
# Abstraction & Encapsulation

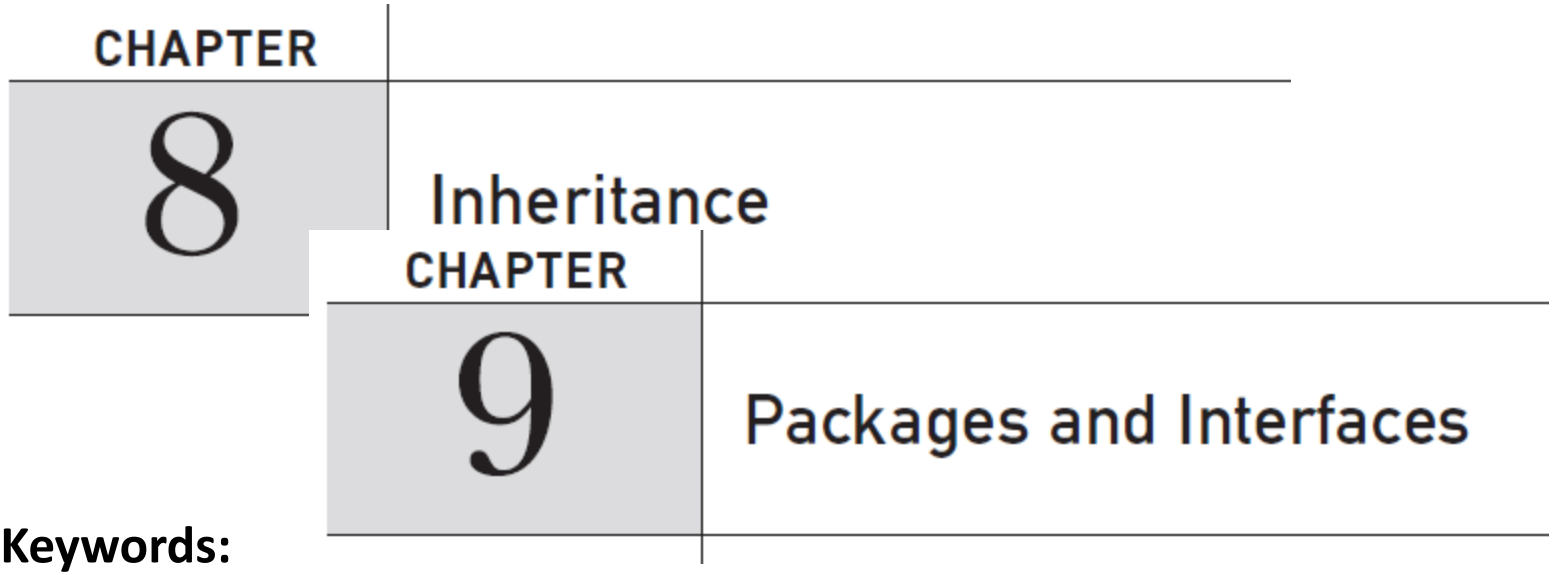

## Agenda:

- Abstract Class
- Interface in Java
- Abstract Vs Interface
- Java Encapsulation

**Md. Mamun Hossain**
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

# Reference Text : CHAPTER -8,9

## CH 8: Abstract Class & CH 9: Interface




**Keywords:**

**Using Abstract Classes**

**Interface**

**Abstract Class Vs Interface**

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Using Abstract Classes

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- **Uses of Abstraction in Java**
  - **Abstraction** is a process of hiding the implementation details and showing only functionality to the user i.e. it shows only essential things to the user and hides the internal details.
- **Example -** sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- There are two ways to achieve abstraction in java
  - Abstract class (0 to 100%)
  - Interface (100%)

