Practical Week 1

Abstract Classes in Java
Abstract classes and Abstract methods
Interfaces in Java

Run below coding at https://www.onlinegdb.com/

Abstract class:
is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method:
can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class.

For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods.
For example,

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same `abstract` keyword to create abstract methods. For example,

```
abstract void display();
```

Here, `display()` is an abstract method. The body of `display()` is replaced by `;`.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

  // abstract method
  abstract void method1();
}
```

Example: Java Abstract Class and Method

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming");
  }
}

class Main extends Language {

  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

In the above example, we have created an abstract class named `Language`. The class contains a regular method `display()`.

We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```

Here, `obj` is the object of the child class `Main`. We are calling the method of the abstract class using the object `obj`.

Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method.

For example,

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

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

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
I can eat.
```

In the example, we have created an abstract class `Animal`.

The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.

We have inherited a subclass `Dog` from the superclass `Animal`.

Here, the subclass `Dog` provides the implementation for the abstract method `makeSound()`.

We then used the object `d1` of the `Dog` class to call methods `makeSound()` and `eat()`.

> **Note:** If the `Dog` class doesn't provide the implementation of the abstract method `makeSound()`, `Dog` should also be declared as abstract. This is because the subclass `Dog` inherits `makeSound()` from `Animal`.

Accesses Constructor of Abstract Classes

An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the `super` keyword. For example,

```
abstract class Animal {
   Animal() {
      ....
   }
}

class Dog extends Animal {
   Dog() {
      super();
      ...
   }
}
```

Here, we have used the `super()` inside the constructor of `Dog` to access the constructor of the `Animal`.

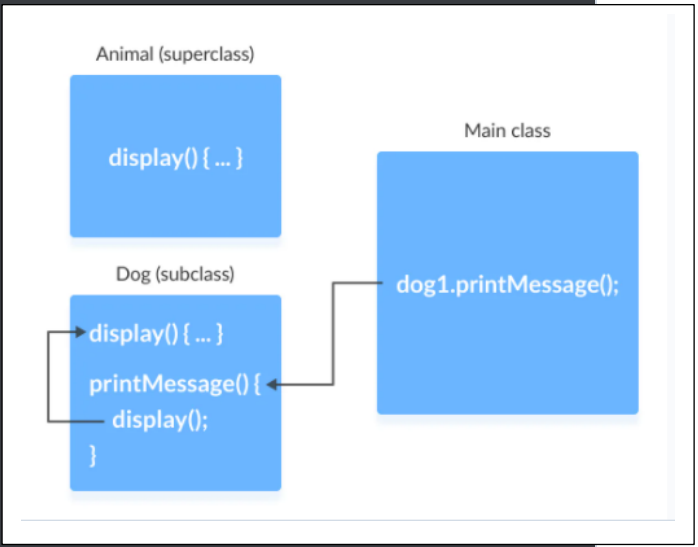
Note that the `super` should always be the first statement of the subclass constructor. Visit Java super keyword to learn more.

Access Overridden Methods of the superclass

If methods with the same name are defined in both superclass and subclass, the method in the subclass overrides the method in the superclass. This is called method overriding.

Example: Method overriding

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```



In this example, by making an object `dog1` of `Dog` class, we can call its method `printMessage()` which then executes the `display()` statement.

Since `display()` is defined in both the classes, the method of subclass `Dog` overrides the method of superclass `Animal`.

Hence, the `display()` of the subclass is called.

