**Method Overloading:-**

**Polymorphism:** Same object having different behavior.

**Compile time Polymorphism:**A polymorphism which exists at the time of compilation is called compile time or early binding or static polymorphism.

Example:-   
***Method Overloading:***Whenever a class contain more than one method with same method name and different types of parameters called method overloading.

Syntax: return-type method\_name (para1);  
 return-type method\_name (para1, para2);

Note:

* Return type of method can be different.
* Different Parameter means, Number of parameters can be different or data type of parameter can be different.
* Method Overloading is an example of Compile time Polymorphism.
* This is the first type of Polymorphism.

**Method Overriding:-**

**Polymorphism:** Same object having different behavior.

**Run time Polymorphism:**A polymorphism which exists at the time of execution is called runtime polymorphism. There is no role of compiler but JVM.

Example:-   
***Method Overriding:***Whenever we writing method in super and sub classes in such a way that method name and parameter must be same called method overriding.

We can’t perform Method Overriding without inheritance

Syntax:

class A{  
 void shape(){

}  
}

Class B extends A{  
void show(){  
}

}

**Rule:**

If method doesn’t present in super class gives compilation error.

If method present in super class then check, is it overridden or not.   
If No, JVM will call the method of super class.  
If Yes, JVM will call the method of sub class.

Note:

* Here, we create object of sub-class & its reference would be of super class.
* We’ve to used @Override annotation above override method to indicate other developers & compilers that the method is overridden method.
* We can use *super* keyword to call the methods of super/parent class.
* In method overriding, Parameter should be same (i.e. number of parameters & parameter having same data types) and return type of the method should be same.

**Static Keyword:-**

**Static Variable:**

* Whenever we don’t want a variable object specific, we make it static variable. (Means as the value of that variable is get change for 1 object of that class, it will get automatically changed for all the objects of that class).

Ex:

If we make an Employee class having attribute String CEO. Then Number of employees belongs to that class having same CEO.

In future if CEO of that company get change, then it will effect all the employee object of employee class.

So, to get rid of changing the name of the CEO from each employee object, we just make the CEO variable static & just change the name of the CEO from the last employee object, it will automatically change every employee object’s CEO name.

* We don’t need to access static variable from the instance of the class.

**Note:**

* If you want to initialize the non-static variables, we can use constructor(). Constructor get executed when we create an object.
* If you want to initialize the static variables, we can use static{} block. Static block executed when we load a class. And as we know that, no matter how many objects we created, the class will load only once.
* The static{} block executed before the constructor.
* We can’t access a non-static variable inside a static method.

Ex: We will get compilation error, if we try to access, a non-static variable inside the main method.

**This Keyword:-**

Every class has a unique reference number(like, A@7a81197d). So whenever we create an object of the class it generate a unique reference number. This unique reference number is referred by the reference variable which directly refer the instance of a class.

Similarly, ‘this’ keyword refer the unique reference number of the current class.

1. So, ‘this’ keyword refers to the current object inside a **method** or **constructor**.
2. Whenever the name of instance and local variables both are same then our runtime environment JVM gets confused that which one is local variable & which one is instance variable, to avoid this problem we should use ‘this’ keyword.  
   Example:

class A{  
 int a; //**Instance variable (associated with the class)**  
 A(int a){ //**Local Variable**  
 a = a;   
 S.O.P(a);  
 }  
}

1. It is also used when we want to call the default constructor of its own class.  
   Example:

class A{

//Default Constructor

A() {

}

//Parameterize Constructor

A(int a){

this();

}

}

1. It is also called parameterized constructor of its own class.  
   Example:

class A{

//Default Constructor

A() {   
 this(10);

}

//Parameterize Constructor

A(int a){

}

}

**Super Keyword:-**

Super Keyword refers to the objects of super class, it is used when we want to call the super class’s variable, method & constructor through sub class object.

Note:-

1. Whenever the Super class & Sub class variable and method name both are same it can be used only.
2. To avoid the confusion between super class and sub classes variables and methods that have same name, we should use Super keyword.
3. We can access variable, method & constructor of super/parent class using super keyword.
4. By default, default constructor have super keyword at its top.  
   By default compiler, reserve the space of super keyword inside default or non-parametrize constructor. Hence, if we have a default constructor inside sub class and as we create its instance, it automatically call the constructor of super /parent class the move back to the constructor of sub class.  
   Paranthesis () is associated with the super keyword, in order to call the parent constructor().  
   But in case of parameterize parent constructor, we’ve to pass some value from the super(100) keyword.  
   Example:

class A{

int a = 10;

//Default Constructor

A(){

System.out.println("This is Class A constructor");

}

void fun() {

System.out.println("This is paren class");

}

}

class B extends A{

int a = 20;

B(){

//By default here, super keyword already reserve space to call super class constructor

System.out.println("This is Class A constructor");

}

void show() {

System.out.println(super.a);

super.fun();

System.out.println(a);

}

}

**Final Keyword:-**

Final is a non-access modifier which provides restriction in Java we can use final in three ways:-

Final Variable, Final Method & Final Class

**Final Variable:**

Once we declare a variable as a final we can’t perform re-assignment.  
Example: final int =12;

**Final Method:**

Whenever we declare a method as a final it can’t be overridden to our extended class.

Example: final void atmPIN(){  
 S.O.Pln(“2342”);  
}

**Final Class:**

Whenever we declare a class as a final it can’t be extended or inherited to sub classes.  
Syntax: final class A{   
 }

**Interface:-**

Interface is just like a class, which contains only abstract method or collection of abstract methods called interface.

To achieve interface java provides a keyword called implements.

**Note:**

1. Interface methods are by default public & abstract.
2. Interface variables are by default public + static + final.
3. Interface method must be override inside the implementing class. If it doesn’t override, the sub class becomes abstract class.
4. Interface nothing but deals between client & developer.  
   The contract between client & developer called interface.

**Abstract Keyword:-**

**Abstract Class:**

A class which contains the abstract keyword in its declaration is called abstract class.

**Note:**

* We can’t create object for abstract class. But we can create its reference variable.
* Any Super/parent class has the capability to store the object of its sub class.
* It may or may not contain abstract methods.
* It can have abstract & non-abstract methods.
* Abstract class should always be super/parent class in order to use it.
* If a class contain partial implementation then we should declare a class as abstract.