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

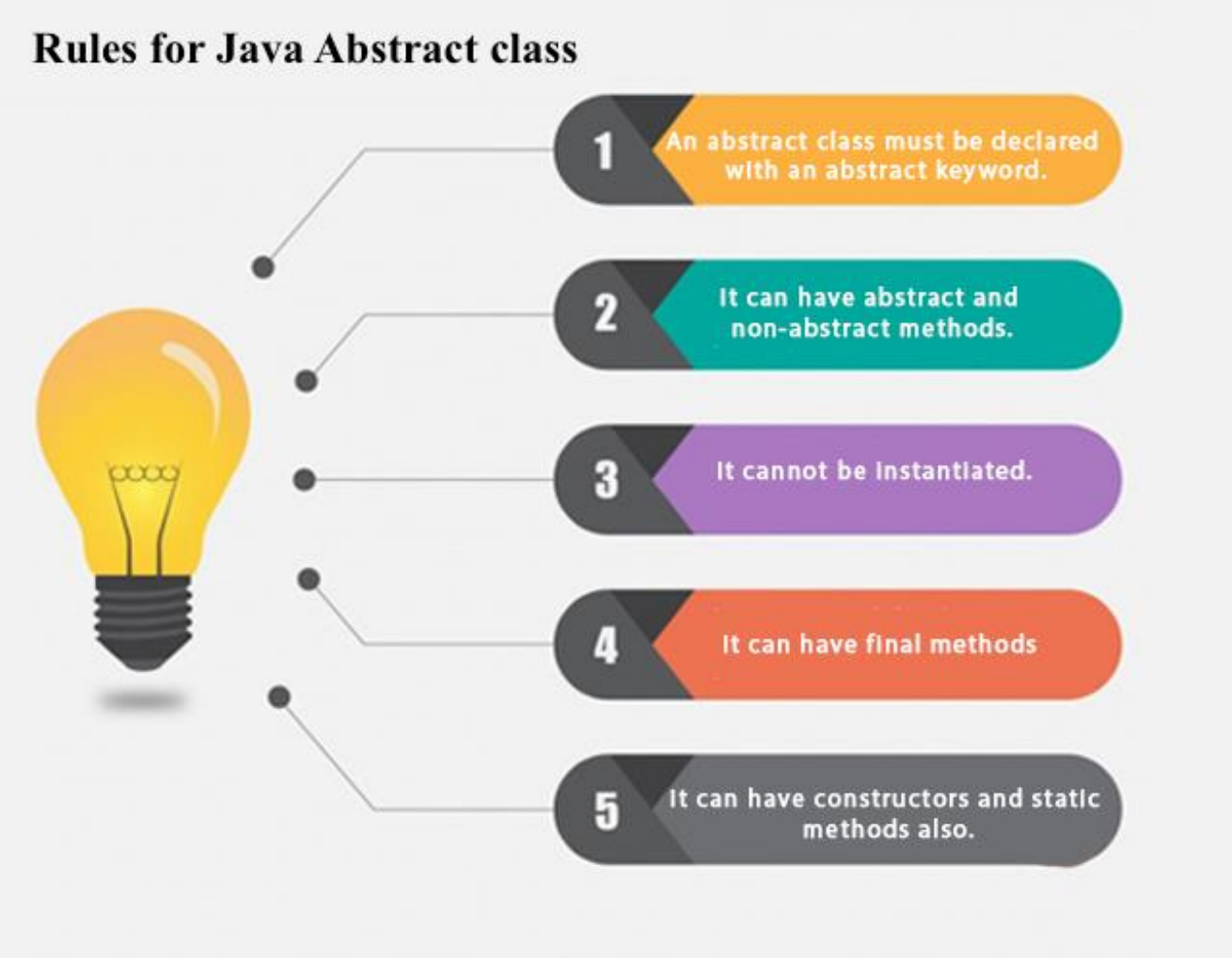
# Using Abstract Classes

**Points to Remember**

- ✓ An abstract class must be declared with an abstract keyword.
- ✓ It can have abstract and non-abstract methods.
- ✓ It cannot be instantiated.
- ✓ It can have constructors and static methods also.
- ✓ It can have final methods which will force the subclass not to change the body of the method.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract Class : Rule




Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class with abstract and non-abstract method

- In this example, Animal is an abstract class that contains one abstract method *Sound and a non-abstract method sleep*.

```
abstract class Animal {
        public abstract void Sound();
        public void sleep() {
                System.out.println("Zzz");
        }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class with abstract method

- In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class : Instantiate

- **You can not instantiate an abstract class**

```
abstract class A{
    abstract vois msg();
}

class B extenda A{
    vois msg(){… … …}
}
```

```
class Access{
    public static void main(String args[]){
        A ob1=new A(); // C.T Error
        B ob2=new B();
    }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Example: Abstract class

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class : Rules

- **Rule:** If there is an abstract method in a class, that class must be abstract.
- **Rule:** If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class : Rules

- **Rule:** If there is an abstract method in a class, that class must be abstract.

```
class Bike12{
abstract void run();
}
```

```
compile time error
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class : Rules

- **Rule:** If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

```
abstract class A{
    abstract vois msg();
}

class B extenda A{
    vois msg(){… … …}
}
```

```
abstract class A{
    abstract vois msg();
}

astract class B extenda A{
    vois show(){… … …}
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Abstract class : When & Why?

