ALL EXAMPLES MUST HAVE 2 CLASSES. DRIVER, AND IMPLEMENTATION

Java Abstraction

Abstraction **refers to the ability to make a class abstract in OOP**.
An a**bstract class is one that cannot be instantiated**. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner.
You just cannot create an instance of the abstract class.
If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

```
public abstract class Bike{
   abstract void run();
 }
```

```
  public  class Honda4 extends Bike{
        void run()
         {  System.out.println("running safely..");
         }

public static void main(String args[])
 {
 Bike obj = new Honda4();
 obj.run();
}
}
```

Abstract Methods:

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract. The abstract keyword is also used to declare a method as abstract. An abstract method consists of a method signature, but no method body.

The abstract keyword is also used to declare a method as abstract. An abstract method consists of a method signature, **but no method body.**

Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces

Declaring a method as abstract has two results:

• The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
• Any child class must either override the abstract method or declare itself abstract.
A child class that inherits an abstract method must override it. If they do not, they must be abstract and any of their children must override it.
Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Java Encapsulation

Encapsulation is the technique of **making the fields in a class private and providing access to the fields via public methods.** If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.

For this reason, encapsulation is also referred to as data hiding.
Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The **main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code.** With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

```
public class EncapTest{
private String name;
private String idNum;
private int age;


public int getAge(){
return age;
}
public String getName(){
return name;
}
public String getIdNum(){
return idNum;
}
public void setAge(int newAge){
age = newAge;
}
public void setName(String newName){

name = newName;
}
public void setIdNum(String newId){
```

```
idNum = newId;
}
}


public class RunEncap{
public static void main(String args[]){

EncapTest encap =new EncapTest();
encap.setName("James");
encap.setAge(20);
encap.setIdNum("12343ms");
System.out.print("Name : "+ encap.getName()+" Age : "+ encap.getAge());
}
}
```

Benefits of Encapsulation:

The fields of a class can be made read-only or write-only.

- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

Java Interfaces

A n interface is a **collection of abstract methods**. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts.

- A class describes the attributes and behaviors of an object. 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 bytecode 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.
• 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.

Declaring Interfaces:

The interface keyword is used to declare an interface. Here is a simple example to declare an interface:

```
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
//Any number of final, static fields
//Any number of abstract method declarations\
}
```

Interfaces have the following properties:

• An **interface is implicitly abstract. You do not need to use the abstract keyword when 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.**

```
interface Animal{
public void eat();
public void travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform **all** the behaviors of the interface, the class must declare itself as abstract.

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
public class MammalInt implements Animal{
public void eat(){
System.out.println("Mammal eats");
}
public void travel(){
System.out.println("Mammal travels");
}
public int noOfLegs(){
return0;
}
```

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

When overriding methods defined in interfaces there are several rules to be followed:

• Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
• The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
• An implementation class itself can be abstract and if so interface methods need not be implemented.

When 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, similarly to the way that a class can extend another class.

Extending Interfaces

An interface can extend another interface, similarly to the 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.

```
//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);
}
```

Extending Multiple Interfaces:

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. For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports,Event
```

----------EXTRA MINE---MULTI IMPLEMENTATION

```
public class Hockey implements Sports,Event
```