Chapter 2 - Unit 2

Interfaces

- **Why use Java interface?**
- The main reasons to use interface. They are given below.
    - **It is used to achieve abstraction.**
    - **Java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.**
- **Important Points:**
- The interface is *a mechanism to achieve* abstraction. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- An interface is a reference type in Java.
- It is similar to class.
- It is a collection of abstract methods.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

- Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways –

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including –

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- The methods that implement an interface must bedeclared public.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.
-

Differences between classes and interfaces

Sr. No.	Key	Class	Interface
1	**Supported Methods**	A class can have both an abstract as well as concrete methods.	Interface can have only abstract methods. Java 8 onwards, it can have default as well as static methods.
2	**Multiple Inheritance**	Multiple Inheritance is not supported.	Interface supports Multiple Inheritance.
3	**Supported Variables**	final, non-final, static and non-static variables supported.	Only static and final variables are permitted.
4	**Implementation**	A class can implement an interface.	Interface can not implement an interface, it can extend an interface.
5	**Keyword**	A class is declared using class keyword.	Interface is declared using interface keyword.
6	**Inheritance**	A class can inherit another class using extends keyword and implement an interface.	Interface can inherit only an inteface.
7	**Inheritance**	A class can be inherited using extends keyword.	Interface can only be implemented using implements keyword.
8	**Access**	A class can have any type of members like private, public.	Interface can only have public members.
9	**Constructor**	A class can have constructor methods.	Interface can not have a constructor.

Defining an interface

- An interface is defined much like a class. The general form of an interface is:

  Interfaceinterface_*name*

  {

  type final-varname1 = value;

  type final-varname2 = value;

  ….

  ….

  type final-varnamen = value;

  return-type method-name1(*parameter-list*)*;*

  return-type method-name2(*parameter-list*)*;*

  ….

  ….

  return-type method-nameN(*parameter-list*)*;*

  }

Implementing interface

- The general form of a class that includes the implementsclause looks like this:

  Classclassname implements interface1 ,interface2...n

  { // class-body }

Example 1:one interface with one class

```
Interfaceinf
{
void method();
}
class sample implements inf
{
public void method()
{
System.out.println("this is implementation of interface method");
}
}
class demo
{
public static void main(String args[])
{
sample s=new sample();
s.method();
}
}
```

Output:

This is implementation of interface method

Example 2: One interface with one class

```
Interfaceinf
{
void method1();
void method2();
}
class sample implements inf
{
public void method1()
{
System.out.println("this is implementation of interface method1()");
}
public void method2()
{
System.out.println("this is implementation of interface method2()");
}
}
class demo
{
public static void main(String args[])
```

```
{
sample s=new sample();
s.method1();
s.method2();
}
}
```

Output:

```
This is implementation of interface method1()
This is implementation of interface method2()
```

Example 3: One interface with multiple classes

```
Interfaceinf
{
void method();
}
class sample1 implements inf
{
public void method()
{
System.out.println("this is implementation of interface method in
sample1 class");
}
```

```
}
class sample2 implements inf
{
public void method()
{
System.out.println("this is implementation of interface method in
sample2 class");
}
}
class demo
{
public static void main(String args[])
{
sample1 s1=new sample1();
s1.method();
sample2 s2=new sample2();
s2.method();
}
}
```

Output:

```
This is implementation of interface method in sample1 class
This is implementation of interface method in sample2 class
```

Example 4: Multiple interfaces with one class

```
interface inf1
{
void method1();
}
interface inf2
{
void method2();
}
class sample implements inf1,inf2
{
public void method1()
{
System.out.println("this is implementation of inf1 method in sample class");
}
public void method2()
{
System.out.println("this is implementation of inf2 method in sample class");
}
}
```

```
class demo
{
public static void main(String args[])
{
sample s=new sample();
s.method1();
s.method2();
}
}
```

Output:

```
this is implementation of inf1 method in sample class
this is implementation of inf2 method in sample class
```

Example 5: Multiple interfaces with multiple classes

```
interface inf1
{
void method1();
}
interface inf2
{
void method2();
}
```

```
class sample1 implements inf1
{
public void method1()
{
System.out.println("this is implementation of inf1 method in sample class1");
}
}
class sample2 implements inf2
{
public void method2()
{
System.out.println("this is implementation of inf2 method in sample class2");
}
}
class demo
{
public static void main(String args[])
{
sample1 s1=new sample1();
s1.method1();
```

```
sample2 s2=new sample2();
s2.method2();
}
}
```

Output:

```
This is implementation of inf1 method in sample class1
This is implementation of inf2 method in sample class2
```

Extending interfaces

- **The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.**
- An interface can extend another interface in the same way that a class can extend another class.