**Question:**

**Why And When To Use Abstract Classes and Methods?**

**Answer:**

To achieve security - hide certain details and only show the important details of an object.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java

Agenda:

- What is Interface?
- Example of Interface
- Multiple inheritance by Interface
- Why multiple inheritance is supported in Interface while it is not supported in case of class.
- Marker Interface
- **Nested Interface**

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java

- 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 100 % 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.

- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body except ***default*** methods.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java : Example

```
//Java interface
 interface Animal {
            // interface method
          public void animalSound(); // does not have a body)
            // interface method
          public void run(); does not have a body
}
```

To access the interface methods, the interface must be "implemented" (like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java

**Notes on Interfaces:**

- ✓ Java Interface also **represents the IS-A relationship**.
- ✓ It cannot be instantiated just like the abstract class.
- ✓ Interface methods do not have a body - the body is provided by the "implement" class
- ✓ On implementation of an interface, you must override all of its methods
- ✓ Interface methods are by default abstract and public
- ✓ Interface attributes are by default public, static and final
- ✓ An interface cannot contain a constructor (as it cannot be used to create objects)
- ✓ we can have **default and static methods** in an interface (Since Java 8,).
- ✓ we can have **private methods** in an interface (Since Java 9)
- ✓ we can have **static methods** in an interface

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java : Uses

There are mainly three reasons to use interface.

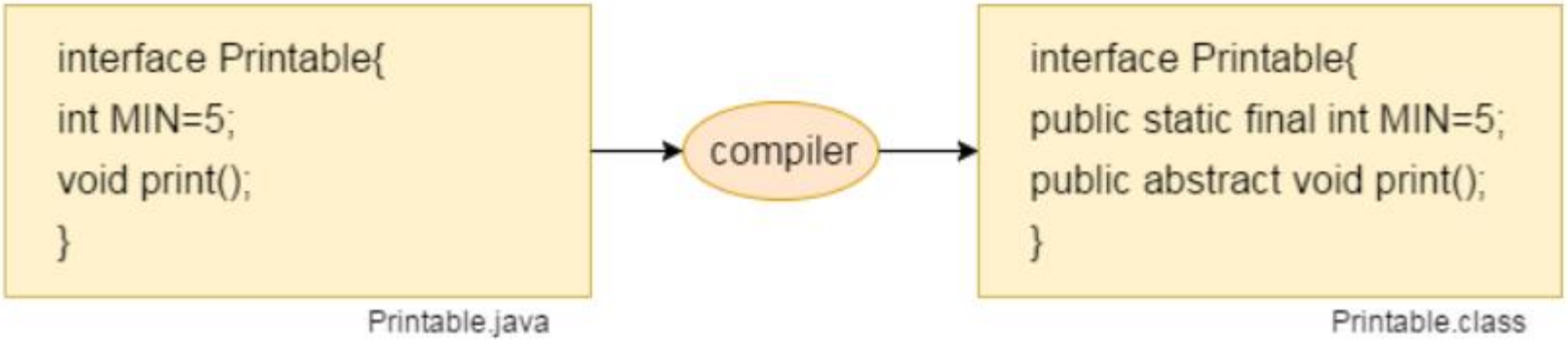
They are given below.

1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling (*).


***Loose coupling:** Loose coupling is an approach to interconnecting the components in a system so that those components or elements, depend on each other to the least extent practicable. Change in one element doesn't (or minor) change another element; means elements are mostly independent.

# Interface : Variable and method

- **Data Member: are public, static and final**
  - The Java compiler adds public, static and final keywords before data members.
- **Member Method: are public and static**
  - The Java compiler adds public and abstract keywords before the interface method.
- ***In other words***, Interface fields are public, static and final by default, and the methods are public and abstract.




Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java : Example

- An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface in Java : Uses

- In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes.

```
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

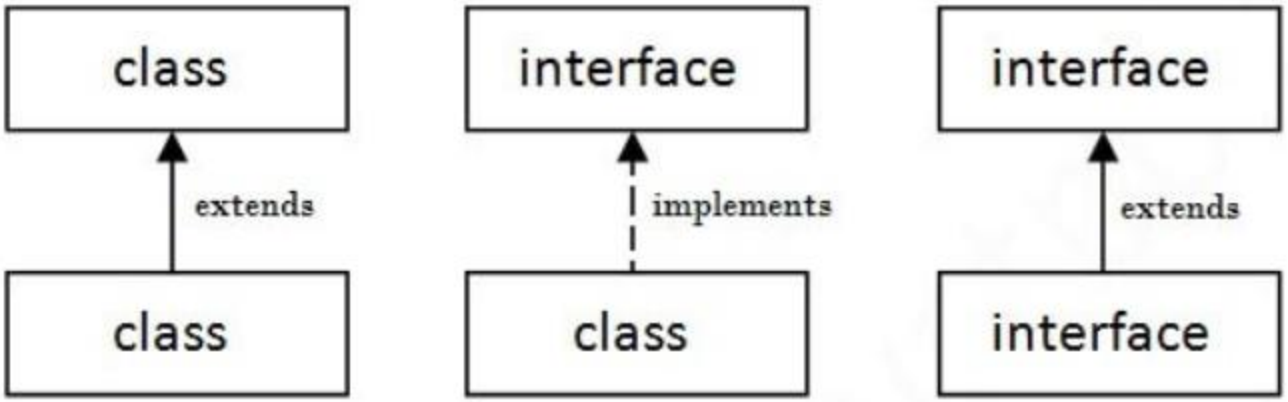
# Interface in Java :Example

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Relationship : classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.




Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.


**Multiple Inheritance in Java**

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Multiple inheritance in Java by interface

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

```
Output:Hello
       Welcome
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Multiple inheritance in Java by interface

- **Multiple inheritance is not supported through class in java, but it is possible by an interface, why?**

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface: Default Method

- Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

# Interface: Static Method

Since Java 8, we can have static method in interface. Let's see an example:

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface: maker or tagger

- What is marker or tagged interface?

  An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Nested Interface

- What is  nested interface?

  An interface can have another interface which is known as a nested interface.

```
interface printable{
 void print();
 interface MessagePrintable{
   void msg();
 }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface : When & Why?

**Question:**

**Why And When To Use Interface?**

**Answer:**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma (see example below).

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

| 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 class**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:**<br>public abstract class Shape{<br>public abstract void draw();<br>} | **Example:**<br>public interface Drawable{<br>void draw();<br>} |

# Example : abstract class and interface

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the imple
abstract class B implements A{
public void c(){System.out.println("I am C");}
}
```

Output:

```
I am a
I am b
I am c
I am d
```

```
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Coupling : Loose & Tight

- **What is Coupling?**

  **Coupling** refers to the degree of direct knowledge that one element has of another.

**What is Loose Coupling?**

- **Loose coupling** is an approach to interconnecting the components in a system or network so that those components, also called elements, depend on each other to the least extent practicable.

**Goal of Loose Coupling**

- The goal of a loose coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Coupling In Java

- **Coupling** - Coupling refers to the usage of an object by another object. It can also be termed as collaboration.

This dependency of one object on another object to get some task done can be classified into the following two types −

- **Tight coupling** - When an object creates the object to be used, then it is a tight coupling situation. As the main object creates the object itself, this object can not be changed from outside world easily marked it as tightly coupled objects.
- **Loose coupling** - When an object gets the object to be used from the outside, then it is a loose coupling situation. As the main object is merely using the object, this object can be changed from the outside world easily marked it as loosely coupled objects.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Tight Coupling - Example

**Tight coupling :** In general, Tight coupling means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

**Example** : If you want to change the skin, you would also have to change the design of your body as well because the two are joined together – they are tightly coupled.
The best example of tight coupling is RMI(Remote Method Invocation).

