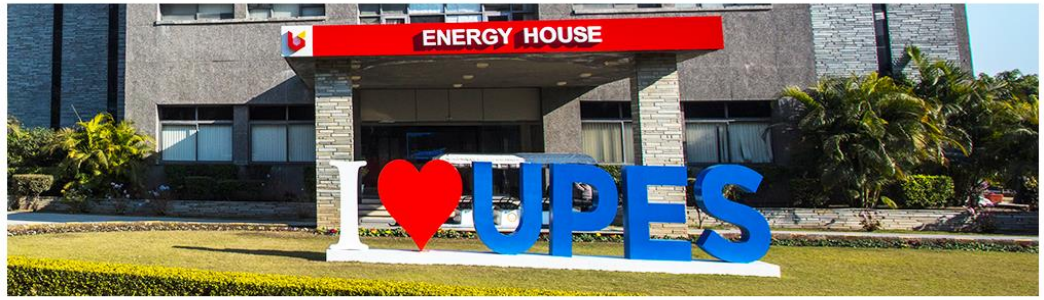



# UPES

UNIVERSITY WITH A PURPOSE

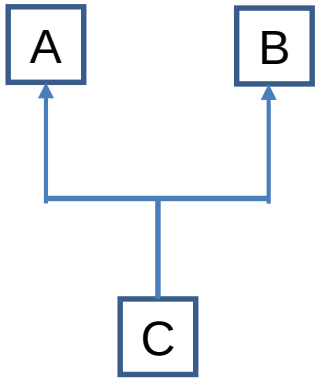
# Object Oriented Programming


UPES
UNIVERSITY WITH A PURPOSE

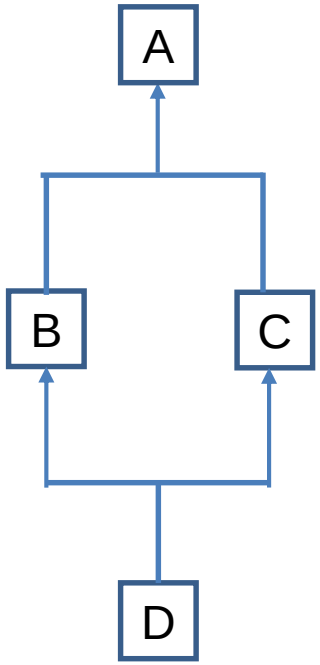
# Single Inheritance Vs Multiple Inheritance

- Java does not support multiple inheritance to avoid the ambiguity caused by it. One example of such problem is the diamond problem.


Single Inheritance




Multiple Inheritance




Diamond problem: hybrid Inheritance

UPES
UNIVERSITY WITH A PURPOSE

# Interface

- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- An interface like that of an abstract class cannot be instantiated.
- Interfaces do not have constructors.
- To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation.
- If a class that implements an interface, does not define all the methods, then it must be declared abstract and the method definitions should be provided by the subclass that extends the abstract class.

# Defining an Interface

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
//...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

**Note:**

- When no access modifier is included, then default access results, and the interface is only available to other members of the **package** in which it is declared.
- When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
- `name` is the name of the interface, and can be any valid identifier.
- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods.
- Each class that includes such an interface must **implement** all of the methods.

# Extending and Implementing

- Multiple extensions are allowed when extending interfaces i.e. one interface can extends none, one, or more interfaces.

```
class classname [extends superclass] [implements interface
[,interface...]] { // class-body }
```

- `Interface I1 {…}`
- `Interface I2 {…}`
- `Interface I3 extends I1, I2 {…}`
- `Class A implements I1 {…}`
- `Class B extends A implements I2, I3 {…}`

- We cannot create object of any interface but creation of object reference is possible.
- Object reference of interface can refer to any of its subclass types.

# Accessing Implementations Through Interface References

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- This is one of the key features of interfaces.
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the “callee.”

# Interface Example

```
interface Callback {
void callback(int param);  }

class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p); }  }

class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42); } }
```

**Note:** Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.

UPES
UNIVERSITY WITH A PURPOSE

# Partial Implementations

- If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**.

```
abstract class Incomplete implements Callback {
int a, b;
void show() {
System.out.println(a + " " + b);
}
//...
}
```

**Note:**

- Here, the class **Incomplete** does not implement **callback( )** and must be declared as **abstract**.
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

# Nested Interfaces

- An interface can be declared as a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**.
- This differs from a top-level interface, which must either be declared as **public** or use the default access level.
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.
- Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

# Nested Interface Example

```
// A nested interface example. This class contains a member interface.
class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);  }  }

class B implements A.NestedIF {// B implements the nested interface.
public boolean isNotNegative(int x) {
return x < 0 ? false: true;  }  }

class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be displayed");
}
}
```

UPES
UNIVERSITY WITH A PURPOSE

# Multiple Inheritance using Interface

```
Interface Printable{
Void print(); }
Interface Showable {
Void show();  }
Class A implements Printable, Showable{
Public void print(){
System.out.println(“Hello”); }
Public void show(){
System.out.pritln(“Welcome”); }
}

Public static void main(String []args){
A obj = new A();
Obj.print();
Obj.show();
}
```

Note: Multiple inheritance is supported by interface because its implementation is provided by the implementation class.

UPES
UNIVERSITY WITH A PURPOSE

# Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

```
interface A {
void meth1(); void meth2(); }
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3(); }
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1()."); }
public void meth2() {
System.out.println("Implement meth2()."); }
public void meth3() {
System.out.println("Implement meth3()."); } }
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1(); ob.meth2(); ob.meth3(); } }
```

# Variables in Interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as **constants**.

```
import java.util.Random;
interface SharedConstants {
int NO = 0; int YES = 1; int MAYBE = 2; int LATER = 3; int SOON = 4; int NEVER = 5; }
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2% } }
```

UPES
UNIVERSITY WITH A PURPOSE

# Variables in Interfaces

```
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO: System.out.println("No");
break;
case YES: System.out.println("Yes");
break;
case MAYBE: System.out.println("Maybe");
break;
case LATER: System.out.println("Later");
break;
case SOON: System.out.println("Soon");
break;
case NEVER: System.out.println("Never");
break; } }

public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask()); } }
```

UPES
UNIVERSITY WITH A PURPOSE

## References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.

UPES
UNIVERSITY WITH A PURPOSE