Extending Multiple Interfaces

- **An interface can extend more than one parent interface.**
- A Java class can only extend one parent class.
- Multiple inheritance is not allowed.
- Interfaces are not classes, however, and an interface can extend more than one parent interface.
- The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

- **Syntax:**

```
interface name1
{
//Body
}
interface name2 extends name1
{
//Body
}
```

Example 1:

```
interface inf1
{
void method1();
}
interface inf2 extends inf1
{
void method2();
}
class sample implements inf2
{
public void method1()
{
```

```
System.out.println("this is implementation of inf1 method in sample class");
}
public void method2()
{
System.out.println("this is implementation of inf2 method in sample class");
}
}
class demo
{
public static void main(String args[])
{
sample s=new sample();
s.method1();
s.method2();
}
}
```

Output:

```
This is implementation of inf1 method in sample class
This is implementation of inf2 method in sample class
```

Example 2:

```
interface inf1
{
void method1();
}
interface inf2 extends inf1
{
void method2();
}
interface inf3 extends inf2
{
void method3();
}
class sample implements inf3
{
public void method1()
{
System.out.println("this is implementation of inf1 method in sample class");
}
public void method2()
{
```

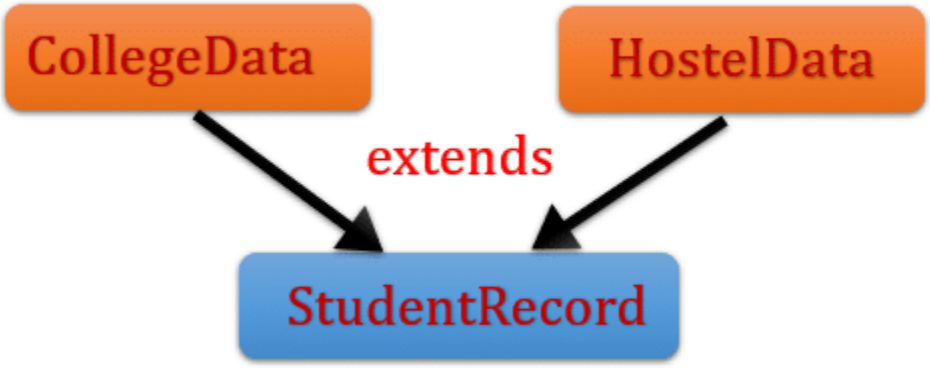
```
System.out.println("this is implementation of inf2 method in sample class");
}
public void method3()
{
System.out.println("this is implementation of inf3 method in sample class");
}
}
class demo
{
public static void main(String args[])
{
sample s=new sample();
s.method1();
s.method2();
s.method3();
}       }
```

Output:

```
This is implementation of inf1 method in sample class
This is implementation of inf2 method in sample class
This is implementation of inf3 method in sample class
```

Multiple inheritance

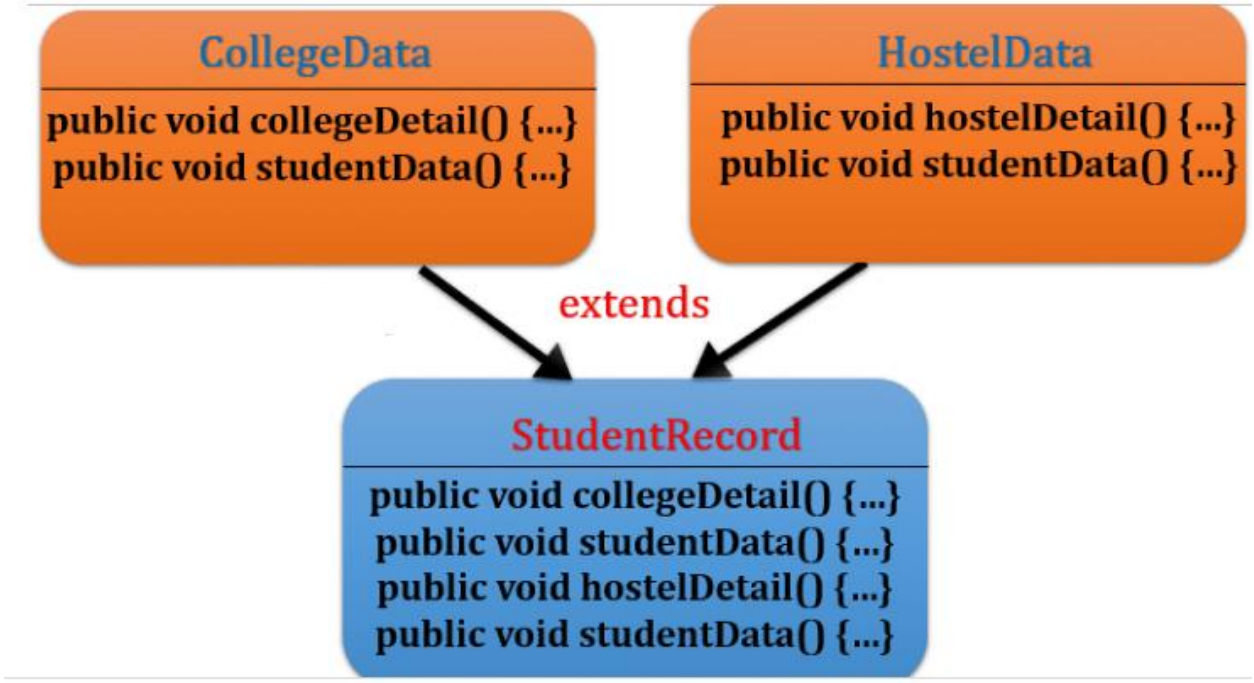
- When a class extending more than one class is known as multiple inheritance.
- **But Java doesn't allow** it because it creates the diamond problem and too complex to manage.