```
class Subject {
    Topic t = new Topic();
    public void startReading()
    {
        t.understand();
    }
}
class Topic {
    public void understand()
    {
        System.out.println("Tight coupling concept");
    }
}
```

*In the above program Subject class is tightly coupled with Topic class it means if any change in the Topic class requires Subject class to change.*

# Tight Coupling - Example

**Tight coupling :** Another Example

```
class Volume
{
      public static void main(String args[])
      {
          Box b = new Box(5,5,5);
          System.out.println(b.volume);
      }
}
class Box
{
      public int volume;
      Box(int length, int width, int height)
      {
          this.volume = length * width * height;
      }
}
```

**Explanation:** In the above example, there is a strong inter-dependency between both the classes. If there is any change in Box class then they reflects in the result of Class Volume.

# Loose Coupling - Example

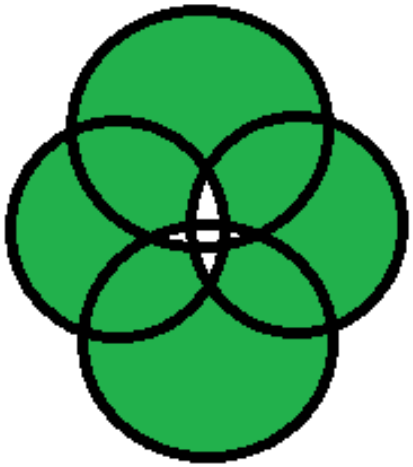
**Loose coupling :** In simple words, loose coupling means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled

**Example :** If you change your shirt, then you are not forced to change your body – when you can do that, then you have loose coupling.

**Explanation :** In the above example, ***Topic1*** and ***Topic2*** objects are loosely coupled. It means Topic is an interface and we can inject any of the implemented classes at run time and we can provide service to the end user.

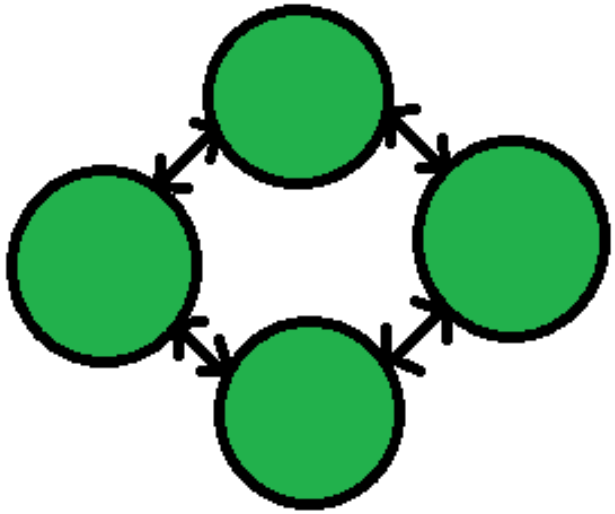
```
public interface Topic
{
    void understand();
}
class Topic1 implements Topic {
public void understand()
    {
        System.out.println("Got it");
    }
} class Topic2 implements Topic {
public void understand()
    {
        System.out.println("understand");
    }
} public class Subject {
public static void main(String[] args)
    {
        Topic t = new Topic1();
        t.understand();
    }
}
```

# Interface : Tight Coupling Vs. Loose Coupling



Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:

1. Less Interdependency
2. Less coordination
3. Less information flow

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface : Tight Coupling Vs. Loose Coupling

- Tight coupling is not good at the test-ability where as loose coupling improves the test ability.
- Tight coupling does not provide the concept of interface. But loose coupling help follow the GOF (*Gang of Four*) principle (*Elements of Reusable Objec*t) of program to interfaces, not implementations.
- In Tight coupling, it is not easy to swap the codes between two classes. But it's much easier to swap other pieces of code/modules/objects/components in loose coupling.
- Tight coupling does not have the changing capability. But loose coupling is highly changeable.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Interface : Tight Coupling or Loose Coupling?

- **Which is better tight coupling or loose coupling?**

- **In general, Tight Coupling is bad**
  - most of the time, because it reduces flexibility and re-usability of code, it makes changes much more difficult, it impedes test ability etc.
- **Loose coupling is a better choice**
  - A loosely coupled will help you when your application need to change or grow. If you design with loosely coupled architecture, only a few parts of the application should be affected when requirements change.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Week 8, Lecture 3

- **Java Encapsulation**
  - Private data members
  - Getter & Setter Methods
  - Read Only methods
  - Write Only methods

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation : What?

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java ***by making all the data members of the class private***. Now we can use ***setter*** and ***getter*** methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class


Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation : Advantages

- **read-only or write-only class** :By providing only a setter or getter method, you can make the class ***read-only or write-only***.
- **control** :It provides you the control over the data.
  - Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
- **data hiding :** It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.
- **unit testing :** The encapsulate class is easy to test. So, it is better for unit testing.
- **easy and fast**: The standard IDE's are providing the facility to generate the getters and setters. So, it is easy and fast to create an encapsulated class in Java.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation : How?

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as private
- provide public **get** and **set** methods to access and update the value of a private variable

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Get and Set

- **Get and Set**

You learned from the previous classes (access control) that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods.

- The get method returns the variable value, and the set method sets the value.

- Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Get and Set

**Example explained**

- *The get method returns the value of the variable name.*
- *The set method takes a parameter (newName) and assigns it to the name variable. The this keyword is used to refer to the current object.*