Output

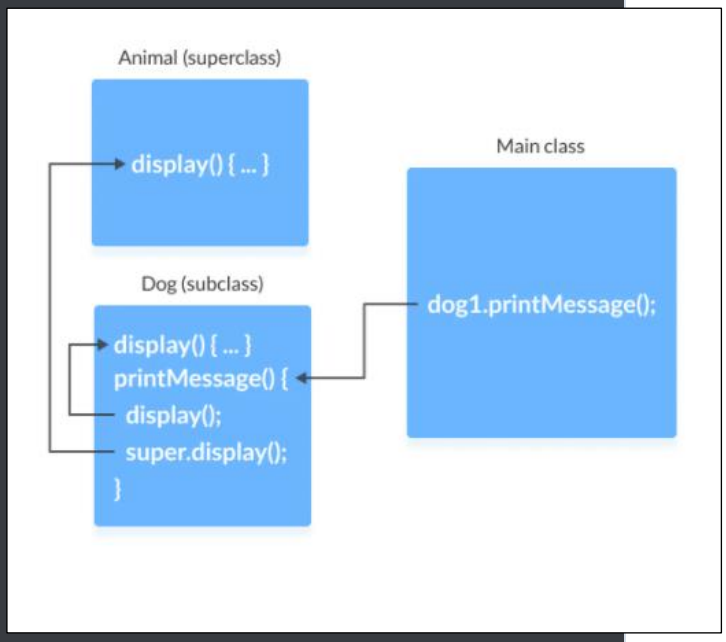
```
I am a dog
```

What if the overridden method of the superclass has to be called?

We use `super.display()` if the overridden method `display()` of superclass `Animal` needs to be called.

Example: super to Call Superclass Method

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```



Output

```
I am a dog
I am an animal
```

Access Attributes of the Superclass

The superclass and subclass can have attributes with the same name. We use the `super` keyword to access the attribute of the superclass.

Example: Access superclass attribute

```
class Animal {
  protected String type="animal";
}

class Dog extends Animal {
  public String type="mammal";

  public void printType() {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

In this example, we have defined the same instance field `type` in both the superclass `Animal` and the subclass `Dog`.

We then created an object `dog1` of the `Dog` class.

Then, the `printType()` method is called using this object.

Inside the `printType()` function,

- `type` refers to the attribute of the subclass `Dog`.
- `super.type` refers to the attribute of the superclass Animal.

Output:

```
I am a mammal
I am an animal
```

Example: Java Abstraction

```java
abstract class MotorBike {
  abstract void brake();
}

class SportsBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("SportsBike Brake");
  }
}

class MountainBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("MountainBike Brake");
  }
}

class Main {
  public static void main(String[] args) {
    MountainBike m1 = new MountainBike();
    m1.brake();
    SportsBike s1 = new SportsBike();
    s1.brake();
  }
}
```

In the example, we have created an abstract super class `MotorBike`. The superclass `MotorBike` has an abstract method `brake()`.

The `brake()` method cannot be implemented inside `MotorBike`. It is because every bike has different implementation of brakes. So, all the subclasses of `MotorBike` would have different implementation of `brake()`.

So, the implementation of `brake()` in `MotorBike` is kept hidden.

Here, `MountainBike` makes its own implementation of `brake()` and `SportsBike` makes its own implementation of `brake()`.

Output:

```
MountainBike Brake
SportsBike Brake
```

https://www.programiz.com/java-programming/super-keyword

Abstract Classes in Java

Question 1
Run below coding one by one and do summary based on coding below.

Summary the output and make your own note **about abstract classes in Java**		
	Output	**Remark/Note**
1	**Derived fun() called**	
2		
3		
4		
5		
6		

Following are some important observations about abstract classes in Java.

1) An instance of an abstract class cannot be created, we can have references to abstract class type though.

```
abstract class Base {
   abstract void fun();
}
class Derived extends Base {
   void fun()
   {
      System.out.println("Derived fun() called");
   }
}
class Main {
   public static void main(String args[])
   {

      // Uncommenting the following line will cause
      // compiler error as the line tries to create an
      // instance of abstract class.
      // Base b = new Base();

      // We can have references of Base type.
      Base b = new Derived();
      b.fun();
   }
}
```

2) An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created. For example, the following is a valid Java program.

