Concept Of OOPS:

Encapsulation, Polymorphism and Abstraction, Inheritance

1.Method Overloading
2. Data Shadowing
3. Data Hiding
4. Method Overriding
5. Function Hiding
6. UpCasting
7. DownCasting

1. Method Overloading:

Method overloading says you can have more than one function with the same name in a one class having a different prototype.

2. Data Shadowing:

Whenever class level variable and local level variable as same name then this concept is called data shadowing.

3. Data Hidding:

Whenever a parent class and a child class both are having same data member then this concept is known as Data Hidding.

4. Method Overriding:

In OOPS, Whenever a Parent class and a Child class both are having complete same function then this concept is known as Method Overriding.

5. Function Hiding:

Whenever a Parent class and Child class both are having complete same static function then this process is known as function Hidding.

6. UpCasting:

The Reference ID of the Child class object can be put into the Referenced variable of Parent class but vise versa is not true and this concept is known as UpCasting.

7. Down Casting:

The process of getting back the Reference ID of a Child class object from the Reference variable of a Parent class into the Reference variable of a Child class is known as Down Casting. Before doing the DownCasting, UpCasting must be done.
OOPS Pillars

OOPS Pillars

- Encapsulation
- Polymorphism
- Abstraction
- Inheritance

Encapsulation:

Combining of state and behavior in a single container is known as encapsulation.

Encapsulation in Java is a process of wrapping code and data together into a single unit. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class.

For example, capsule i.e. mixed with several medicines.

Example: A Bank. It only allows limited access, when we want to withdraw or deposit, we do this through application form same in java encapsulation, work happens with member functions.

Another example is file sharing apps they don't give us direct access to other device but they only allow us to receive and send particular data.

- Private fields are an example of Strong Encapsulation.

- Public fields are an example of Weak Encapsulation. In this case, fields are still encapsulated in the class, but visible to outside world.

- Protected/default fields show moderate Encapsulation.

Polymorphism: One name many form.

Whenever an object performes a different different behavior in a different different circumstances then this behavior of an object is known as polymorphism. Polymorphism reduces the complexity of an object.

Polymorphism is always achieved via behaviour of an object. Properties of an object do not play any role in case of polymorphism.

Polymorphism is divided into two parts, compile time polymorphism and runtime polymorphism.

- Whenever an object is bound with their functionality at compile time then this is called compile time polymorphism.

- Whenever an object is bound with their functionality at runtime then this is known as runtime functionality.

Complile time: **Method Overloading** and **Operator Overloading.**

Run-time: Function Overriding.

#. Java does not support operator overloading explicitly. this only work implicitly we can't add our own extra functionality with operator overloading.

#. Java does not support compile time polymorphism. In Java all the instance functions of a class are implicit virtual.

#. Java programming language provides two kinds of methods, Instance methods and Class methods. Class methods also called static method.

The difference between these two kinds of methods are that Instance methods require an instance before they can be invoked; Class Methods do not need instance to invoke.

Instance methods use dynamic binding also called late binding. Class methods use static also called early binding.

The JVM uses two different instructions to invoke these two types of methods. InvokeVirtual for Instance methods and InvokeStatic for class methods (Static Methods).

Method Overloading

Function overloading says you can have more than one function with the same name in a one class having a different prototype.

There are 5 function prototypes:-

1. Access specifier
2. Access modifier
3. Return type
4. Function name
5. Arguments

#. In Java access specifiers and access modifier is called as access modifier.

#. There are 10 types of access modifiers are available in Java programming language. Public, Private, Protected, Default, Static, Final, Abstract, Synchronized, Transient, and Volatile.

#. Access modifiers and access specifier do not play any role in case of function overloading.

#. Return type of functions do not play any role in case of function overloading.

#. Function overloading can only be achieved by changing only in arguments.

#. To achieve function overloading we need to change the number of argument in each function.

#. If we want to keep number for arguments same in each function then we need to change the data type of their arguments.

#. If it does not get exact function match then it will apply type promotion that can create Ambiguty error.