- Here **CollegeData** and **HostelData** are two classes that are extended by the **StudentRecord** class. This is known as multiple inheritances. But in java, it is not possible to implement it. We will learn what is the reason behind and **why java doesn't support multiple inheritances**.

Why Java doesn't support multiple inheritances?

- Let's say we have two classes **CollegeData** and **HostelData**. Both classes have student records and having some methods. Here we have two methods in each class and one method is common in both classes.
- **StudentRecord** is another class that extends both the classes(**CollegeData** and **HostelData**).
- It means the **StudentRecord** inherited all the methods of **CollegeData** and **HostelData.**



- **Output: Compilation error**

Example :

```
ClassCollegeData
{
VoidcollegeDetail()
{
System.out.println("College Name : DCSA");
System.out.println("College Grade : A");
System.out.println("University of College : KUK");
}
VoidstudentData()
{
System.out.println("courses of Student : MCA, MTECH, MBA,
BCA");
}
}
ClassHostelData
{
VoidhostelDetail()
{
System.out.println("Hostel Name : RAMA");
System.out.println("Hostel location : KUK");
}
```

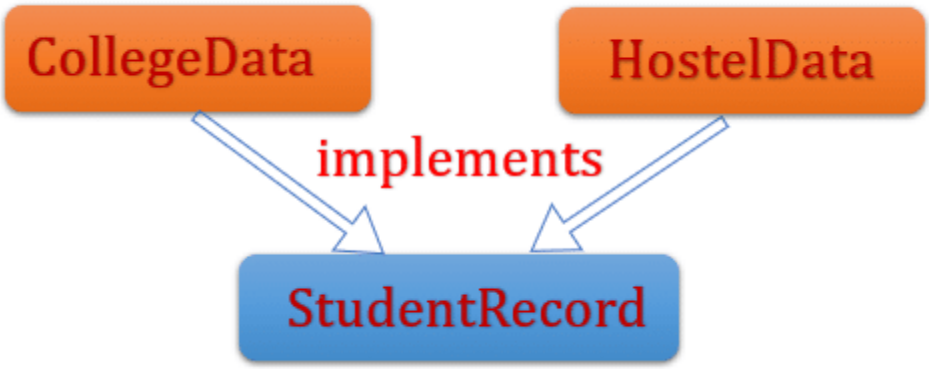
```
VoidstudentData()
{
System.out.println("Student selected on based : Percentage,
Financial condition");
}
}
StudentRecord extends CollegeData,HostelData
{
public static void main (String[] args)
{
StudentRecordobj = new StudentRecord();
obj.collegeDetail();
obj.studentData();
obj.hostelDetail();
obj.studentData();
}
}
```

Output:

Compilation error

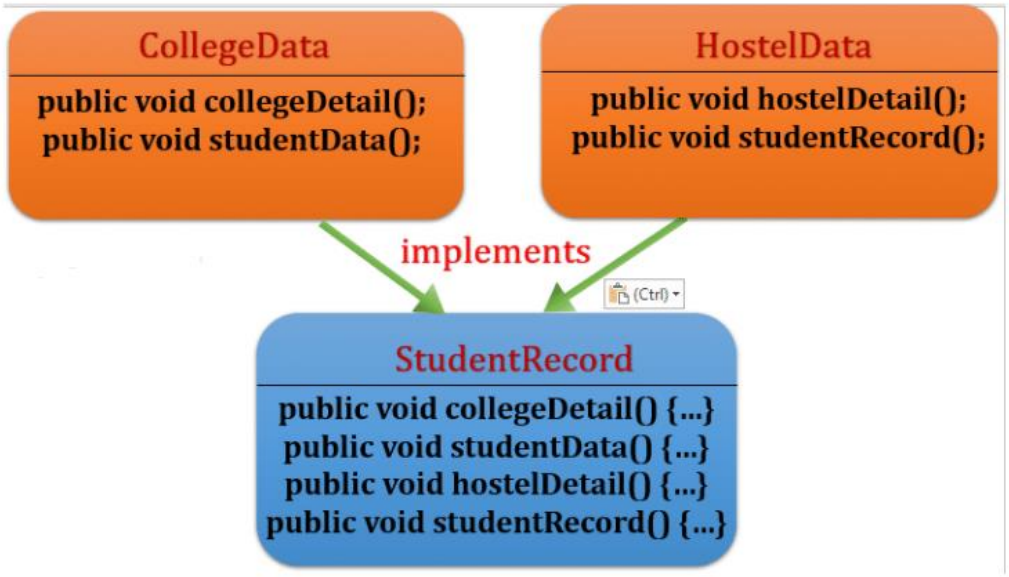
Multiple inheritance using interface in java

- We can achieve multiple inheritances by the use of interfaces.
- As you already know a class can implement any number of interfaces, but it can extend only one class.
- Before Java 8, Interfaces could have only abstract methods. It just defined the contract implementing by concrete classes.



- Here we will discuss it by two examples.
- **Firstly, we will see if both interfaces don't contain any common method.**
- It will not create any ambiguity.
- After that, if the interface has some common method then how to resolve the ambiguity.

- Let's discuss the first scenario when interfaces haven't any common method. Here we will create two interfaces and one concrete class that implements them and provide the implementation to the abstract method.



- Let's say we have two interfaces **CollegeData** and **HostelData**. Both classes declared some method, but they haven't any method. Here we have two methods in each class that inherited by concrete class.
- **StudentRecord** is a concrete class that implements both interfaces. It means the StudentRecord inherited all the methods and provide the implementation to the method of **CollegeData** and **HostelData** interface**.**

```
InterfaceCollegeData
{
VoidcollegeDetail();
VoidstudentData();
}
InterfaceHostelData
{
VoidhostelDetail();
VoidstudentRecord();
}
ClassStudentRecord implements CollegeData, HostelData
{
public void hostelDetail()
{
System.out.println("Hostel Name : RAMA");
System.out.println("Hostel location :guntur");
}
public void studentRecord()
{
System.out.println("Student selected on based : Percentage,
Financial condition");
```

```
}
public void collegeDetail()
{
System.out.println("College Name :rvr");
System.out.println("College Grade : A");
System.out.println("University of College : anu");
}
public void studentData()
{
System.out.println("courses of Student : MCA, MTECH, MBA,
BCA");
}
}
Classinterfacedemo
{
public static void main (String[] args)
{
StudentRecordobj = new StudentRecord();
obj.collegeDetail();
obj.studentData();
obj.hostelDetail();
obj.studentData();
```

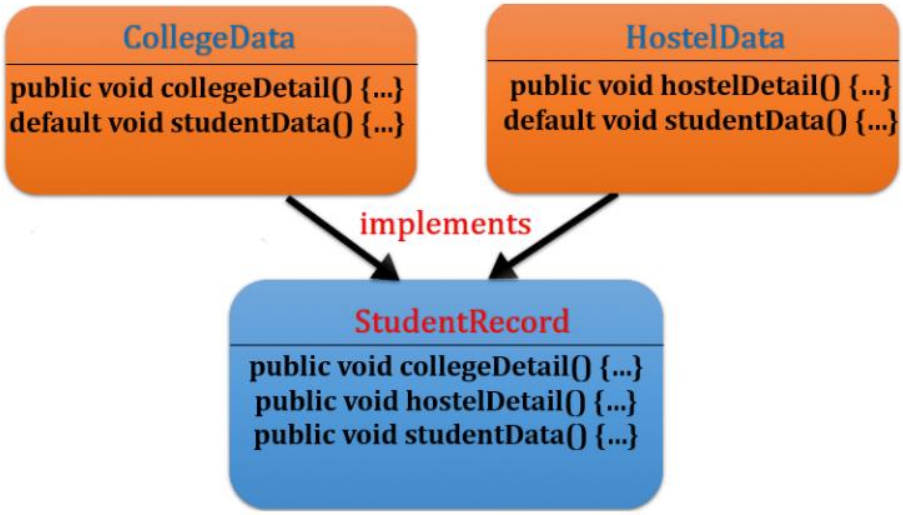
```
}
}
```

```
D:\>javac  interfacedemo.java

D:\>java  interfacedemo
College Name :rvr
College Grade : A
University of College : anu
courses of Student : MCA, MTECH, MBA, BCA
Hostel Name : RAMA
Hostel location : guntur
courses of Student : MCA, MTECH, MBA, BCA

D:\>
```

- Let's discuss the second scenario where we have a common method in both interfaces. When a concrete class implements both the interfaces and provides the implementation then how compiler take decision what method should be invoked?
- **<u>To resolve this problem we will use default methods that was introduced in Java 8. You can read the default method in detail.</u>**



```
InterfaceCollegeData
{
public void collegeDetail();
default void studentData()
{
System.out.println("courses of Student : MCA, MTECH, MBA,
BCA");
}
}
InterfaceHostelData
{
public void hostelDetail();
default void studentData()
{
System.out.println("Student selected on based : Percentage,
Financial condition");
}
}
ClassStudentRecord implements CollegeData, HostelData
{
public void hostelDetail()
{
```

```
System.out.println("Hostel Name : RAMA");
System.out.println("Hostel location : KUK");
}

public void collegeDetail()
{
System.out.println("College Name : DCSA");
System.out.println("College Grade : A");
System.out.println("University of College : KUK");
}
public void studentData()
{
CollegeData.super.studentData();
HostelData.super.studentData();
}
public static void main (String[] args)
{
StudentRecordobj = new StudentRecord();
obj.collegeDetail();
obj.hostelDetail();
obj.studentData();
}       }
```

```
D:\>javac  interfacedemo.java

D:\>java  interfacedemo
College Name :rvr
College Grade : A
University of College : anu
courses of Student : MCA, MTECH, MBA, BCA
Hostel Name : RAMA
Hostel location : guntur
courses of Student : MCA, MTECH, MBA, BCA

D:\>_
```

- In the above example to resolve the diamond problem**,** we are using the super keyword and overriding the method again. You can read the **diamond problem in detail**.

Variables in interface

- **The variable in an interface is public, static, and final by default.**
- **If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.**
- No access modifier is allowed except the public for interface variables.
- Every variable of an interface must be initialized in the interface itself.
- The class that implements an interface cannot modify the interface variable, but it may use as it defined in the interface.

Example:

```
Interfaceinfvar
{
//Public int a = 100;    //(or)
//Static int a = 100;
//(or)
//final int a=100;
//(or)
int a = 100;
//int b; // Error
}
ClassInfdemo implements infvar
{
public static void main(String[] args)
{
System.out.println("the value of a is = " + a);
// a = 150; // Can not be modified
}
}
```

Output:

The value of a is =100

Variable Argument (Varargs)

- **The varrags allows the method to accept zero or muliple arguments.**
- Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem.
- If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

- We don't have to provide overloaded methods so less code.
- Internally, the Varargs method is implemented by using the single dimensions arrays concept. Hence, in the Varrargs method, we can differentiate arguments by using Index. A variable-length argument is specified by three periods or dots(…).

```
return_type method_name(data_type... variableName)
{
}
```

- For Example,

```
public static void fun(int ... a)
{
 // method body
}
```

- This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here, a is implicitly declared as an array of type int[].

Example:

```
class Test1
{
static void fun(int... a)
{
System.out.println("Number of arguments: "+ a.length);
for (inti : a)
System.out.print(i + " ");
System.out.println();
}
public static void main(String args[])
{
fun(100);
fun(1, 2, 3, 4);
fun();
}
}
```

Output

Number of arguments: 1

100

Number of arguments: 4

1 2 3 4

Number of arguments: 0

- **Rules for varargs:**

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

- **There can be only one variable argument in the method.**
- **Variable argument (varargs) must be the last argument.**

- **Examples of varargs that fails to compile:**

```
void method(String... a, int... b){}//Compile time error
void method(int... a, String b){}//Compile time error
```

- **Example of Varargs that is the last argument in the method:**

```
class VarargsExample
{
static void display(int num, String... values)
{
System.out.println("number is "+num);
for(String s:values){
System.out.println(s);
}
}
```

```
public static void main(String args[]){
display(500,"hello");//one argument
display(1000,"my","name","is","varargs");//four arguments
}
}
```

Output

```
number is 500
hello
number is 1000
my
name
is
varargs
```