```java
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

*However, as the name variable is declared as private, we **cannot** access it from outside this class.*

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Get and Set

```java
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

*As the name variable is declared as private, we **cannot** access it from outside this class.*

```java
public class MyClass {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";  // error
    System.out.println(myObj.name); // error
  }
}
```

But if the variable was declared as public, we would expect the output as: **John**

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Get and Set

```java
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

```java
public class MyClass {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("John"); // Set the value of the name variable to "John"
    System.out.println(myObj.getName());
  }
}

// Outputs "John"
```

Instead, we use the getName() and setName() methods to acccess and update the variable.

Md. Mamun Hossain
B. Sc. (Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Read Only Class

- **A Java class which has only getter methods.**

```
public class Student{

//private data member
        private String college="BAUST";
//getter method for college
    public String getCollege(){
                return college;
    }
}
```

**Student s=new Student();**

Now, you can't change the value of the college data member which is "AKG".

```
s.setCollege("KITE");//will render compile time error
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Write Only Class

- **A Java class which has only setter methods.**

```
public class Student{
    //private data member
        private String college;
    //getter method for college
            public void setCollege(String college){
                this.college=college;
            }
}
```

Now, you can't get the value of the college
But you can change the value of college data member.

**Student s=new Student();**

```
System.out.println(s.getCollege());//Compile Time Error, because there is no such method
System.out.println(s.college);//Compile Time Error, because the college data member is private.
//So, it can't be accessed from outside the class
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Encapsulation: Why?

- **Why Encapsulation?**
  - Better control of class attributes and methods
  - Class attributes can be made
    - **read-only** (if you only use the get method), or
    - **write-only** (if you only use the set method)
  - Flexible: the programmer can change one part of the code without affecting other parts Increased security of data

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Miscellaneous

## Two uncommon keywords/operators

- **instanceof : Compare between objects**
  - Upcasting and Downcasting
- **strictfp : force to keep same format for floating-point variables across platforms**

| | | | | |
|---|---|---|---|---|
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

# *instanceof* **operator**

- **Java instanceof**
- The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Miscellaneous : instanceof - Example

Let's see the simple example of instance operator where it tests the current class.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Miscellaneous : instanceof - Example

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal


 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
 }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 }
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Down casting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Possibility of Downcasting with and without instanceof

## Downcasting with instanceof

```
class Animal { }

class Dog3 extends Animal {
  static void method(Animal a) {
    if(a instanceof Dog3){
      Dog3 d=(Dog3)a;//downcasting
      System.out.println("ok downcasting performed");
    }
  }

  public static void main (String [] args) {
    Animal a=new Dog3();
    Dog3.method(a);
  }

}
```

## Downcasting **without instanceof**

```
class Animal { }
class Dog4 extends Animal {
  static void method(Animal a) {
      Dog4 d=(Dog4)a;//downcasting
      System.out.println("ok downcasting performed");
  }
   public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
  }
}
```

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Strictfp keyword

- Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable.
- The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform.
- So, now you have better control over the floating-point arithmetic.

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

```
strictfp class A{}//strictfp applied on class
```

```
strictfp interface M{}//strictfp applied on interface
```

```
class A{
strictfp void m(){}//strictfp applied on method
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Illegal code for strictfp keyword

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

```
class B{
strictfp abstract void m();//Illegal combination of modifiers
}
```

```
class B{
strictfp int data=10;//modifier strictfp not allowed here
}
```

```
class B{
strictfp B(){}//modifier strictfp not allowed here
}
```

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

# Gang of Four (GOF)

- **What is Gang of Four (GOF)?**

In 1994, four authors ***Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides*** published a book titled ***Design Patterns - Elements of Reusable Object-Oriented Software*** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as ***Gang of Four (GOF).*** According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Md. Mamun Hossain
B. Sc. ( Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST