recompiling and it exhibits the same expected functionality.

For example GSM mobile will work any company SIM, and exhibits same way of functioning with all company SIMs.

Tight coupled application means if we change object the application will not work. For example CDMA mobile, it will not work with other SIMs.

How is it possible to plug different objects of same category to an application at runtime?
With specification, means with abstraction methods, the same abstract methods are implemented in all objects. And in the application also the same abstract methods are called. Hence this application can plug with any of the objects which is implementing the same specification.

Conclusion:
Both people service provides (objects developers) and service users (application developers) will follow specification (interface) to plug each other.

Assignment
Design a project to represent Vehicle type objects in Java.
From this project you will understand when to use

- Interface
- Abstract class
- Concrete class
- Final Class
- Abstract method
- Concrete method
- Final method

Q) What is the right design to add more services or method to an existed business operations specification (interface)?

We should define new interface deriving from the existed interface. In this new interface we must declare all new service's methods. By following this new interface service providers and service users must develop new subclasses and user classes by deriving them from old interface's service provider and service user classes as shown below.

```
interface SIM3G{
 void vedioCall();
}
class Airtel3G extends Airtel implements SIM3G{
        public void vedioCall(){
                System.out.println("Airtel3G: vedioCall");
        }
}
class Mobile3G extends Mobile{
        void intersertSIM(SIM3G sim){
                sim.dialCall();
                sim.sendSMS();
                sim.vedioCall();
        }
}
```

Chapter 19

Types of objects & Garbage Collection

- In this chapter, You will learn
  - Definition of referenced variable
  - Types of referenced variable
  - Two different types of objects
  - Accessing object members from two types of objects
  - Object creation and destruction process
  - Different ways to unreferencing objects
  - *java.lang.OutOfMemoryError* (OOME)
  - Need of Garbage collection
  - GC Thread responsibilities
  - Requesting JVM to start GC
  - Need of finalize method

- By the end of this chapter- you can find out available and unavailable objects and objects those are eligible for GC.

Interview Questions

By the end of this chapter you answer all below interview questions

Referenced varaibles and types of referenced varaibles

- Definition of reference variable.
- Difference between primitive and referenced variables
- What are the values can be stored in a referenced variable?
- Types of reference variables?
    - Local
    - Static
    - Non-Static
    - Final
    - Volatile
    - Transient
- Where referenced variables provided memory location by whom and when?
- Write a program to show the creation of all three types of reference variables with JVM architecture.

Types of objects

- Two different types of objects
- When an object is called referenced and unreferenced object?
- How can we access members from referenced and unreferenced objects?
- Project Scenarios forced to create unreferenced and referenced objects.
- How many referenced variables can an object has pointing to it?
- How can we convert referenced object as unreferenced object?

Garbage collection

- What is the meaning of garbage, and garbage collection?
- Need of Garbage collection
- *java.lang.OutOfMemoryError*
- How JVM can destroy unreferenced objects?
- GC Thread responsibilities
- What type of thread is Garbage collector thread?
- Is garbage collector a daemon thread?
- Garbage Collector is controlled by whom?
- Can the Garbage Collection be forced by any means?
- How can the Garbage Collection be requested?
- What is the algorithm JVM internally uses for destroying objects?
- Which part of the memory is involved in Garbage Collection? Stack or Heap?
- Need of finalize method
- When does an object become eligible for garbage collection?

- What are the different ways to make an object eligible for Garbage Collection when it is no longer needed?
- What is the purpose of overriding finalize() method?
- Can we call finalize method?
- Can an unreachable Java object become reachable again?
- How many times does the garbage collector calls the finalize() method for an object?
- If an object becomes eligible for Garbage Collection and its finalize() method has been called and inside this method the object becomes accessible by a live thread of execution and is not garbage collected. Later at some point the same object becomes eligible for Garbage collection, will the finalize() method be called again?
- What happens if an uncaught exception is thrown from during the execution of the finalize() method of an object?
- How to enable/disable call of finalize() method of exit of the application

Definition of referenced varaible

The variables created by using referenced datatypes are called referenced variables.
Referenced variables are created by using array, class, interface or enum.

For Example

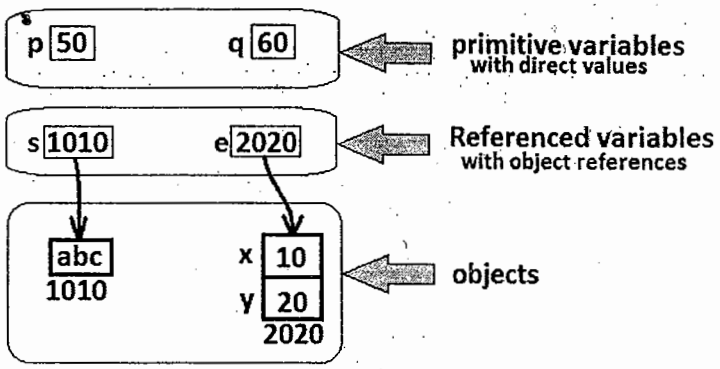
```
int[]       ia;
Example     e;
String      s;
```

Q) What is the difference between referenced variable and primitive variable?

A) primitive variables stores value directly, but referenced variables stores reference of the object.

For Example

```
//primitive variables
int     p   = 50;
int     q   = 60;

//referenced variables
String  s   = "abc";
Example e   = new Example();
```



Q) Why the variable created by using referenced types is called referenced variable?

A) Because it points or referenced to object's memory.

Q) What are the values allowed to store in a referenced variable?

Below are the two possible values

1. Default value *null* -> it can be stored in all types of referenced variables
2. Object reference -> the object should be same type of referenced variable.

For Example

```
//storing object reference
String      str     = "abc";        ✓
Example     e1      = new Example(); ✓
Example     e2      = "abc";        ✗ CE: incompatible types
                                       found: String
                                       required: Example
```

```
//storing null
String      str     = null;         ✓
Example     e       = null;         ✓
```

Types of referenced variables

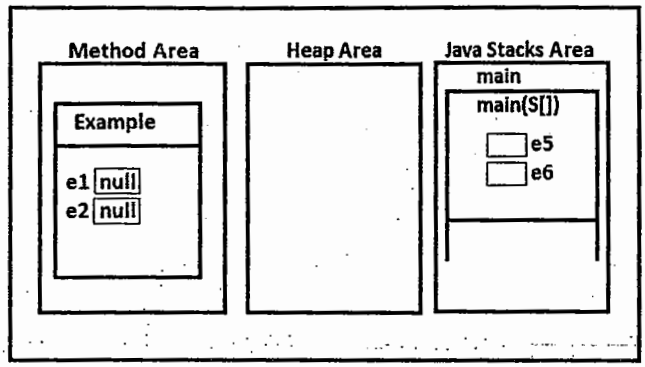
Like primitive variables, referenced variables are also divided in three main categories

1. Local referenced variable
2. Static referenced variable
3. Non-static referenced variable

Below program shows creating all three types of variables

```
class Example
{
        //static referenced variables
        static Example e1;
        static Example e2;

        //non-static referenced variables
        Example e3;
        Example e4;

        public static void main(String[] args)
        {
                //local referenced variables
                Example e5;
                Example e6;
        }
}
```

JVM Architecture with referenced variables



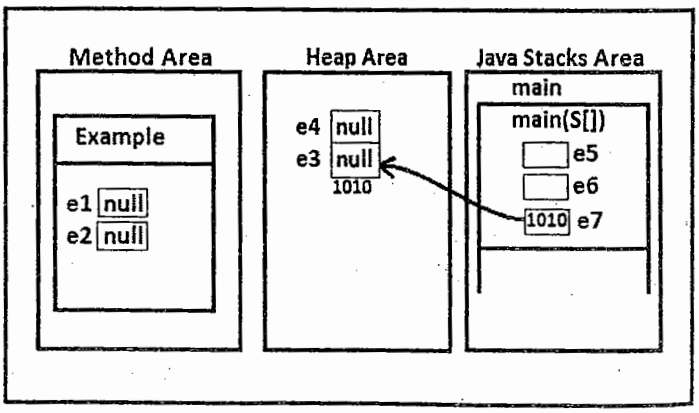
e1, e2 variables are created in method area with default value *null*, as they are static variables.

e3, e4 referenced variables are ***not*** created as they are non-static referenced variables. These two variables will be created in *heap area* if we create object of Example class.

e5, e6 referenced variables are created in main method stack frame as they are local referenced variables.

```
class Example{
        static Example e1, e2;

        Example e3, e4;

        public static void main(String[] args) {

                Example e5, e6;

                //object creation with local
                //referenced variable

                Example e7 = new Example();

        }
```

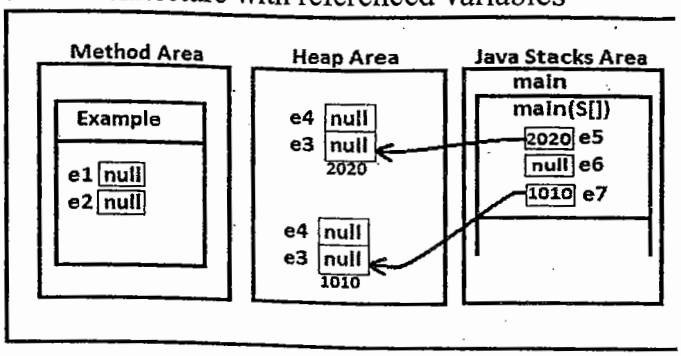
JVM Architecture with referenced variables



//Accessing referenced variables

```
class Example{
        static Example e1, e2;

        Example e3, e4;

        public static void main(String[] args) {

                Example e5, e6;

                //object creation with local
                //referenced variable

                Example e7 = new Example();

                System.out.println(e1);
                System.out.println(e2);

                System.out.println(e3);
                System.out.println(e4);

                System.out.println(e7.e3);
                System.out.println(e7.e4);

                System.out.println(e5);
                System.out.println(e6);

                //converting local null
                //referenced variables as
                // object referenced variable
                e5 = new Example();
                e6 = null;

                System.out.println(e5);
                System.out.println(e6);
        }
}
```

JVM Architecture with referenced variables



Q) What is the output will be printed when we print object?

A) To print object information, println() and print() methods internally calls ***toString()*** on the current object. This method is originally defined in ***java.lang.Object*** class to return current passed object's

"***classname@hashcode in hexadecimal form***"

If we print number referenced variable, these methods will not call *toString()* method.

Simply these methods prints *null.*

Where referenced variables are provided memory location?

Referenced variables get memory in one of the three runtime areas based on their type of declaration

- *If they are declared as static*, they will get memory in MA in individually memory locations at time of class loading
- *If they are declared as non-static*, they will get memory in HA in continuous memory locations as part of other object.
- *If they are declared as local*, they will get memory in JSA in individual memory locations in its method stack frame.

Note:

objects created using any of the above three referenced variables are always created in HA.

object life-time and scope, non-static referenced variables life-time and scope

- Non-static variables are created when object is created, and are destroyed when object is destroyed.
- object is destroyed when its referenced variables is destroyed or initialized with another object's reference or null.
- So we can conclude, the scope of the object is the scope of its referenced variable.

What is the output from the below program?

```
class Example{
        //static referenced variables
        static Example e1;
        static Example e2;

        //non-static referenced variables
        Example e3;
        Example e4;
        int x =10, y = 20;

        public static void main(String[] args){
                //local referenced variables
                Example e5;
                Example e6;

                //If we execute this program with above lines code only static and local
                //referenced variables are created. To create non-static referenced variables are
                //must create object this class.

                Example e7 = new Example();
```

```
//printing variables
System.out.println(e1);//null
System.out.println(e2);//null

//System.out.println(e3);
        //CE: non-static variable e3 cannot be referenced from static context
//System.out.println(e4);
        //CE: non-static variable e4 cannot be referenced from static context

//System.out.println(e5); CE: variable e5 might not have been initialized
//System.out.println(e6); CE: variable e5 might not have been initialized

System.out.println(e7);//Example@1234
/*
- If we print null referenced variable JVM prints null
- If we print object referenced variable JVM prints its <classname>@hashcode.

- We can also call toString explicitly on object referenced variable,
- But it will not be called on null referenced variable, because it leads to
        NullPointerException.

as shown below
*/
System.out.println(e7); //Example@1234567
System.out.println(e7.toString()); //Example@1234567

//what is output from below statements
System.out.println(e1);//null
//System.out.println(e1.toString()); RE: NPE

//converting null referenced variables as object referenced variables.
e1 = new Example();
e1.e3 = new Example();
e1.e3.e4 = new Example();

//printing all bove variables with different combination
System.out.println(e1); //Example@12345
System.out.println(e1.e3); //Example@4567
System.out.println(e1.e3.e4); //Example@7986
System.out.println(e1.e3.e4.e4); //null
System.out.println(e1.e3.e4.e3); //null

System.out.println(e1.e3.e3.e3); //RE: NPE
System.out.println(e1.e3.e3.e1); //Example@12345
System.out.println(e1.e3.e3.e2); //null
```

```
        System.out.println(e1.e3.e3.e1.x); //10
        System.out.println(e1.e3.e3.e2.x); //RE: NPE

        System.out.println(e7.e1); //Example@12345
        System.out.println(e1.e1); //Example@12345

        System.out.println(e1.e7); //CE: c f s
        System.out.println(e7); //Example@1234567

        /*
        - A local referenced variable can access static and non-non-static
        referenced variables, but not local referenced variables

        - A static and non-static referenced variables can access other and same (itself)
        static and non-static referenced variables but not local referenced variables.

        //after executing below statement how many objects are eligible for
        //garbage collection
        */
        e1 = null;

        //A) 3 objects, e1 object and all its internal objects are eligible for
        //garbage collection.

        //If base object is eligible for garbage collection, all its internal objects are also
                //eligible for gc.
    }
}
```

Types of objects

In Java we have two types of objects based on the referenced variables

1. Referenced or Reachable object
2. Un-referenced or Un-reachable object

If an object is pointed by at least one referenced variable, it is called referenced or reachable object, else it is called unreferenced or unreachable object.

For Example

```
//referenced object creation
Example e = new Example();

//unreferenced object creation
new Example();
```

How can we access members from referenced and unreferenced objects?

After the referenced object creation, we can access it's all non-static members using its referenced variable as many times as we need.

But we cannot access non-static members from unreferenced object after its creation statement, because its reference is not stored. There is a way to access non-static members from unreferenced object, we must call non-static member in its creation statement itself using "." operator as shown in the below program

```
class Example
{
        int x = 10;
        int y = 20;

        void m1()
        {
                System.out.println("m1");
        }
}
```

```
class Test{
        public static void main(String[] args) {
                //referenced object creation
                Example e = new Example();
                //accessing members from this object
                System.out.println(e.x + " ... " + e.y);
                e.m1();

                //creating unreferenced object
                new Example();
    //We cannot access members from the above object
    //We can access non-static members from
    //unreferenced object as shown below
                new Example().m1();
        }
}
```

Project Scenarios forced to create unreferenced and referenced objects.

Case #1:

If we want to access all members of an object or single member more than once, we must create that object as referenced object.

If we want to access only one member that too only one time from an object, it is recommended to create that object as unreferenced object. So that memory will be saved as we do not create referenced variable. Below program shows above points

```
class Example {
        int x = 10;
        int y = 20;

        void m1(){
            System.out.println("m1");
        }
}
```

```
class Test{
        public static void main(String[] args) {
                Example e = new Example();
                System.out.println("x: "+e.x);
                System.out.println("y: "+e.y);
                e.m1();

                System.out.println("x: "+new Example().x);
        }
}
```

Case #2:

After a method execution if we want to use current object or argument object, we must create them as referenced object else it is recommended to create them as unreferenced objects.

If we call a method with unreferenced object it is referenced while executing method, because they are referenced by either "*this*" or "*method parameter*". After that method execution, they become unreferenced hence we cannot access that object members after that method call statement. *This is the main reason we create unreferenced objects in projects*

The below program explains above point

```
class Example {

        int x = 10;
        int y = 20;

        void m1(Example e)
        {
                e.x = e.x + 1;
                e.y = e.y + 2;
        }
}
```

```
class Test{
        public static void main(String[] args) {
                Example e1 = new Example();
                Example e2 = new Example();
                e1.m1(e2);

        System.out.println(e1.x + " ... "+ e1.y);//10 ... 20
        System.out.println(e2.x + " ... "+ e2.y);//11 .. 21

                e1.m1(new Example());
        System.out.println(e1.x + " ... "+ e1.y);//10 ... 20

                new Example().m1(new Example());
        }
}
```

Object creation and destruction process

In Java,

- Developer is responsible to create object using new keyword and available constructor.
- JVM is responsible to destroy those objects.

Developer is free from destroying objects, means clearing memory location.

Q) What type of objects does JVM destroy?

JVM destroys only unreferenced objects, it will not destroy referenced objects.

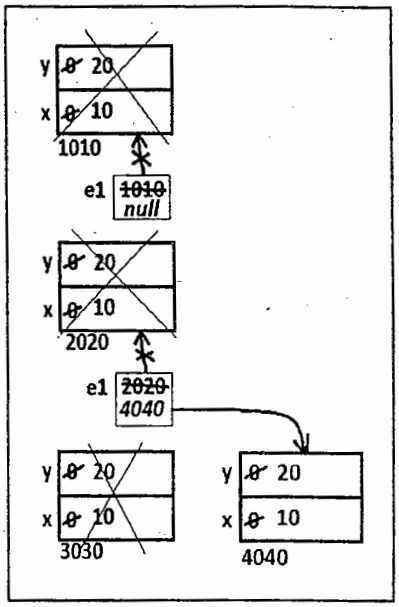
Q) How can we convert referenced objects to unreferenced?

We have three ways to convert referenced object to unreferenced

1. Storing ***null*** to all its referenced variables
2. Strong ***another object reference*** to all its referenced variables
3. Creating ***Islands of Isolations***

Check below programs

```
class Example{

        int x = 10;
        int y = 20;

        public static void main(String[] args) {

          //creating referenced object
          Example e1 = new Example();
          Example e2 = new Example();

          //creating unreferenced object
          new Example();

          //unreferencing first object by storing null
          e1 = null;

         //unreferencing second object by assigning another object

          e2 = new Example();

          Q) At this line number how many objects are unreferenced?
          A) 3 objects, 1st, 2nd and 3rd.

          Q) How many objects are still referenced?
          A) only one object, that is crated at last.

    }
}
```



For easy counting, *strike of* object's memory immediately after all its references are destroyed.

java.lang.OutOfMemoryError:

"If there is no sufficient memory in heap area to create new object", JVM terminates program execution by throwing this exception.

Solution: Developer is responsible to unreferencing the object after its use.

JVM has internal mechanism to destroy all unreferenced or unused objects when there is not sufficient memory to create new memory.

Find out how many objects are unreferenced in the below program?

```
class Example
{
        int x = 10;
        int y = 20;

        void m1(Example e)
        {
                e = null;

                Example e1 = new Example();
        }

        public static void main(String[] args)
        {
                Example e1 = new Example();
                Example e2 = new Example();

                e1.m1(e2);
                Q) At this line number how many objects are unreferenced?

                new Example();

                e1.m1(new Example());
                Q) At this line number how many objects are unreferenced?

                e1 = e2;
                Q) At this line number how many objects are unreferenced?

                e2 = null;

                Q) At this line number how many objects are totally unreferenced?
                A) 5 objects are unreferenced.
        }
}
```

Garbage Collection

The process of destroying unreferenced objects is called Garbage Collection.
After object is unreferenced it is considered as unused object, hence JVM automatically destroys that object. In Java, developer's responsibility is only creating objects and unreferencing those objects after their usage. So that JVM takes care of destroying those objects. In Java, we do not have destructors like in C++.

java.lang.OutOfMemroyError

If there is no space in heap area to create new objects, JVM terminates program execution by throwing exception "java.lang.OutOfMemoryException".

To utilize memory effectively developer must unreference objects after their usage.

How JVM can destroy unreferenced object?

JVM internally uses a daemon thread called "garbage collector" to destroy all unreferenced objects. A daemon thread is a service thread. Garbage collector is called daemon thread because it provides services to JVM to destroy unreferenced objects.

This thread is a low priority thread. Since it is a low priority thread we cannot guarantee this thread execution.

So, can you guarantee objects destruction?

No, we cannot guarantee objects destruction even though it is unreferenced, because we cannot guarantee garbage collector execution.

So, we can only confirm whether object is eligible for garbage collection or not.

Can we force garbage collector?

No, we cannot force garbage collector to destroy objects, but we can request it.

How can we *request* JVM to start garbage collection process?

We have a method called "***gc()***" in *System* class as *static method* and also in *Runtime* class as *non-static method* to request JVM to start garbage collector execution.

So, we can call it in our program as below
"*System.gc()*" or "*Runtime.getRuntime().gc()*"

Q) In the previous application how many objects are eligible for garbage collection?

A) 5 objects. There are five unreferenced objects.

GC with IS-A relation

```
class A {
        int x = 10;
}
```

```
class B extends A{
        int y = 20;
};

B b1 = new B();
B b2 = new B();

b1 = b2;
```

How many objects are created, and how many objects are eligible for GC?
A) 2 objects, 1 object

GC with HAS-A relation

```
class Student{
        Address add = new Address();
};

class Address{
};

Student s1 = new Student();
Student s2 = new Student();

s1 = null;
s2 = new Student();
```

How many objects are created, and how many objects are eligible for GC?
6, 4

Q) What is the algorithm JVM internally uses for destroying objects?
A) "mark and sweep" is the algorithm JVM internally uses.

Note: *For more details on Garbage Collection check "java.lang.Object" chapter.*

Garbage Collections Interview Questions

Q1) Which part of the memory is involved in Garbage Collection? Stack or Heap?
Ans) Heap

Q2)What is responsibility of Garbage Collector?
Ans) Garbage collector frees the memory occupied by the unreachable objects during the java program by deleting these unreachable objects.
It ensures that the available memory will be used efficiently, but does not guarantee that there will be sufficient memory for the program to run.

Q3) Is garbage collector a daemon thread?
Ans) Yes GC is a daemon thread. A daemon thread runs behind the application. It is started by JVM. The thread stops when all non-daemon threads stop.

Q4)Garbage Collector is controlled by whom?
Ans) The JVM controls the Garbage Collector; it decides when to run the Garbage Collector. JVM runs the Garbage Collector when it realizes that the memory is running low, but this behavior of jvm can not be guaranteed.
One can request the Garbage Collection to happen from within the java program but there is no guarantee that this request will be taken care of by jvm.

Q5) When does an object become eligible for garbage collection?
Ans) An object becomes eligible for Garbage Collection when no live thread can access it.

Q6) What are the different ways to make an object eligible for Garbage Collection when it is no longer needed?
Ans)
1. Set all available object references to null once the purpose of creating the object is served :

```
public class GarbageCollnTest1 {
  public static void main (String [] args){
        Student s1 = new Student();
        //Student object referenced by variable s1 is not eligible for GC yet

        S1 = null;
        /*Student object referenced by variable s1 becomes eligible for GC */
  }
}
```

2. Make the reference variable to refer to another object : Decouple the reference variable from the object and set it refer to another object, so the object which it was referring to before reassigning is eligible for Garbage Collection.

```
publc class GarbageCollnTest2 {

  public static void main(String [] args){
        Student s1 = new Student();
        Student s2 = new Student();
        //Student object referred by s11 is not eligible for GC yet

        s1 = s2;
        /* Now the s1 variable referes to the second Student object and the first Student object
        is not referred by any variable and hence is eligible for GC */

  }
}
```

3) Creating Islands of Isolation : If you have two instance reference variables which are referring to the instances of the same class, and these two reference variables refer to each other and the objects referred by these reference variables do not have any other valid reference then these two objects are said to form an Island of Isolation and are eligible for Garbage Collection.

```
public class GCTest3 {
        GCTest3 g;

        public static void main(String [] str){
                GCTest3 gc1 = new GCTest3();
                GCTest3 gc2 = new GCTest3();
                gc1.g = gc2; //gc1 refers to gc2
                gc2.g = gc1; //gc2 refers to gc1
                gc1 = null;
                gc2 = null;
                //gc1 and gc2 refer to each other and have no other valid //references
                //gc1 and gc2 form Island of Isolation
                //gc1 and gc2 are eligible for Garbage collection here
          }
}
```

Q7) Can the Garbage Collection be forced by any means?
Ans) No. The Garbage Collection can not be forced, though there are few ways by which it can be requested there is no guarantee that these requests will be taken care of by JVM.

Q8) How can the Garbage Collection be requested?
Ans) There are two ways in which we can request the jvm to execute the Garbage Collection.

1)The methods to perform the garbage collections are present in the Runtime class provided by java. The Runtime class is a Singleton for each java main program.
The method getRuntime() returns a singleton instance of the Runtime class. The method gc() can be invoked using this instance of Runtime to request the garbage collection.
2)Call the System class System.gc() method which will request the jvm to perform GC.

Q9) What is the purpose of overriding finalize() method?
Ans) The finalize() method should be overridden for an object to include the clean up code or to dispose of the system resources that should to be done before the object is garbage collected.

Q10) If an object becomes eligible for Garbage Collection and its finalize() method has been called and inside this method the object becomes accessible by a live thread of execution and is not garbage collected. Later at some point the same object becomes eligible for Garbage collection, will the finalize() method be called again?
Ans) No

Q11) How many times does the garbage collector calls the finalize() method for an object?
Ans) Only once.

Q12) What happens if an uncaught exception is thrown from during the execution of the finalize() method of an object?
Ans) The exception will be ignored and the garbage collection (finalization) of that object terminates.

Q13) What are different ways to call garbage collector?
Ans) Garbage collection can be invoked using System.gc() or Runtime.getRuntime().gc().

Q14) How to enable/disable call of finalize() method of exit of the application
Ans) Runtime.getRuntime().runFinalizersOnExit(boolean value) . Passing the boolean value will either disable or enable the finalize() call.

Q) Can an unreachable Java object become reachable again?
Yes. It can happen when the Java object's finalize() method is invoked and the Java object performs an operation which causes it to become accessible to reachable objects.

Chapter 20

Arrays

- In this chapter, You will learn
  - Definition and Need of Array
  - Limitations and Advantages of Array
  - Three kinds of array creation syntaxes
  - Array memory structure
  - Array creation rules
  - Common *exceptions* raised in array creation and modifications
  - *length* property
  - Array Casting
  - Garbage collection with Arrays
  - Declaring array as final
  - Need of Anonymous array.
  - Array creation with different dimensions
  - Variable-argument method
- By the end of this chapter- you can feel more comfortable programming with Arrays and variable-argument.

Interview Questions

By the end of this chapter you answer all below interview questions

1. Array definition.
2. Need of array
3. Data type of array
4. Syntaxes to create an array and its rules
5. Array limitation
6. Finding length of an array
7. Storing and retrieving array elements and rules
8. Difference in storing primitive and reference type of data in arrays.
9. Array casting and exception in array casting
10. Passing array as method argument and return type
11. Anonymous arrays
12. Garbage Collection with arrays
13. Declaring array as final
14. Three types of array referenced variables and their memory locations
15. Types of Arrays based on dimension
    a. Single dimensional arrays
    b. Multi dimensional arrays
16. Purpose of the multidimensional arrays
17. Jogged Array
18. Var-arg parameter method, and rules on var-arg parameter

Array

Definition of Array

Array is a referenced data type used to create fixed number of multiple variables of same type to store multiple values of similar type in continuous memory locations with single variable name.

Need of array

In projects array is used to collect / group similar type of values or objects to send all those multiple values with single call from one method to another method either as an argument or as a return value.

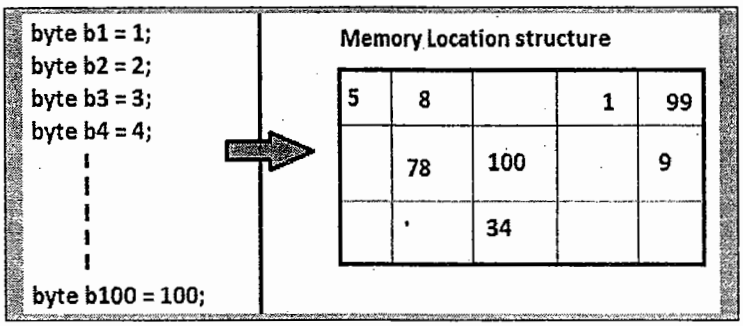
Let us first understand the problem of variables

Using variable we cannot store multiple values in continuous memory locations.

Duet to this limitation we have below two problems.

Problem #1

For instance if we want to store multiple values, say 1 to 100, we must create 100 variables. All those 100 variables are created at different locations as show below. Hence it takes more time to retrieve values.



Problem #2

Also using primitive or class type variables we cannot pass multiple values and objects to a method as argument and also we cannot return multiple values and objects from a method at a time. If we use these types of variables we must define multiple overloaded methods with required number of parameters.

Below diagram shows the above limitation.

```
void m1( int a ){}              <= we can only pass one int value
void m1( int a, int b ){}       <= we can only pass two int values

void m1( Example e ){}                  <= we can only pass one Example class object
void m1( Example e1, Example e2 ){}     <= we can only pass two Example class objects

int m1(){  return 50;  }                        <= we can only return one int value
Example m1() { return new Example(); }          <= we can only return one Example class object
```

Solution

To solve above two problems, we must group all values and objects to send them as a single unit from one application to another application as method argument or return type. To group them as a single unit we must store them in continuous Memory Locations. This can be possible by using referenced datatypes array.

In Java, Array is a reference data type. It is used to store fixed number of multiple values and objects *of same type* in continuous Memory locations.

Note: Like other data types Array is not a keyword rather it is a concept. It creates continuous memory locations using other primitive or reference types.

Array limitation

Its size is fixed, means we cannot increase or decrease its size after its creation.

Array declaration syntax:

<Accessibility modifier> <Non-Accessibility Modifier> <datatype>[] <array variable name>;

For Example:

```
public static int[] i;
public static Example[] e;
```

Note: we can place []

- after data type
- before variable name
- after variable name
- But not before data type

Rule: Like in C or C++, in Java we cannot mention array size in declaration part. It leads CE.

For Example:

```
int[5] i;   CE: illegal start of expression
int[] i;
```

Find out valid array declaration statements

1. int[] i;
2. int []i;
3. int [] i;
4. int []i;
5. int i[];
6. int[5] i;
7. []int i;

Below syntax shows multidimensional array declaration

```
int[][] i;          <---- two dimensional
int[][][] i;        <---- three dimensional
```

Find out valid multidimensional array declarations

1. int[][] i;
2. int[] []i;
3. int[] i[];
4. int [][]i;
5. int []i[];
6. int i[][];

Find out variables type from the below list

1. int i, j; <---- both i, j are of type int
2. int[] i, j; <---- both i, j are of type int[]
3. int i[], j; <---- i is of type int[], and j is of type int
4. int []i1, i2; <---- both i, j are of type int[]
   if we place [] before variable that is applicable to data type not to variable. In this case Compiler moves [] to after datatype.
5. int[][] i, j; <---- both i, j are of type int[][]
6. int[] i[], j; <---- i is of type int[][], and j is of type int[]
7. int[] i[][], j; <---- i is of type int[][][], and j is of type int[]

Rule: [] is allowed before the variable only for first variable that is placed immediately after data type.

```
Ex:   int   []p, q[];
      int   []p, []q;
```

Array object creation

We have two ways to create array object

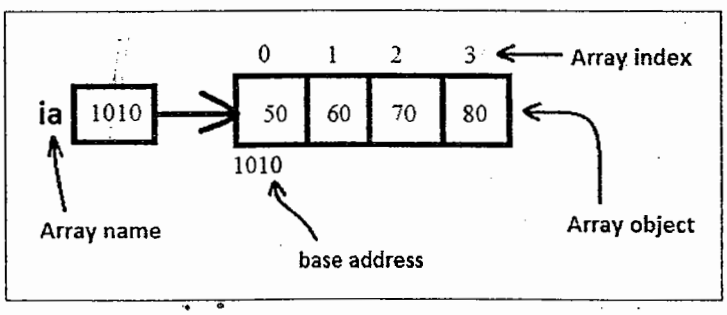
Syntax #1: Array creation with values

<Accessibility Modifier> <Modifier> <data type>[] <array name> = {<list of values with , separator>};

Example #1: Array creation with primitive data type

int[] ia = {50, 60, 70, 80};

In this array object creation, array contains 4 continuous memory locations with some starting base address assume 1010, and that address is stored in "ia" variable as shown in the below diagram.



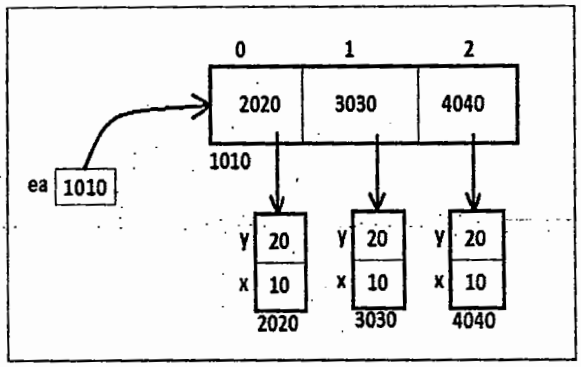
As you noticed, every array location has index starts with ZERO. This array index is used for storing, reading, and modifying array values.

Example #2: Array creation with referenced data type.

```
class Example {
        int x = 10;
        int y = 20;
}
```

Example[] ea = {new Example(), new Example(), new Example()};

In this array object creation, array contains 3 continuous memory locations with some starting base address assume 1010, and that address is stored in "ea" variable as shown in the below diagram.



What is the difference in creating array object with primitive types and referenced types?
As shown in the above diagrams

- If we create array object with primitive type, all its memory locations are of primitive type variables, so values are stored directly in those locations.

- If we create array object with referenced type, all its memory locations are of referenced type variables, so object reference is stored in those locations.

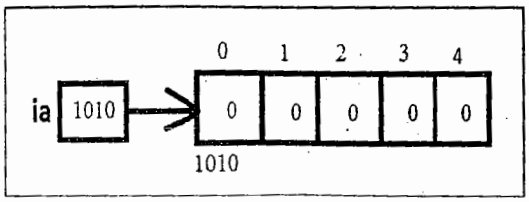
Syntax #2: Array object creation without explicit values or with default values

<Accessibility Modifier> <Modifier> <data type>[] <array name> = new <data type>[<size>];

Ex #1: Array creation with primitive type

int[] i = new int[5];

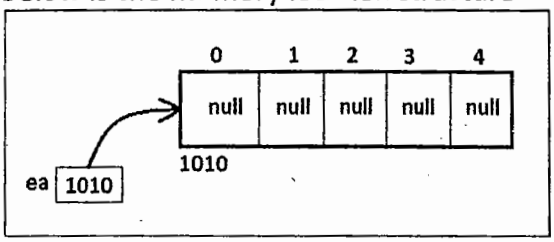
From the above statement array object is created with five int type variables. All locations are initialized with default value ZERO, because the array object is created with primitive datatype int. Below is the memory location structure



Ex #2: Array creation with referenced type
Example[] ea = new Example[5];

From the above statement array object is created with five Example type variables. All locations are initialized with default value ***null***, because the array object is created with referenced datatype Example.

Below is the memory location structure



The point to be remembered is in this array object creation statement, Example class objects are not created; rather only Example class type referenced variables are created to store Example class objects further.

Q) How many String objects are created from the below statement?

```
String[] s = new String[5];
```

A) **ZERO** String objects are created. It creates ONE String array object with 5 variables of type String with default value "null".

Rules in creating above array object

Rule #1: array size must only be specified at array creation syntax not at declaration syntax, it leads CE: ']' expected.

Find out CE from the below list

```
int[] i1 = new int[4];
int[5] i1 = new int[4];
```

Rule #2: array size is mandatory. If we do not pass array size it leads to
CE: ***array dimension missing***

Find out CE from the below list

```
int[]   ia1 = new int[5];
int[]   ia2 = new int[];
int[5]  ia3 = new int[];
```

Rule #3: size should be "0" or "+ve int number". If we pass int greater range value or incompatible type value it leads to CE, If we pass "-ve number" program compiles fine but leads to exception "***java.lang.NegativeArraySizeException***"

Find out errors in the below lines of code

```
int[] i1 = new int[3];
int[] i2 = new int[0];
int[] i3 = new int[-5];
int[] i4 = new int['a'];
int[] i5 = new int[10L];
int[] i6 = new int[10.345];
int[] i7 = new int[true];
int[] i8 = new int["a"];
```

Rule #4: While storing, reading and modifying array values, we must pass array index within the range of [0, array length-1]. If we pass index negative value or value >= length, it leads to RE: "***java.lang.ArrayIndexOutBoundsException***"

Find out errors in the below lines of code

```
int[] i = new int[5];

i[0]    = 6;
i[1]    = 5;
i[2]    = 4;
i[3]    = 7;
i[4]    = 8;
i[5]    = 9;
i[-3]   = 10;
i[10L]  = 10;
i['a']  = 10;
i[true] = 10;
```

Find out errors in the below lines of code

```
int[] i1 = new int[2];
int[] i2 = new int[-4];
int[] i3 = new int['a'];
int[] i4 = new int["a"];
int[] i5 = new int[34.5];
int[] i9 = new int[(int) 45.34];
int[] i6 = new int[0];
int[] i7 = {};
int[3] i8 = {1,2,3};

System.out.println(i3[91]);
System.out.println(i6[0]);
System.out.println(i3[34.56]);
System.out.println(i3['a']);
System.out.println(i3["a"]);
System.out.println(i1[-1]);
```

"*length*" property

length is a non-static final int type variable. It is created in every array object to store array size. We must use this variable for finding array object size dynamically for retrieving its elements.

Q) What is the output from the below statement?

```
Thread[] th = new Thread['a'];
System.out.println(th.length);//97
```

Q) How many Thread objects are created from above program?

Zero Thread objects are created. Only one Thread array object is created with 97 locations.

Write a program to create int type array with size 5. Then print its values on console.

```
class ArrayValues
{
    public static void main(String[] args)
    {
        int[] a = {50, 60, 70, 80, 90};
        System.out.println( a[0] );
        System.out.println( a[1] );
        System.out.println( a[2] );
        System.out.println( a[3] );
        System.out.println( a[4] );
```

Wrong Code

Reason:
static code, if size is change, code should be modified according to the current array size.

```
        for (int i = 0; i < a.length ; i++)
        {
            System.out.println( a[i] );
        }
```

this is code we implement in real time project.

Correct code

```
    }
}
```

Q) What is the output from the below program?

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

                int[] a;
                for (int i = 0; i < 10; i++)
                {
                  a[i] = i * i;
                }
        }
}
```

```
class Example{

        int[] a;
        public static void main(String[] args){
                for (int i = 0; i < 10; i++){
                  a[i] = i * i;
                }
        }
}
```

```
class Example{

        static int[] a;
        public static void main(String[] args) {
                for (int i = 0; i < 10; i++)        {
                  a[i] = i * i;
                }
        }
}
```

Rule #5: Source datatype and destination datatype must be compatible, else it leads to CE: *incompatible types*

For Example

```
Example[] ea1 = new Example[5];
Example[] ea2 = new Sample[5];
Example[] ea3 = new String[5];        CE: incompatible types
```

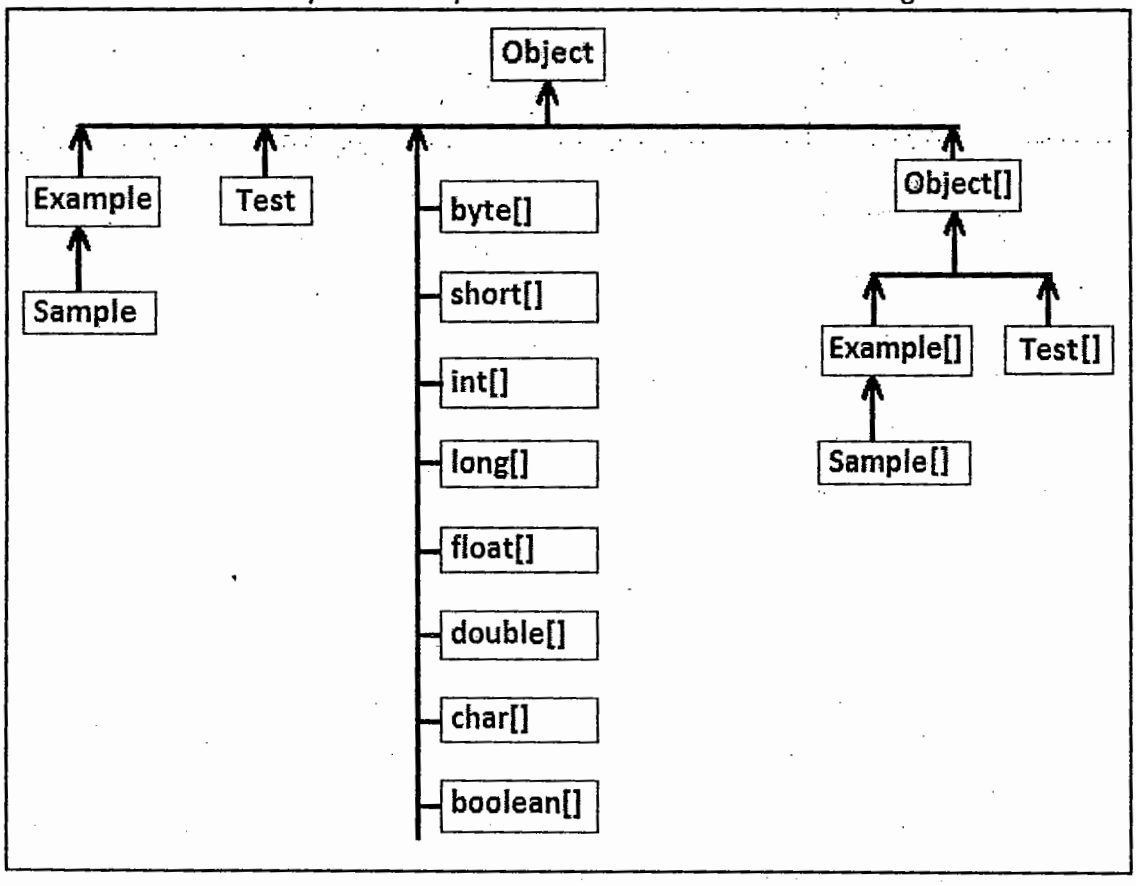
Array casting & exception in array casting

1. For all referenced types and arrays including primitive arrays java.lang.Object is super class.
2. For every array object, JVM internally creates a class with name "*datatype[]*".

   For example
   - the class for int type array is "int[]"
   - the class for Example type array is "Example[]"
   - the class for Object type array is "Object[]"

3. For all array objects created with referenced types the super class is "Object[]", which is subclass of "Object"
4. The array objects created with primitive types are not compatible with each other and their super class is "Object" not "Object[]".

Now observe below hierarchy and strictly remember and follow it in solving next bits.



Find out compile time errors in below list of array objects conversion

1.	int[]	i1	= new int[5];	**6.**	Example	e1	= new Example[5];
2.	int[]	i2	= new short[5];	**7.**	Object[]	obj	= new Example[5];
3.	Object[]	obj	= new short[5];	**8.**	Object	obj	= new Example[5];
4.	Object	obj1	= new int[5];	**9.**	Example[]	ea	= new Sample[5];
5.	Object	obj2	= new Object[5];	**10.**	Example[]	ea	= new Test[5];

SCJP Question:
Given:

```
11. public static void main(String[] args) {
12. Object obj = new int[] { 1, 2, 3 };
13. int[] someArray = (int[])obj;
14. for (int i : someArray) System.out.print(i + " ");
15. }
```

What is the result?
A. 1 2 3
B. Compilation fails because of an error in line 12.
C. Compilation fails because of an error in line 13.
D. Compilation fails because of an error in line 14.
E. A ClassCastException is thrown at runtime.

java.lang.ArrayStoreException

JVM throws this exception, if it identifies the incompatible object is passed to store in array location. If this problem is identified by compiler, it throws CE: "incompatible types".

Check below example

```
Object[] obj = new Example[5];
```

In this statement compiler thinks "Example array object is created with five locations and is stored in Object[] variable". This assignment is allowed as Example[] is a subclass of Object[].

1. obj[0] = new Example();
2. obj[1] = new Sample();
3. obj[2] = new Test();

At third line we experience "*java.lang.ArrayStoreException*", because, Test object is not compatible with Example.

It is not identified by compiler because it checks only type of referenced variable. In this case it considered obj[0], obj[1], obj[2] variables are of type java.lang.Object not Example. So it allows assignment. Actually they are of type Example, it is only known to JVM.

Find out CE and RE in the below lines of code

1. Example[] e = new Sample[5];

2. e[1] = new Sample();
3. e[2] = new Test();
4. e[3] = new Example();
5. e[4] = new Example[2];

Passing Array as Argument

To pass array object as an argument the method parameter must be the passed array object type or its super class type.

Below program shows passing int[] object as argument

```
//Example.java
class Example{
        static void m1(int[] ia){
                System.out.println("Array Size: "+ia.length);
                System.out.pritln("Its elements");
                for(int i = 0 ; i < ia.length ; i++){
                        System.out.print(a[i] + "\t");
                }
        }
}
//Test.java
class Test{
        public static void main(String[] args){

                //calling m1() method by passing referenced array objects
                int[] i1= {5, 3, 6, 7};
                Example.m1(i1);

                int[] i2 = new int[5];
                Example.m1(i2);

                //calling m1() method by passing unreferenced array objects
                Example.m1( new int[7] );  //array is passed with default values

                //Example.m1( {3, 4 , 5} );  CE:

                /*
                Q) How can we pass an array with user values without referenced variable?
                A) Anonymous array

                Syntax of anonymous array
                Combine both array creation syntaxes.
                Rule: Do not mention array size in [].
                Its size will be the number of values passing in {}.
                For example:
                */

                Example.m1(new int[] {3, 4, 5} );
```

```
        /* anonymous array can also be created with referenced variable. */
        int[] i = new int[]{3, 4, 5};

        /* But it is not recommended. It is only recommended to pass array as
        argument with explicit values without referenced variable*/
    }
}
```

If we modify array values with method parameter, the modification is effected to original referenced variable.

What is the output from the below program?

```
class Example{

    static void m1(int[] ia){
        ia[2] = 5;
    }

    public static void main(String[] args){

        int[] ia = {10, 20, 30, 40};
        m1(ia);

        for(int i = 0; i < ia.length ; i++){
            System.out.print(ia[i] + "\t");
        }
    }
}
```

What is the output if we call method by passing below array?

```
int[] ia2 = {1,2};
m1(ia2);
```

A) It leads to AIOBE in m1() method.

What is the output from the below program?

```
class Example{
    int x = 10;
    int y = 20;

    void m1(){
        x = 5;
    }
}
```

```
class Test{
        static void m1(Example[] e){
                e[2].m1();
        }
        public static void main(String[] args){
                Example[] e = {new Example(), new Example(), new Example(), new Example()};

                m1(e);

                for(int i = 0 ; i < e.length ; i ++){
                        System.out.println(e[i].x);
                        System.out.println(e[i].y);

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

In the below program what argument we must pass to execute else block.

```
class Example{
        static void m1(Object obj){
                if (obj instanceof Object){
                        System.out.println("If");
                }
                else{
                        System.out.println("Else");
                }
        }
}
```

Garbage collection with arrays

If array object is unreferenced, then all its internal objects are also unreferenced. So we can say when an array object is unreferenced not only array object, all its internal objects are eligible for garbage collection provided they do not have explicit references from other referenced variables.

For example:

```
Example[] e = {new Example(), new Example(), new Example(), new Example()};
```

e[1] = null;

- at this line number, e[1] referencing object is eligible for garbage collection.

e = null;

- at this line number, all 5 objects, including array object, are eligible for garbage collection.

Q) In the below program how many objects are eligible for gc?

```
class Test{
        static void m1(Example[] e){
                e[1] = null;
                e = null;
        }
        public static void main(String[] args){
                Example[] e = new Example[5];
                e[0] = new Example();
                e[1] = new Example();
                e[2] = new Example();
                Example e1 = new Example();
                e[3] = e1;

                line #1: e1 = null;
                line #2: m1(e);
                line #3: e = null;
        }
}
```

Declaring array as final

It is possible to declare array as final

For example:

```
//normal array, means non-final array
int[] ia1 = new int[5];

//final array
final int[] ia2 = new int[5];
```

Q) If we declare array as final, will all its locations are also final?

A) No, only array object referenced variable is final. It means in the above example only "ia2" is final not its array locations. It means we can modify array locations value, but we cannot assign new array object reference to this final referenced variable. It leads compile time error.

Find out CE in the below program. Comment the CE, execute and print output.

//ArrayAsFinal.java

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

                final int[] ia = new int[5];

                //modifing array locations value
                ia[1] = 5;
                ia[2] = 6;

                //modifing array referenced variable
                //ia = new int[6]; CE: cannot assign a value to final variable ia

                //printing array locations value
                for (int i = 0; i < ia.length ; i++){
                        System.out.println( "ia[" + i + "] --> " +  ia[i] );
                }
        }
}
```

Output

```
ia[0] --> 0
ia[1] --> 5
ia[2] --> 6
ia[3] --> 0
ia[4] --> 0
```

Q) Can we declare array locations as final?
A) No, because we are not creating array locations.

Q) Can we declare a class referenced variable as final?
A) Yes, in this case also only that referenced variable is final but not its object's variables.

```
final Example e = new Example();
```

- Here "e" only final, not "x, y" variables

Q) Can we declare a class object's variables as final, in the above case x, y?
A) Yes it is possible, because those variables are created by us. We must declare them as final in the class definition.

Types of array referenced variables
Like primitive variables and other referenced variables, we can also create array referenced variable as

- static
- non-static
- local

- If we create array object with static referenced variable, it is created at the time of class loading. That static referenced variable is created in method area and its array objet is created in heap area.
- If we create array object with non-static referenced variable, it is created at the time of that enclosing class object creation. That referenced variable is created in heap area in that enclosing class object and array object is also created in heap area.
- If we create array object with local referenced variable, it is created when that method is called. That referenced variable is created in that method's stack frame and object is created in heap area.

Check below program, it has CE comment it, then run and print out also draw JVM Architecture

//Test.java

```
class Test{
        static int[] ia1 = new int[5];
        int[] ia2 = {40, 50, 60, 70};

        public static void main(String[] args) {
                int[] ia3 = new int[3];

                System.out.println(ia1[1]);
                System.out.println(ia2[1]);
                System.out.println(ia3[1]);

                Test t = new Test()
                System.out.println(t.e2[1]);
        }
}
```

Q) If we create array object of a class is its class bytecodes are loaded into JVM?
No, if we create Example class array object, Example class bytecodes are not loaded into JVM. If at all in that array object if you are creating and adding "Example object", Example class is loaded into JVM.

For example:

```
class Example{
        static{
                System.out.printn("Example is loaded");
        }
}
```

Below statement ***does not load*** Example class
Example[] e = new Example[5];
Output: no output

Below statement ***loads*** Example class
Example[] e = {new Example(), new Example()};
Output: Example is loaded

Check below program, give output also JVM architecture. Find out CE, RE in Test.Java

//Example.java

```
class Example{

        int x = 10; int y = 20;

        static{
                System.out.println("Example is loaded");
        }

        Example(){
                System.out.println("Example object is created");
        }
}
```

//Test.java

```
class Test{
        static Example[] e1 = new Example[5];
        Example[] e2 = {new Example(), new Example()};

        public static void main(String[] args) {

                System.out.println("Test main");
                Example[] e3 = new Example[2];

                System.out.println("e3 array object is created");

                e1[1] = new Example();
                e3[1] = new Example();
                System.out.println(e1[1].x);
```

```
        System.out.println(e2[1].x);
        System.out.println(e3[1].x);

        Test t = new Test()
        System.out.println(t.e2[1].x);

        System.out.println(e1[0].x);
        System.out.println(t.e2[0].x);
        System.out.println(e3[0].x);
        System.out.println(t.e1[1].y);
    }
}
```

Types of Arrays based on dimensions

Java supports two types of arrays

1. Single dimensional arrays <- stores normal objects and values
2. Multidimensional arrays <- stores array objects

- Single dimensional array is also called "*array of objects or values*"
- Multi dimensional array is also called "*array of arrays*".

Note: In Java, multidimensional array is *array of arrays*.
Basically multidimensional arrays are used for representing data in table format.

Below is the syntax to create two dimensional array.

Two dimensional array

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

From the above statement

- one parent array is created with 3 locations and
- three child arrays are created with 2 locations.

- Parent array size gives two information
    1. parent array's number of locations
    2. number of child arrays

Below is the array object diagram for this two dimensional array.

Below is the table format diagram for the above two dimensional array object.

Below is the array object diagram for this two dimensional array.

Below is the table format diagram for the above two dimensional array object.

Rule: Base array size is mandatory where as child array size is not mandatory.

For Example

```
int[][] ia = new int[3][];
//int[][] ia = new int[][2]; ->CE: missing array dimension
```

- If *we do not pass* child array size, child array objects are not be created, only parent array object is created. Later developer has to pass array objects.
- If *we pass* child array size, all child arrays are created with same size. If we want to create child arrays with different sizes we must not pass child array size.

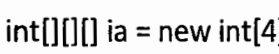
Jogged arrays

Multidimensional array with different sizes of child arrays is called Jogged array. It creates a table with different sizes of columns in a row. To create jogged array, in multidimensional array creation we must not specify child array size instead we must assign child array objects with different sizes as shown in the below diagram.

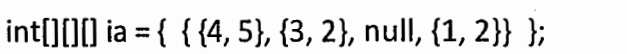
For Example:

```
int[][] ia = new int[3][]; -> base array is created with size 3

ia[0] = new int[2];
ia[1] = new int[3];
ia[2] = new int[4];

ia[0][0] = 50;
ia[2][3] = 70;
```

Below is the array object diagram for this two dimensional array.



Below is the table format diagram for the above two dimensional array object.



Below program shows printing multi dimensional array elements in table format

```
//MultiDimentionalArrayPrinter.java
class MultiDimentionalArrayPrinter {
        public static void main(String[] args) {

                int[][] ia = {{5, 6, 7}, { 4, 3}, {1}, {4, 5, 7, 8, 9}};


                for (int i = 0; i < ia.length ; i++ ){
                        for (int j = 0; j < ia[i].length ; j++ ){
                                System.out.print(ia[i][j] + "\t");
                        }
                        System.out.println();
                }
        }
}
```

Chapter 21

Working with Jar

- In this chapter, You will learn
  - Definition and need of jar
  - Difference between zip and jar files
  - Java binary file to create jar file
  - jar command options
  - syntax to create jar file
  - What is an executable jar file
  - Difference between executable jar file and normal jar file
  - Need of manifest.mf file
  - java command syntax to execute executable jar file
  - jar file execution using a batch file
- Ultimately by the end of this chapter- you will understand developing a normal and executable jar file.

Interview Questions

By the end of this chapter you answer all below interview questions

1. Array definition.
2. Need of array
3. Data type of array
4. Syntaxes to create an array and its rules
5. Array limitation
6. Finding length of an array
7. Storing and retrieving array elements and rules
8. Difference in storing primitive and reference type of data in arrays.
9. Array casting and exception in array casting
10. Passing array as method argument and return type
11. Anonymous arrays
12. Garbage Collection with arrays
13. Declaring array as final
14. Three types of array referenced variables and their memory locations
15. Types of Arrays based on dimension
    a. Single dimensional arrays
    b. Multi dimensional arrays
16. Purpose of the multidimensional arrays
17. Jogged Array
18. Var-arg parameter method, and rules on var-arg parameter

Definition and need of jar

jar stands for Java Archive. It is a java based compressed file. It is used for grouping java library files (i.e.; class files) for distributing library files as part of software.

Q) What is the difference between "zip" and "jar" extension files?

zip is a platform dependent compressed file it works only in windows, where as jar is a platform independent compressed file that can work in all OS, because it is created by using Java.

Java binary file for creating jar file

Using "jar" command we can create compressed files. It is a java binary file available in "jdk\bin" folder. It is used to compress or group packages, and project supporting files like properties and xml files for distributing project library files. The extension of this file is ".jar"

Q) How can we create a jar file by using "jar" command?

Jar command has different options to create and use jar file.

To know all its options: Open command prompt -> type jar -> and press Enter key on keyboard -> you will find the usage of jar command with various options.

The important options are

- -c => creates archive
- -v => shows list of files adding to this archive (verbose)
- -f => specify file name
- -x => extracts the files from jar
- -m => copies manifest file entries from an external file

Syntax to create jar file

Open command prompt -> change directory path to present working directory (pwd) -> run below command

jar -cvf <jar file name> <folders and files name with space separator>

For example:

- jar -cvf test.jar abc xyz bbc.txt
  only abc, xyz, and bbc.txt files are added to jar file from present working directory and jar file is stored in the same pwd.

- jar -cvf test.jar *.*
  all files & folders of the pwd are added.

- jar -cvf test.jar *.txt
  only .txt extension files are added to this jar file.

The above jar file is a normal jar file, not an executable jar file.

Creating an executable jar file

The jar that allows us to execute a main method class available in it is called executable jar file.

A short story on manifest.mf file

To develop executable jar file we must configure main method class name in its "manifest.mf" file. This file is available in META-INF folder which is created automatically in every jar file by jar command. This file is a properties file contains (name, value) pairs. All names are predefined given by SUN, and that name's value must be assigned by developer. The required name and its associated value both must be placed by developer in the manifest.mf file.

The property name for configuring the class name is "Main-Class"

For Example, if we want to run Student class via a jar file we must place *Main-Class* key with class name *Student* as shown below in manifest file

Main-Class: Student

Q) How can we execute this jar file?

By using "java" command with the option "-jar"

Syntax:

> java -jar <jar file name>

For example:

>java -jar test.jar

Then java command executes the class that is configured in manifest.mf file with the property name "Main-Class"

Procedure to develop an executable jar files to start its execution with *Calculator* class.

Step #1: create a folder called "test" in "D" drive

D:\test

Step #2: Create a java file "Test.java" and save it in this test folder

// *Calculator*.java

```
import java.util.*;
public class Calculator {
        public static void main(String[] args){
                Scanner scn = new Scanner(System.in);

                System.out.print( "Enter first number: ");
                int i1 = scn.nextInt();

                System.out.print("Enter second number: ");
                int i2 = scn.nextInt();

                Addition.add(i1, i2);
```

```
    }
}
//Addition.java
public class Addition{
    public static void add(int a, int b){
        System.out.println("Addition result: " + (a+b));
    }
}
```

Step# 3: Compile above java files

```
D:\test>javac *.java
```

Step #4: Creating jar file

Step #4.1: Create MANIFEST.MF file explicitly in "test" folder with the property name Main-Class as follows

- Open Editplus
- Click File -> New -> Normal Text
- Type Main-Class: Calculator
- Save it in "D:\test" folder with name MANIFEST.MF

Rule: After all properties there should be one empty line in manifest.mf files else properties are not copied into jar file

Step #4.2: Create jar file with this explicit manifest file entries using below command.

```
D:\test>jar -cvfm test.jar MANIFEST.MF *.class
```

-m option copies manifest file entries into jar's manifest file.

** executable jar file is ready**

Now let us execute it

Step# 5: Execution

```
D:\test>java -jar test.jar
Enter first number: 5
Enter second number: 6
Addition result: 11
```

Executing jar file using a batch file

Q) Can user execute above command?

No, because user do not know java.

Solution: write a batch file with above command as shown below

1. Open Editplus
2. Click File -> New -> Normal Text
3. Type java -jar test.jar

4. Save this in "D:\test" folder with nametest.bat

Batch file execution

1. Open Command prompt
2. Change directory path to "D:\test"
3. Run it as below

```
D:\test>test.bat
Automatically it executes "java" command and prints output
D:\test>java -jar test.jar
Enter first number: 765
Enter second number: 123
Addition result: 888
```

Q) How can we access classes from jar files from other directories/packages?

A) We must update Classpath environment variable with that jar file path including its name

```
E:\examples>Set Classpath=D:\test\test.jar;%Classpath%
E:\examples>java Calculator
Enter first number: 111
Enter second number: 222
Addition result: 333
```

Note: We cannot run executable jar file from other directory

========== END ==========