AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

19CSE204
Object Oriented Paradigm
2-0-3-3

Interface in Java

Interface in Java- Abstraction

- **Abstraction** is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user
  - **Abstract Class** was used for achieving **partial abstraction.**
  - An **interface** is used for **full abstraction**.

- An interface in Java is a **blueprint of a class.** It has static constants and abstract methods.
- The interface in Java 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.

Defining an Interface

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body

```
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;
}
```

***access* is either public or not used**

When no access specifier 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.

Implementing an interface

- Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

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

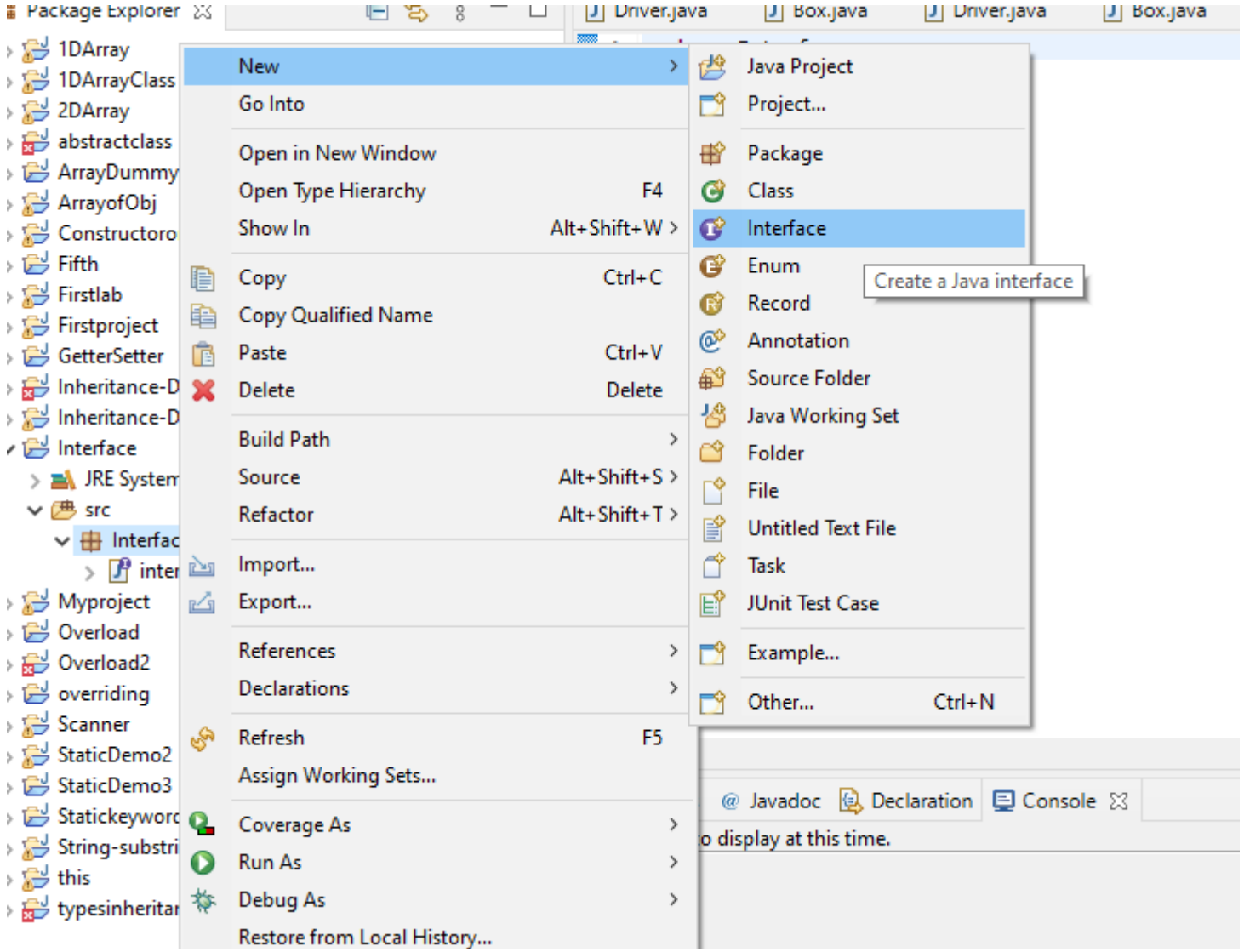
Here, *access* is either public or not used

- If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.

- The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Create a project

- Create a project
- Create a package
- Create an interface
- Create other class files if required
- Driver class



Example - Interface

```java
package Interface;

public interface Animal {
    /* File name : Animal.java */

        public void eat();
        public void travel();

}
```

```java
package Interface;

/* File name : MammalInt.java */
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

Output

Mammal eats

Mammal travels

- Inside a package named Interface, create an interface file, and define the Animal interface.
- Create a driver classs MammalInt and implement the interface

Predict the output

```
interface MyInterface{
  public void sample();
  public void display();
}
public class InterfaceExample implements MyInterface{
  public void sample(){
    System.out.println("Implementation of the sample method");
  }
  public static void main(String args[]) {
    InterfaceExample obj = new InterfaceExample();
    obj.sample();
  }
}
```

Compile-time error