Programs: Method Overloading

```
class methodOverLoading{

    public static void main(String... arg){
        AddClass ac = new AddClass();
        System.out.println("string: " + ac.add("Ahmad ", "Sayeed"));
        System.out.println("int: " + ac.add(15, 20));
        System.out.println("Float: " + ac.add(7.5f, 8.0f));
        System.out.println("Double: " + ac.add(5.5d, 9.90d));
        System.out.println("Long:" + ac.add(5L,6L));
        System.out.println("Boolean:" + ac.add(true,false));
        System.out.println("Char: " + ac.add('C', 'R'));
    }
}

class AddClass{

    String add(String s1, String s2){
        return s1 + s2;
    }

    int add(int n1, int n2){
        return n1 + n2;
    }

    float add(float f1, float f2){
        return f1 + f2;
    }

    double add(double d1, double d2){
        return d1 + d2;
    }

    long add(long l1, long l2){
        return l1 + l2;
    }

    int add(char c1, char c2){
        return c1 + c2;
    }
    /* We can't do this
    char/String add(char c1, char c2){
        return c1 + c2;
    }
    */
    /* We can't do this
    boolean add(boolean b1, boolean b2){
        return b1 + b2;
    }*/
```

```
}
```

Program: Method Overloading Bad Situation

```
/* If It does not found exact match function then it will search for bigger data
type. It means, it will do TypePromotion that will create confusion for
compiler.*/

/*Let's see an example*/
class MOBadSituation{
    public static void main(String... a){

        new MOBadSituation().add(10, 20L); // right & working
        new MOBadSituation().add(25L, 10); // right & working

        /*
            Bad Code example
            As we can see arguments are in int.
            We have both function long and int so match function is not avaible*/
            Now it will type promotion and find if long argument is avaible to
            place int value. And now both function has that long argument so JVM
            will confuse and print error
        */

            new MOBadSituation().add(10, 10); // will give compilation error

    }

    int add(int x, long y){
        System.out.println(x + y);
        return 0;
    }

    int add(long x, int y){
        System.out.println( x + y);
        return 0;
    }
}
```

Program: via parameter

```
class MOThroughArgument{
    public static void main(String...a){
        /*Way 1*/
      new MOThroughArgument().display("String 1 : ", "String 2 : ", "String 3");
      new MOThroughArgument().display(5, 11, 16);

      /*Way 2*/
      int sum1 = new MOThroughArgument().add(12, 13);
      int sum2 = new MOThroughArgument().add(10, 12, 13);


      System.out.println(sum1);
      System.out.println(sum2);
  }
```

```
    /*  Way 1: Creating function with same number of arguments and with
             different data type arguments*/

    /*  If you want to keep number of arguments in each function same and still
           want to overload then change the data type of their arguments*/

    void display(String s1, String s2, String s3){
        System.out.println(s1 + s2 + s3);
    }

    int display(int n1, int n2, int n3){
        System.out.println(n1 + n2 + n3);
        return 0;
    }

    /*  Way 2: creatting function with different number of arguments with same
        data type*/
    /*  Change the number of arguments in each function*/

    int add(int x, int y){
        return x + y;
    }

    int add(int x, int y, int z){
        return x + y + z;
    }
}
```

Program: via return type

return type of function do not play any role in case of function overloading

Below program will show error because their arguments are same and retun type does not affect. Access specifier/modifier don't play any role in case of function overloading.

```
class MOThroughReturn{
    public static void main(String... a){

        new MOThroughReturn().add("Ahmad ", "Sayeed");
        new MOThroughReturn().add("Ahmad ", "Sayeed");
    }

    public float add(String s1, String s2){
        System.out.println(s1);
    }

    public int add(String s1, String s2){
        System.out.println(s1);
    }

}
```

Method Overriding Member Function Inheritance: Method Overriding:

#. In OOPS, Whenever a Parent class and a Child class both are having complete same function then this concept is known as Method Overriding.

#. Function Overriding concept is made to acheive dynamic binding only. In run-time, compiler decides which parent's or child's function needs to run is called dynamic binding.

#. From JDK 1.5 we can do Overriding between parent function and child function after changing the return type but we need to follow 3 rules strictly.

- **R1.** Parent class function and a Child class function both must be having a return type as Referenced that means both functions must return the object of some class.
- **R2.** The class whose objects are returned by the parent class function and by the child function must be having a parent to child to relationship.
- **R3.** Parent class function must return other parent class object and child class function must return other child class object.
-

Rule: The role of "access Modifiers" in case of method overriding:

- Same to same access modifiers are allowed.
- Parent's weaker access modifiers To Child's strong access modifiers are allowed.
- Parent's Strong access modifiers To Child's Weakest access modifiers are not allowed.

Rule: Whenever a Parent class and Child class both are having complete same static function then this process is known as function Hidding.

#. UpCasting: To acheive dynamic binding, it requires to do UpCasting.

Rule: The Reference ID of the Child class object can be put into the Referenced variable of Parent class but vise versa is not true and this concept is known as UpCasting.

Rule: If we have done UpCasting then we can not access the personal method/Child class method by the reference variable of Parent class.