```
// An abstract class with constructor
abstract class Base {
    Base()
    {
        System.out.println("Base Constructor Called");
    }
    abstract void fun();
}
class Derived extends Base {
    Derived()
    {
        System.out.println("Derived Constructor Called");
    }
    void fun()
    {
        System.out.println("Derived fun() called");
    }
}
class Main {
    public static void main(String args[])
    {
        Derived d = new Derived();
    }
}
```

3) We can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base {
}

class Main {
    public static void main(String args[])
    {
        Derived d = new Derived();
        d.fun();
    }
}
```

4) Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base {
    final void fun()
    {
        System.out.println("Derived fun() called");
    }
}

class Derived extends Base {
}

class Main {
    public static void main(String args[])
    {
        Base b = new Derived();
        b.fun();
    }
}
```

5) For any abstract java class we are not allowed to create an object i.e., for abstract class instantiation is not possible.

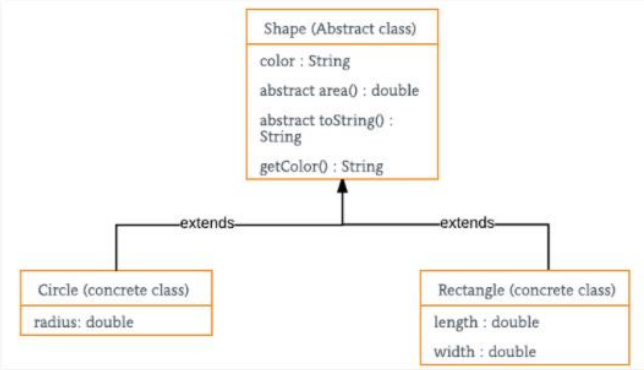
```
// An abstract class example
abstract class Test {
    public static void main(String args[])
    {
        // Try to create an object
        Test t = new Test();
    }
}
```

6) Similar to the interface we can define static methods in an abstract class that can be called independently without an object.

```
abstract class Party {
    static void doParty()
    {
        System.out.println("Lets have some fun!!");
    }
}

public class Main extends Party {
    public static void main(String[] args)
    {
        Party.doParty();
    }
}
```

Abstract classes and Abstract methods:

- **An abstract class is a class that is declared with abstract keyword.**
- **An abstract method is a method that is declared without an implementation.**
- **An abstract class may or may not have all abstract methods. Some of them can be concrete methods**
- **A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.**
- **Any class that contains one or more abstract methods must also be declared with abstract keyword.**
- **There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.**
- **An abstract class can have parametrized constructors and default constructor is always present in an abstract class.**



Question 2

(a) Run below coding and shown us the output.

```
// Java program to illustrate the
// concept of Abstraction
abstract class Shape {
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have constructor
    public Shape(String color)
    {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() { return color; }
}
class Circle extends Shape {
    double radius;
```

```
    public Circle(String color, double radius)
    {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override public String toString()
    {
        return "Circle color is " + super.getColor()
            + "and area is : " + area();
    }
}
class Rectangle extends Shape {

    double length;
    double width;

    public Rectangle(String color, double length,
                     double width)
    {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override double area() { return length * width; }

    @Override public String toString()
    {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}
public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

(b) Summary in table **about abstract classes and abstract methods in Java** based on above yellow notes and coding output Q2 (a).

Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

Question 3.
Run and study coding below.

```
**********
import java.io.*;

    interface Vehicle {

       // all are the abstract methods.
       void changeGear(int a);
       void speedUp(int a);
       void applyBrakes(int a);
    }

    class Bicycle implements Vehicle{

       int speed;
       int gear;

        // to change gear
       @Override
       public void changeGear(int newGear){

          gear = newGear;
       }

       // to increase speed
       @Override
       public void speedUp(int increment){

          speed = speed + increment;
       }

       // to decrease speed
       @Override
       public void applyBrakes(int decrement){

          speed = speed - decrement;
       }

       public void printStates() {
           System.out.println("speed: " + speed
              + " gear: " + gear);
```

```
        }
    }

class GFG {

    public static void main (String[] args) {

        // creating an inatance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        // creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

a) What is the output for Q2?
b) Extend the above coding by adding class Bike implements Vechicle.
c) What is the output for Q2 (b)?