```
InterfaceExample.java:5: error: InterfaceExample is not abstract and does not override abstract method display() in MyInterface
public class InterfaceExample implements MyInterface{
       ^
1 error
```

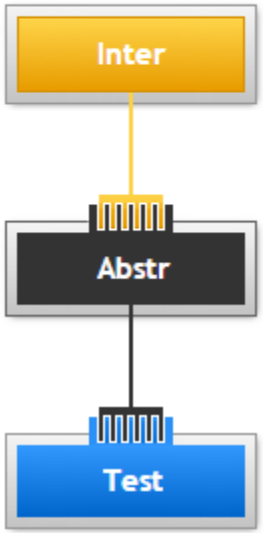
Once you implement an interface from a concrete class you need to provide implementation to all its methods.

But if you implement an interface from an abstract class, you need not provide full implementation of all the methods in the interface

But, If you still need to skip the implementation.

- **You can either provide a dummy implementation to the unwanted methods by throwing an exception such as UnsupportedOperationException or, IllegalStateException from them.**

```java
interface Inter {

    void methodOne();
}
abstract class Abstr implements Inter {

    void methodTwo()
    {
        System.out.println("MethodTwo Called");
    }
}
class Test extends Abstr {

    public void methodOne()
    {
        System.out.println("MethodOne Called");
    }
    void methodThree()
    {
        System.out.println("MethodThree Called");
    }
}
public class Javaapp {

    public static void main(String[] args) {

        Test t = new Test();
        t.methodOne();
        t.methodTwo();
        t.methodThree();
    }
}
```



An interface is different from a class

- 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.
  - 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.

Properties of an interface

- **Interfaces have the following properties –**
  - An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
  - Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  - Methods in an interface are implicitly public.

For implementation interfaces, there are several rules –

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class.

- The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
- The **Hockey interface has four methods**, but it inherits two from Sports;
- thus, a **class that implements Hockey needs to implement all six methods**.
- Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

```
// Filename: Sports.java
public interface Sports {
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}
// Filename: Football.java
public interface Football extends Sports {
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
// Filename: Hockey.java
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

Extending Multiple Interfaces

- The interface AnimalEat and AnimalTravel have one abstract method each
  - i.e. eat() and travel().
  - The class Animal implements the interfaces AnimalEat and AnimalTravel
- In the method main() in class Demo, an object a of class Animal is created. Then the methods eat() and travel() are called.

```
interface AnimalEat {
  void eat();
}
interface AnimalTravel {
  void travel();
}
class Animal implements AnimalEat, AnimalTravel {
  public void eat() {
    System.out.println("Animal is eating");
  }
  public void travel() {
    System.out.println("Animal is travelling");
  }
}
public class Demo {
  public static void main(String args[]) {
    Animal a = new Animal();
    a.eat();
    a.travel();
  }
}
```

Output

Animal is eating
Animal is travelling

Difference between Abstract Class and Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Applying Interfaces- Example (Stack)

Interface: IntStack.java

```
1  package applyinterface;
2
3  public interface IntStack {
4
5      void push(int item); // store an item
6      int pop(); // retrieve an item
7  }
8
```

- Here is the interface that defines an integer stack. **IntStack.java**. This interface will be used by both stack implementations.

- The program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

- Following is **DynStack** another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition.

FixedStack.Java

```
1  package applyinterface;
2      // An implementation of IntStack that uses fixed storag
3      class FixedStack implements IntStack {
4      private int stck[];
5      private int tos;
6      // allocate and initialize stack
7      FixedStack(int size) {
8      stck = new int[size];
9      tos = -1;
10     }
11     // Push an item onto the stack
12     public void push(int item) {
13     if(tos==stck.length-1) // use length member
14     System.out.println("Stack is full.");
15     else
16     stck[++tos] = item;
17     }
18     // Pop an item from the stack
19     public int pop() {
20     if(tos < 0) {
21     System.out.println("Stack underflow.");
22     return 0;
23     }
24     else
25     return stck[tos--];
26     }
27     }
```

DynStack.java

```java
package applyinterface;
    // Implement a "growable" stack.
    class DynStack implements IntStack {
    private int stck[];
    private int tos;
    // allocate and initialize stack
    DynStack(int size) {
    stck = new int[size];
    tos = -1;
    }
    // Push an item onto the stack
    public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
    int temp[] = new int[stck.length * 2]; // double size
    for(int i=0; i<stck.length; i++) temp[i] = stck[i];
    stck = temp;
    stck[++tos] = item;
    }
    else
    stck[++tos] = item;
    }
    // Pop an item from the stack
    public int pop() {
    if(tos < 0) {
    System.out.println("Stack underflow.");
    return 0;
    }
    else
    return stck[tos--];
    }
    }
```

Driver.java

```java
package applyinterface;

public class Driver {

    public static void main(String[] args) {
        // Dynamic Stack execution

        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);
        // these loops cause each stack to grow
        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);
        System.out.println("Stack in Dynamic mystack1:");
        for(int i=0; i<12; i++)
        System.out.println(mystack1.pop());
        System.out.println("Stack in Dynamic  mystack2:");
        for(int i=0; i<20; i++)
        System.out.println(mystack2.pop());


        // Fixed Stack execution
        FixedStack mystack3 = new FixedStack(5);
        FixedStack mystack4 = new FixedStack(8);
        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack3.push(i);
        for(int i=0; i<8; i++) mystack4.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in Static mystack1:");
        for(int i=0; i<5; i++)
        System.out.println(mystack3.pop());
        System.out.println("Stack in Static mystack2:");
        for(int i=0; i<8; i++)
        System.out.println(mystack4.pop());
    }
```

OUTPUT

```
Stack in Static mystack1:
4
3
2
1
0
Stack in Static mystack2:
7
6
5
4
3
2
1
0
```

```
Stack in Dynamic mystack1:
11
10
9
8
7
6
5
4
3
2
1
0
0
```

```
Stack in Dynamic  mystack2:
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
```

Namah Shivaya

Which should you use, abstract classes or interfaces?

- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.