Rule: If a Child class has overridden the Parent class method then by the Child class object always a overridden method is called. Hardly matters whether the Referenced ID of the Child class object kept into the reference variable of Child class or into the reference variable of Parent class.

#. DownCasting: The process of getting back the Reference ID of a Child class object from the Reference variable of a Parent class into the Reference variable of a Child class is known as Down Casting. Before doing the DownCasting, UpCasting must be done.

Rule: If a Parent class has made any function "**final**" then it can not be overridden by the Child class. It means to stop function overriding we make a function final.

#. After Method Overriding, "super" is the only way to access the Parent's function.

Program: Method Overriding

```
class Parent{

    //Same to same to overriding: Allowed
    protected void sameToSame(){
        System.out.println("I am SameToSame From Parent");
    }

    //Weakest To Strong: Allowed
    protected void weakToStrong(){
        System.out.println("I am weakToStrong From Parent");
    }

    //Strong to Weak: Not Allowed
    protected void strongToWeak(){
        System.out.println("I am strongToWeak From Parent");
    }

    //Static To Static: Allowed: But it is not Overriding, It is Function Hidding
    static void staticToStatic(){
        System.out.println("I am staticToStatic (Function Hidding) From Parent");
    }

    //Static To Non Static: Not Allowed
    //Non-Static To Static: Now Allowed

    //Final: We can't override this parent function
    final public void finalToAny(){
        System.out.println("I am finalToAny From Parent");
    }

}

class Child extends Parent{

    protected void sameToSame(){
        System.out.println("I am SameToSame From Child");
    }

    public void weakToStrong(){
        System.out.println("I am weakToStrong From Child");
    }

    // It will create compile time error becasue we can't overriding from
    strongToWeak

    //So i am commenting
    /*private void strongToWeak(){
        System.out.println("I am strongToWeak from Child");
```

```
    }*/

    static void staticToStatic(){
        System.out.println("I am staticToStatic (Function Hiding) From Child");
    }

    /* //Will create error becasue with final we can't method overriding
    public void finalToAny(){
        System.out.println(" I am finalToAny From Child");
    }
    */

    public static void main(String...a){

        Child c = new Child();

        // Same to Same MethodOverriding
        c.sameToSame();
        // Let's proof that overriding is having
        ((Parent)c).sameToSame();

        // Weak To Strong MethodOverriding
        c.weakToStrong();
        // Let's Proof that overriding is having
        ((Parent)c).weakToStrong();

        // StrongToWeak : Not allowed
        /*
            c.strongToWeak();
        */

        //StaticToStatic: Allowed (Function Hiding)
        c.staticToStatic();
        //Let's Proof that it is only function hidding not overriding
        ((Parent)c).staticToStatic();

        //final To Any: can't happen overriden with final modifier
        //c.finalToAny();

    }
}
```

Inheritance:

Inheritance is a process of achieving a reusability among the objects and it is also use for dynamic binding.

#. In run-time, compiler decides which parent's or child's function needs to run is called dynamic binding.Function Overriding concept is made to acheive dynamic binding only.

Why use inheritance in java?
The major use of Inheritance is Method overriding (so runtime polymorphism can be achieved) and For Code Reusability.

#. There are different types of inheritance are available in OOPS, Single Inheritance, Multilevel inheritance, Multiple Inheritance, Hierarchical Inheritance, and Hybrid Inheritance.

#. When a class extends multiple classes i.e. known as multiple inheritance. For Example:
Java does not support Multiple Inheritance cause of diamond inheritance problem.
To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

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

To achieve an inheritance, we need to create a relationship between two classes and this is known as "IS A" relationship.

#. Is there any alternate of inheritance? Yes there is an alternate of inheritance called Association (Has A Relationship).

There are only two ways to use the properties and behaviour of a class using association and inheritance.

Types of association's are composition and aggregation. Aggregation represents weak relationship whereas Composition represents strong relationship.

Q8. What do you mean by aggregation?

Aggregation is a specialized form of Association where all object have their own lifecycle but there is ownership and child object can not belongs to another parent object. Let’s take an example of Department and teacher. A single teacher can not belongs to multiple departments, but if we delete the department teacher object will not destroy.

Q9. What is composition in Java?

Composition is again specialized form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. Child object dose not have their lifecycle and if parent object deletes all child object will also be deleted. Let’s take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room can not belongs to two different house if we delete the house room will automatically delete.

Q10. Inheritance is having at Complile Time or Run time. Ans: **Run-time**.

Q11. Whenever we create the object of a child class then parent class object will be create automatically or not.
And: NO.

#. By default all the data members and Member functions of a parent class are available for child class if they are not private.

#. Whenever we apply Inheritance then all parent and child classes stores inside child's memory heap area in separate block.

#. In Java all private data members of a parent class are inherits in child class but we can't use them directly. But don't argue on this because OOPS clearly says we can't use private data members of a parent class in child class.

#. Interface does not represent objects so we can't say this is inheriting objects properties and behavior.

#. To get data members and member functions of inherited class we can use this syntex.

((ClassName)this).dataMember

((ClassName)this).function

Program: **This is how inheritance works.**

```
class ClassSub extends ClassSuper{
    ClassSub(){
        super(25);
        System.out.println("ClassSub Constructor and " + this.x);
    }

    public static void main(String...d){
        new ClassSub();
    }
}

class ClassSuper{
    int x;
    ClassSuper(int y){
        this.x = y;
        System.out.println("ClassSuper Constructor");
    }
}
```

Data Member Inheritance

Whenever a parent class and a child class both are having same data member then this concept is known as Data Hidding.

We can use Super class data members through (super) keyword. "Super" can not be used in any static function.

However we can't print "super" like we print Reference variable or "this" variable.

If we don't do data hiding then we can access the data members and functions of a super class through three ways.
Direct, with "super", and with "this" keyword.

Actually in Child class by default in behind of every data members and functions of a super class has "super" keyword implicitly.

Data member overriding does not happens.

Program: Data Member Inheritance

```
class Teacher{
    int knowledge = 10;
    int classHour = 1;
    private int exp = 18;
    static String bloodGroup = "A+";
}

class Student extends Teacher{

    //Data Hiding
    int knowledge = 6;
    String bloodGroup = "O+";
    public static void main(String...a){
        staticGet();

        Student s = new Student();
        s.get();
    }

    //inside static function situation
    static void staticGet(){
        System.out.println("------------Static Output --------------");
        Student s = new Student();
        System.out.println(s.classHour); // working
        System.out.println(s.knowledge); // working

        / if we want to get parent data member
        System.out.println(((Teacher)s).knowledge);

        //System.out.println(((Teacher)this).knowledge); Not allowed
        //System.out.println(((Teacher)super).knowledge); Not allowed
    }

    //inside Non-Static function situation
```

```
    void get(){
        System.out.println("----------- Non-Static Output ---------------");

        System.out.println(knowledge); // working with data hiding
        //System.out.println(exp); // we can't cause private
        System.out.println(bloodGroup);
        System.out.println(this.bloodGroup); // it wil fetch current class
        //// if we want to get parent data member
        System.out.println(super.bloodGroup); // super will fetch parent data
        System.out.println(((Teacher)this).bloodGroup); // it wil fetch parent
class
        //System.out.println(((Teacher)super).bloodGroup); // does not working
    }

}

class Nephew extends Student{
    public static void main(String... q){
        System.out.println("---------------- Nephew ----------------");
        Nephew n1 = new Nephew();
        n1.nephewFunc();
    }

    void nephewFunc(){
        System.out.println(((Teacher)this).bloodGroup);
        Nephew n1 = new Nephew();
        //System.out.println(((Teacher)super).bloodGroup); //Does not works
        System.out.println(((Teacher)n1).bloodGroup); //Working
    }
}

//All over:
//((ClassName)this/Refid).dataMember does not depends on inheritance type
```

Program Multiple Inheritance

```
// Let's proof that Java does not support multi-inheritance
// This program will give compilation error becasue java does not multi-
inheritance
class MultiInheritance extends ClassSuper, ClassSuperSuper{
    MultiInheritance(){
        System.out.println("MultiInheritance Constructor");
    }

    public static void main(String...d){
        new MultiInheritance();
    }
}

class ClassSuper{
    ClassSuper(){
        System.out.println("ClassSuper Constructor");
    }
}

class ClassSuperSuper{
```

```
    ClassSuperSuper(){
        System.out.println("ClassSuperSuper Constructor");
    }
}
```

Program: Inheritance is also used for improvements on existent program.

```
class Swift extends Vehicle {

    String color;
    int noOfSeats;
    Swift(String color, int noOfSeats){
        this.color = color;
        this.noOfSeats = noOfSeats;
        this.wheels = 6;      // Improvement
    }

    public static void main(String...a){
        Swift s = new Swift("red", 5);
        System.out.println("Vehicle Configuration");
        System.out.println("Wheels = " + s.wheels );
        System.out.println("Color = " + s.color );
        System.out.println("No Of Seats = " + s.noOfSeats );
    }
}


class Vehicle{
    int wheels = 4;
}
```